Разработка встроенных операционных систем реального времени

Версия 1.1

Права на данный текст в полном объёме принадлежат Компании ЭРЕМЕКС и защищены законодательством Российской Федерации об авторском праве и международными договорами.
Никакая часть настоящей книги ни в каких целях не может быть воспроизведена в какой бы
то ни было форме и какими бы то ни было средствами, будь то электронные или механические,
включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения Компании ЭРЕМЕКС.
Компания ЭРЕМЕКС оставляет за собой право изменить содержание данного документа в любое время без предварительного уведомления.
Компания ЭРЕМЕКС не несёт ответственности за содержание, качество, актуальность и достоверность данного документа и используемых в документе материалов, права на которые принадлежат другим правообладателям, а также за возможный ущерб, связанный с использованием данного документа и содержащихся в нём материалов.
Обозначения ЭРЕМЕКС, EREMEX, FX-RTOS, Delta Design, TopoR, SimOne являются товарными знаками Компании ЭРЕМЕКС.
Остальные упомянутые в документе торговые марки являются собственностью их законных владельцев.
© Компания ЭРЕМЕКС, 2020 г. Все права защищены.

Содержание

Предисловие 8
Зачем понадобилась еще одна книга об операционных системах 8
Обзор содержания 9
Аудитория 10
История FX-RTOS 10
1 Основные концепции 12
1.1 Встроенные системы 12
1.2 Системы реального времени 15
1.3 Детерминизм 15
1.4 Резюме 16
2 Обзор аппаратного обеспечения 17
2.1 Классическая архитектура 17
2.1.1 Абстрактный процессор 17
2.1.2 Иерархия памяти 19
2.1.3 Ввод-вывод 22
2.1.4 Прерывания 24
2.1.5 Исключения 27
2.1.6 Атомарность 27
2.2 Многопроцессорные системы 29
2.2.1 Ложное разделение данных 31
2.2.2 Прерывания в многопроцессорных системах 31
2.2.3 Атомарность в многопроцессорных системах 32
2.2.4 Неблокирующие алгоритмы 35
2.2.5 Модель памяти 38
2.3 Защита 40
2.3.1 Режимы исполнения 40
2.3.2 Переключения между режимами 41
2.3.3 Права доступа к памяти 42
2.3.4 Инвертированные таблицы страниц 46
2.4 Типичный микроконтроллер 47
2.5 Резюме 48
3 Подходы к проектированию встроенного ПО 49
3.1 Требования 49
3.2 Реализация управляющей логики в приложении 49

3.2.1 Главный цикл........49
3.2.2 Циклический планировщик........51
3.2.3 Синхронизированный цикл........52
3.2.4 Прерывания........54
3.3 Использование ОСРВ........55
3.3.1 Акторы........55
3.3.2 Потоки........57
3.3.3 Процессы........59
3.3.4 Задачи ОС........60
3.3.5 Архитектуры ОС........61
3.4 Резюме........64
4 Абстракция оборудования и переносимость........65
4.1 Проблема переносимости........65
4.2 Универсальный слой абстракции........66
4.2.1 Структура HAL........67
4.2.2 Специальные команды процессора........69
4.3 Инициализация........72
4.3.1 Системы с одним процессором и микроконтроллеры........72
4.3.2 Инициализация многопроцессорной системы........73
4.4 Модель прерываний........74
4.4.1 Аппаратные прерывания........75
4.4.2 Исключения........79
4.4.3 Переключение контекста........79
4.4.4 Программные прерывания........84
4.4.5 Фрейм прерывания........86
4.4.6 Уровень маски прерываний........88
4.5 Взаимодействие между процессорами........91
4.5.1 Спинлоки........92
4.5.2 Спинлоки с очередью........95
4.6 Разделение привилегий........97
4.6.1 Стек ядра........99
4.6.2 Системные вызовы........100
4.6.3 Реализация переключения контекста........102
4.7 Защита памяти........103
4.7.1 MPU........103

4.7.2 MMU ... 104
4.7.3 Распределение адресного пространства ... 105
4.8 Пример реализации приложения на интерфейсах HAL ... 106
4.9 Резюме ... 109
5 Построение ядра ... 110
5.1 Политика и механизм ... 110
5.2 Исполнительная подсистема ... 111
5.3 Проектирование интерфейса ... 112
5.3.1 Выделение ресурсов ... 112
5.3.2 Потоки ... 113
5.3.3 Интерфейс потоков ... 115
5.3.4 Контекст функций ядра ... 118
5.3.5 Планирование ... 120
5.3.6 Блокирование вытеснения ... 123
5.3.7 Ожидание ... 124
5.3.8 Программные таймеры ... 126
5.4 Структура системы ... 127
5.5 Взаимодействие с прерываниями ... 129
5.5.1 Латентность ... 129
5.5.2 Отложенные процедуры ... 130
5.5.3 Схема синхронизации ... 134
5.6 Многопроцессорные системы ... 137
5.6.1 Симметричный мультипроцессор ... 137
5.6.2 Асимметричный мультипроцессор ... 140
5.6.3 Балансировка нагрузки ... 142
5.7 Завершение работы потока ... 143
5.8 Реализация ... 146
5.8.1 Реализация планировщика ... 146
5.8.2 Синхронизация планировщика ... 149
5.8.3 Реализация ожидания ... 154
5.8.4 Реализация потоков ... 162
5.8.5 Реализация таймеров ... 169
5.9 Резюме ... 176
6 Взаимодействие потоков ... 178
6.1 Задача синхронизации ... 178

6.2 Низкоуровневые механизмы синхронизации ... 181
6.2.1 Запрет прерываний и вытеснения ... 181
6.2.2 Неблокирующая синхронизация ... 182
6.2.3 Использование спинлоков ... 182
6.2.4 Использование приоритетов ... 183
6.2.5 Синхронизация с прямым уведомлением ... 183
6.3 События ... 185
6.4 Семафоры ... 188
6.5 Разновидности семафоров ... 191
6.6 Счетчики событий и секвенсоры ... 192
6.7 Инверсия приоритета ... 193
6.8 Мьютекс ... 195
6.8.1 Наследование приоритета ... 196
6.8.2 Потолок приоритета ... 198
6.8.3 Реализация мьютексов ... 198
6.9 Мониторы ... 200
6.10 Специфические примитивы ... 204
6.10.1 Флаги событий ... 204
6.10.2 Барьеры ... 205
6.10.3 RW-блокировки ... 206
6.10.4 Блоки памяти ... 207
6.10.5 Очереди сообщений ... 208
6.11 Резюме ... 210
7 Защита и изоляция ... 212
7.1 Постановка задачи ... 212
7.2 Концепция процесса ... 214
7.2.1 Модель защиты с одним процессом ... 217
7.2.2 Модель защиты с несколькими процессами ... 218
7.2.3 Планирование процессов ... 220
7.3 Подходы к реализации процессов ... 222
7.3.1 Виртуальные машины ... 222
7.3.2 Разделяемое ядро ... 225
7.3.3 Корректность ядра при некорректном поведении процесса ... 226
7.4 Структура системы с разделяемым ядром ... 226
7.5 Поддержка непривилегированных потоков ... 229

7.5.1 Переключение процесса ... 231
7.5.2 Обработчики завершения потока ... 232
7.6 Структура исполнительной подсистемы ... 235
7.7 Реализация системных вызовов ... 237
7.7.1 Использование объектов ядра ... 238
7.7.2 Выделение памяти ... 238
7.7.3 Подсчет ссылок на объекты ... 239
7.7.4 Идентификаторы объектов ... 242
7.7.5 Квоты ... 247
7.7.6 Проверка указателей ... 248
7.7.7 Завершение потока во время выполнения системного вызова ... 250
7.7.8 Блокирующие вызовы из режима пользователя ... 252
7.7.9 Динамическое создание процессов ... 253
7.8 Оптимизация производительности ... 254
7.8.1 Обработка прерываний в пользовательском режиме ... 254
7.8.2 Обработка исключений в пользовательском режиме ... 255
7.8.3 Реализация примитивов синхронизации в пользовательском режиме ... 255
7.8.4 Расширение интерфейса ядра ... 257
7.9 Ввод-вывод ... 259
7.10 Пример реализации процессов: FX-RTOS ... 261
7.10.1 Запуск и завершение процесса ... 263
7.10.2 Библиотеки поддержки ... 264
7.11 Резюме ... 265
8 Взаимодействие процессов ... 267
8.1 Необходимость взаимодействия процессов ... 267
8.2 Обзор механизмов межпроцессного взаимодействия ... 268
8.3 Синхронизация с помощью разделяемых объектов ... 271
8.4 Коммуникация с общей памятью ... 273
8.5 Асинхронный обмен сообщениями ... 274
8.6 Синхронный обмен сообщениями ... 277
8.7 Особенности реализации синхронного обмена сообщениями ... 280
8.8 Резюме ... 281

Предисловие

Зачем понадобилась еще одна книга об операционных системах

История операционных систем тесно связана с историей компьютеров как таковых. Ранние исследования в этой области уходят в начало 60-х годов. С тех пор был проделан большой объем исследовательской и инженерной работы, позволившей операционным системам стать тем, чем они являются сегодня. Было написано множество книг по данной тематике, однако практически любую из них, посвященную более или менее вопросу в целом, можно отнести к одному из двух основных типов.

Книги первого типа посвящены теории и традиционно включают в себя четыре основных раздела: процессы, управление памятью, ввод-вывод и управление файлами. Однако, прочтя такую книгу, довольно сложно себе представить работу реальных систем. Сталкиваясь с необходимостью разобраться в работе существующей системы, обнаруживается большой разрыв между тем, что описывается в теории, и тем, что существует на практике. Понимание затрудняется тем, что решаемые кодом проблемы зачастую неочевидны и не всегда лежат в плоскости теории.

Книги второго типа - "практического", - посвящены конкретной операционной системе и подробно описывают её исходный код. Являясь, по сути, своеобразной программной документацией, книга вынуждена систематически изменяться, отражая изменения происходящие в коде описываемой системы. Описание механизмов конкретной реализации смещает фокус от общего к частному, причем так, что законные вопросы читателя "А почему именно так, а не иначе?" нередко остаются без ответа.

Настоящая книга призвана в какой-то мере заполнить пробел между этими двумя типами. С одной стороны, здесь описаны проблемы, возникающие при проектировании ядра операционной системы и их возможные решения. Уровень детализации при описании достаточен для самостоятельной реализации рассматриваемых механизмов. С другой стороны, изложение не затрагивает нюансов реализации конкретных ОС, описываемые абстракции и механизмы имеют универсальный характер. Внимание уделяется не столько объяснению того, как работает какая-то сущность, сколько объяснению, почему она должна работать именно так, то есть выводу логики её работы из проблемы, стоящей перед разработчиком.

Разумеется, описание на таком уровне детализации всех аспектов реализации современной операционной системы привел бы к существенному росту объема книги, поэтому обсуждаются только вопросы реализации таких фундаментальных вещей как потоки, процессы и взаимодействие между ними. Существует мнение, что ядро не является главной частью операционной системы, а потому вопросы его реализации можно обсудить лишь вкратце, рассматривая "упрощенное" или "учебное" ядро и связанные с этим примеры. Но на практике это не так, поскольку именно ядро обеспечивает сцену, на которой действуют все остальные части ОС. Распределение памяти и управление файловой системой являются, в какой-то степени, прикладными

программами, хотя и используются не напрямую пользователем, а другими программами. В своей реализации, они, как правило, опираются на сервисы, предоставляемые ядром, поэтому при обсуждении последнего они могут быть вынесены "за скобки".

Ядро операционной системы относится к одному из самых сложных типов программных проектов. На технические сложности, связанные с высокой степенью параллелизма, накладывается концептуальная сложность декомпозиции системы. Поэтому вторая тема, неявно проходящая через большинство глав, это тема модульности и масштабируемости. Обсуждаются принципы, позволяющие декомпозировать систему на интерфейсные компоненты, чьи реализации могли бы разрабатываться независимо и в идеале дать возможность пользователю построить «профиль системы», оптимально настроенный на данную прикладную задачу.

Обзор содержания

Рассмотрение темы ведется в контексте встроенных систем, основные концепции которых рассматриваются в главе 1. К современным встроенным системам зачастую предъявляются требования работы в реальном времени. Поэтому описание механизмов операционных систем ведётся с акцентом на связанных с этими требованиями ограничениях. Основные определения так же приведены в первой главе.

Глава 2 описывает аппаратное обеспечение, которое используется для построения встроенных систем. Аппаратное обеспечение дает отправную точку для дальнейшего проектирования программного обеспечения, которое работает на этой аппаратуре. Описываются принципы исполнения кода, прерывания, синхронизация и отличия однопроцессорных и многопроцессорных архитектур.

В главе 3 описываются подходы к решению задач поставленных в главе 1, с помощью имеющейся аппаратуры, описанной в главе 2. Рассматриваются методы разработки встроенных систем как основанные на применении ОСРВ, так и без нее.

Глава 4 посвящена обсуждению отличий между различными процессорами и построению универсальной абстракции, называемой слоем абстракции оборудования (hardware abstraction layer, HAL), которая позволяет писать переносимый код. Помимо проектирования абстракции аппаратуры разбираются также принципы разработки кроссплатформенного встроенного ПО.

В главе 5 подробно рассматривается проектирование и реализация ядра ОСРВ на основе интерфейсов HAL. Подробно рассмотрены вопросы взаимодействия приложений с прерываниями, проблемы, возникающие при реализации поддержки симметричной многопроцессорности. Хотя, на первый взгляд, кажется, что небольшие ядра для встроенных систем не имеют ничего общего с "большими" ОС для настольных компьютеров, на самом деле, ядро и тех и других построено на базе одних и тех же принципов.

Глава 6 состоит из обсуждения механизмов синхронизации, используемых в приложениях, на основе ядра, спроектированного в предыдущей главе. Приводится пример реализации примитивов синхронизации, таких как семафоры, рассматриваются классические проблемы, возникающие при их использовании и методы их решения.

В главах 7 и 8 вводится понятие процесса как контейнера ресурсов и обсуждается реализация изолированной вычислительной среды, которая позволяет сосуществовать нескольким независимым приложениям в пределах одного физического устройства. Рассматриваются реализация ядра, поддерживающего исполнение кода с различными уровнями привилегий, а также организация взаимодействия между процессами.

Аудитория

Книга ориентирована в первую очередь на программистов, которые интересуются архитектурой ядра операционных систем, а также практикующих разработчиков встроенного ПО, которые используют ОСРВ в своей работе и хотели бы более детально разобраться с тем, что происходит "под капотом".

Поскольку довольно сложно обсуждать реализацию абстрактно, в качестве примера используется ОСРВ FX-RTOS. Выбранная ОСРВ обладает модульной архитектурой, которая хорошо подходит для демонстрации принципов разработки. Тем не менее, стоит еще раз подчеркнуть, что это книга не о внутреннем устройстве FX-RTOS. Упомянутая ОСРВ используется только в качестве иллюстрации одного из возможных подходов к решению обсуждаемых инженерных проблем. Чтобы соблюсти необходимый уровень универсальности, исходные тексты, хотя и написаны на С, приведены в виде фрагментов, иллюстрирующих наиболее важные или интересные с точки зрения реализации концепции.

Книга также будет полезна разработчикам периферийных устройств, процессоров и микроконтроллеров. Времена, когда и процессоры и ПО для них разрабатывалось внутри одной компании, прошли. Поэтому существует проблема эффективного взаимодействия между разработчиками аппаратуры и разработчиками системного ПО. ОС является наиболее тесно взаимодействующим с процессором программным компонентом, поэтому понимание требований со стороны ОС к процессору и его интерфейсам в конечном итоге позволит улучшить эти интерфейсы и сделать встроенные устройства более простыми, быстрыми и надежными.

История FX-RTOS

История FX-RTOS началась в 2010 году в виде внутреннего исследовательского проекта компании Fastwel. Встраиваемые ОС работают на большом многообразии устройств и производителям зачастую приходится использовать несколько ОС, потому что одна кодовая база с учетом устоявшихся подходов к проектированию и имеющихся в языке С возможностей абстракции не позволяет должным образом переиспользовать имеющийся код. Поэтому планировалось отказаться от концепции ОС, как некоторого неделимого компонента. Предполагалось с самого начала сосредоточиться на проектировании такой

системы, которая представляла бы собой набор простых модулей с четкими интерфейсами, хорошо масштабировалась без накладных расходов времени выполнения и по максимуму использовала код повторно в различных конфигурациях. В идеале, этот набор компонентов и инструментов должен был предоставить инфраструктуру для создания ОС с произвольной архитектурой. В отличие от "конфигурируемых" ОС-конструкторов, которые предоставляют некоторый конечный набор конфигурационных опций, система должна была быть открытой, то есть замена компонентов могла использовать не только компоненты, которые поставлялись с ОС, но и внешние, в том числе написанные пользователем.

В качестве основных целевых систем предполагались микроконтроллеры, но архитектура должна была иметь возможность поддержки и более продвинутых процессоров, которые постепенно завоёвывают популярность и во встроенном ПО, а также позволять строить защищенные системы с изоляцией выполняющихся приложений для повышения надёжности.

Таким образом, основные цели разработки были следующими:

- Конфигурируемость и компонетная архитектура
- Масштабирумость на разные классы устройств
- Возможность использования в системах жёсткого реального времени

Из-за своей модульной архитектуры, позволяющей наращивать функциональность очень мелкими шагами, FX-RTOS идеально подходит для изучения внутреннего устройства ядра ОС, поскольку принципы, лежащие в его основе, в основном определяются оборудованием и не меняются уже десятилетия.

1 Основные концепции

Темы главы:

- Что такое встроенные системы
- Особенности встроенных систем реального времени

1.1 Встроенные системы

По мере развития и усложнения техники возникла проблема сложности управления устройствами. Простые устройства, такие как, например, фонарик, имеют внутренний интерфейс управления совпадающий с внешним интерфейсом пользователя: система управления состоит из единственного выключателя, которым и управляет пользователь. Напротив, более сложные устройства, от стиральных машин и микроволновых печей, до жестких дисков и систем управления автомобилями и самолетами, имеют внутренний интерфейс на порядки превосходящий по сложности интерфейс пользователя. Стиральная машина работает по программе: управляет вращением барабана, набором воды, ее нагревом и так далее. Все эти действия скрыты от пользователя, в задачу которого входит только выбор программы и ее запуск. Таким образом, устройство управления представляет собой сложную функцию от входных данных или сигналов, как показано на Рис. 1.

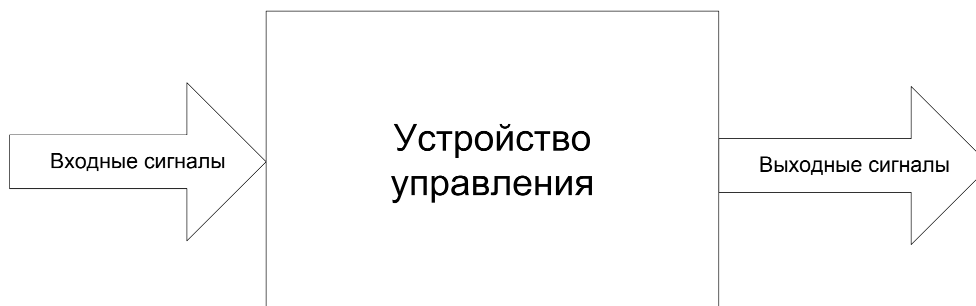

Рис. 1 Устройство управления

В роли входной информации необязательно используется именно некоторый пользовательский интерфейс. Например, автомобильные системы стабилизации используют в качестве входных данных информацию от датчиков скорости, ускорения и т.п.; система управления "умным домом" для поддержания заданной температуры опирается на датчики внутренней и внешней температуры.

Очевидно, простые механические решения, как в случае с фонариком, в данном случае не подходят. Существует несколько подходов к тому, как может быть реализована функция управления:

- Внешнее управление с помощью универсального компьютера
- Специализированные микросхемы (application-specific integrated circuit, ASIC)
- Программируемые микросхемы (field programmable gate arrays, FPGA)
- Микропроцессорная система управления

При этом было бы желательно, чтобы получившаяся система управления была дешевой, эффективной и удобной для отладки, поддержки и модификаций.

Самый простой вариант: вывести сигналы управления наружу и подключить к ним внешний компьютер, для которого написать программу, которая занималась бы

управлением. В некоторых случаях такой подход приемлем. Например, в банкоматах и платёжных терминалах часто используется небольшой ПК, но, поскольку подключенный компьютер необходим все время работы устройства, об эффективности и дешевизне такого решения говорить не приходится.

Другой вариант: заменить компьютер специализированной микросхемой, которая была бы спроектирована для управления данным конкретным устройством. Специализированные микросхемы хотя и дешевы сами по себе, требуют больших вложений на этапе проектирования. Кроме того, затрудняется поддержка, поскольку исправление ошибок в уже выпущенных микросхемах также сопряжено с большими финансовыми затратами.

Программируемые микросхемы упрощают отладку и модификации, но в зависимости от сложности устройства могут иметь довольно существуенную цену. Они незаменимы в тех случаях, когда требуется наивысшая возможная скорость работы устройства. В силу параллельной природы аппаратной составляющей FPGA (в отличие от последовательно исполняемых программ), устройство управления, реализованное с помощью FPGA может реагировать на входные данные, а также реализовывать функции, такие как декодирование видеосигнала или шифрование со скоростью, которая недостижима для программных решений сравнимой стоимости [1].

Наконец, можно использовать микропроцессорную систему, которая, выполняя управляющую программу, реализует функции управления. В качестве аппаратной составляющей используется микроконтроллер общего назначения с интегрированной памятью и периферией (Рис. 2).

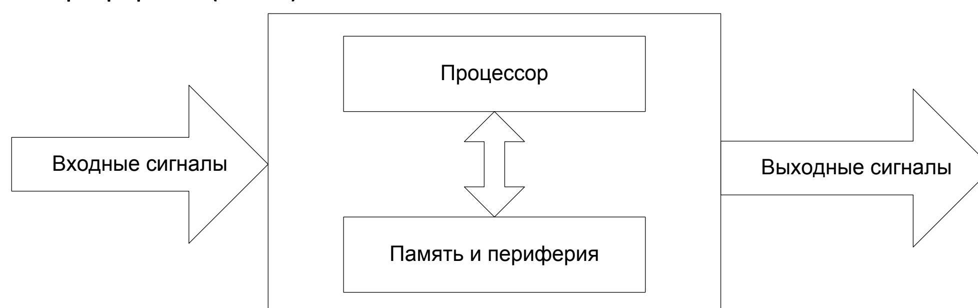

Рис. 2 Микропроцессорное устройство управления

Для большинства применений, такое решение является наиболее подходящим: микроконтроллеры дешевы, вся логика управления реализуется программно, поэтому ее проще отлаживать и поддерживать. Поскольку используются стандартные компоненты, разработку аппаратуры (самой системы управления и ее сопряжения с управляемым устройством) и ПО можно вести параллельно, что сокращает время выхода устройств на рынок.

Часто функция управления подразумевает сосуществование нескольких не связанных друг с другом задач. Например, mp3-плеер параллельно с декодированием и воспроизведением музыки должен отрисовывать экран и реагировать на пользовательский интерфейс. Выделение отдельных процессоров для каждой из подобных "фоновых" работ довольно затратно, поэтому, по аналогии с настольными компьютерами, используется принцип разделения времени: ПО предназначенное для

различных задач может использовать процессор по очереди. Такой подход позволяет наращивать функциональность встроенного ПО до тех пор, пока есть свободные вычислительные ресурсы.

Таким образом, **встроенная система** - это компьютер, встроенный непосредственно в целевое устройство и реализующий функции управления этим устройством. Программное обеспечение, работающее внутри встроенной системы называется **встроенным ПО** (embedded software, firmware). То есть программное обеспечение разрабатывается под конкретную микропроцессорную систему и поставляется вместе с ним. Этим встроенные системы отличаются от компьютеров общего назначения, которые предоставляют платформу для написания прикладных программ, которые разрабатываются и продаются отдельно.

Несмотря на то, что встроенные системы и не похожи на компьютеры внешне, они имеют в своем составе все традиционные компоненты компьютерной системы, такие как процессор, память, устройства ввода-вывода и прочее, а также к ним применимы все правила, касающиеся программирования. Принципы разработки встроенного ПО во многом сходны с принципами разработки ПО для персональных компьютеров, но имеются и существенные отличия. С одной стороны, для встроенного ПО так же как и для любого другого, важны такие общие качественные критерии как поддерживаемость, расширяемость, производительность и т.п., с другой стороны, встроенное ПО зачастую работает в условиях жёстких ресурсных ограничений и помимо алгоритмической корректности обязано также соответствовать требованиям работы в реальном времени, которые будут подробно обсуждаться далее.

В связи с тем, что количество продаваемых встроенных систем пропорционально продажам бытовой техники, автомобилей и т.д., количество процессоров работающих в составе встроенных систем намного превышает количество процессоров работающих в компьютерах. По некоторым оценкам, на компьютеры и серверы приходится всего 2% всех выпускаемых в мире процессоров [2].

Помимо бытовой техники, множество встроенных систем содержатся внутри самих компьютеров. Например, программный интерфейс жестких дисков (представляющий диск как массив секторов, которые можно читать и писать) или твердотельных накопителей намного проще внутренней логики управления диском, требующей управления вращением, позиционированием головок, шифрованием и так далее. Все эти функции берет на себя встроенная система, которая, по описанным выше причинам, обычно реализуется в виде микропроцессорной системы и управляющей программы. Использование программного решения позволяет унифицировать используемую аппаратуру и в зависимости от меняющихся требований использовать на одном и том же устройстве разное ПО. Например, в настольных дисках может применяться вариант ПО, позволяющий добиться большей производительности, тогда как в мобильных дисках для ноутбуков приоритет может быть отдан энергопотреблению.

В настоящее время практически любое устройство, от USB-накопителей до сложных медицинских и научных приборов, содержит управляющий микропроцессор и встроенное ПО.

1.2 Системы реального времени

Встроенные системы часто имеют реактивную природу - должны реагировать на события. Под событием обычно понимается изменение входных данных. Время между изменением входных данных и соответствующим изменением выходных называется **временем реакции**. При этом под временем реакции понимается не среднее, а худшее время. Если время реакции ограничено, то есть система должна реагировать на входные сигналы с определенной, заданной наперед скоростью, то такая система называется **встроенной системой реального времени**.

Корректность работы при этом зависит не только от самих выходных сигналов, но и от времени, которое необходимо на работу самой системы управления.

Как правило, большинство встроенных систем имеет требования, связанные с временем реакции. В зависимости от жесткости этих требований существует разделение на системы мягкого и жесткого реального времени. Системы мягкого реального времени стараются удовлетворять заданным временным ограничениям, но не гарантируют их выполнения в любой момент времени. В качестве примеров можно привести всевозможные системы связи или потоковой обработки мультимедийной информации: от видеоплеера требуется выполнение декодирования в определенном темпе так, чтобы видеопоток для пользователя оставался плавным, однако, если по каким-то причинам, иногда плеер не будет "успевать", то катастрофических последствий это не повлечёт (например, не более 0,01% потерянных кадров или пакетов в день).

Системы жесткого реального времени должны гарантировать выполнение заданных ограничений на время реакции в любой момент времени. Подобные требования обычно предъявлются к ответственным системам, которые используются для управления транспортом или технологическими процессами. Даже единичное пропущенное событие может спровоцировать катастрофические последствия.

Для разработки таких систем на этапе проектирования применяются техники верификации, позволяющие удостовериться в соблюдении заданных временных ограничений.

1.3 Детерминизм

Из определения системы жесткого реального времени следует, что для корректной работы, ее разработчик должен иметь возможность определить максимальное время, требуемое для работы любой функции. Если для любого набора входных сигналов и состояния системы может быть определен набор выходных сигналов, причем затрачиваемое на это время конечно и предсказуемо, то такая система называется детерминированной или обладающей детерминизмом.

Для достижения детерминизма требуется не только использование специальных алгоритмов при разработке ПО, но и использование соответствующего аппаратного обеспечения. Этим встроенные системы реального времени также отличаются от компьютеров общего назначения, в которых применяются множество оптимизаций для улучшения средней скорости работы, несмотря на то, что при этом может пострадать предсказуемость. В качестве примера можно привести организацию виртуальной памяти в ПК: каждой программе предоставляется больше оперативной памяти, чем физически

имеется в компьютере. Непосредственно в оперативной памяти находятся только наиболее часто используемые фрагменты, остальное выгружается во внешнюю память, например на жесткий диск. При попытке обратиться к выгруженной памяти, обращение перехватывается, требуемый участок загружается с диска (для чего, возможно, на диск выгружается другой участок) и выполнение программы продолжается. Нетрудно увидеть, что задержки в выполнении программы (для обращения к диску) возникают случайным и непредсказуемым заранее образом.

1.4 Резюме

Использование микропроцессорных устройств управления является наиболее обоснованным с экономической точки зрения в большинстве случаев. В таких системах логика управления реализуется программно. В зависимости от типа устройства, на встроенное ПО могут накладываться временные ограничения, в этом случае управляющая встроенная система называется встроенной системой реального времени. Соответствие требованиям реального времени предполагает реализацию встроенного ПО таким образом, чтобы на этапе разработки можно было определить максимальное время реакции. Этим встроенные системы отличаются от систем общего назначения.

[1] Karen Parnell, Roger Bryner, "Comparing and Contrasting FPGA and Microprocessor System Design and Development" 2004
[2] Barr Michael, "Real men program in C". Embedded Systems Design. 2009

2 Обзор аппаратного обеспечения

Темы главы:

- Программный интерфейс процессоров и устройств ввода-вывода
- Кэширование
- Прерывания
- Низкоуровневые механизмы синхронизации
- Особенности многопроцессорных систем
- Механизмы изоляции и защиты

2.1 Классическая архитектура

2.1.1 Абстрактный процессор

Хотя существует большое разнообразие процессоров и микроконтролеров, все их можно рассматривать как некоторый абстрактный вычислитель.

Архитектура фон Неймана, которая традиционно используется в качестве универсальной абстракции, предусматривает в составе компьютерной системы процессор, память и устройства ввода-вывода объединенные шиной (Рис. 3).

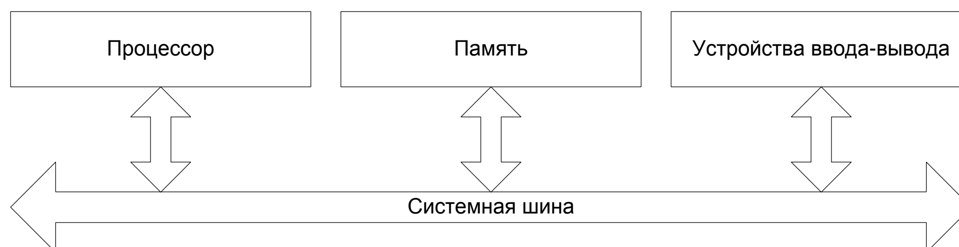

Рис. 3 Архитектура фон Неймана

Приведенная схема несколько упрощена, но дает хорошее приближение к реальным архитектурам с точки зрения ПО. Хотя существуют и компьютеры с архитектурами отличными от архитектуры фон Неймана, они применяются довольно редко и в этой книге не рассматриваются.

Память содержит выполняемую программу в виде последовательности инструкций. Процессор имеет регистр, указывающий на текущую исполняемую команду и последовательно выбирает из памяти инструкции для исполнения. Работа процессора, таким образом, заключается в бесконечном цикле выборки и исполнения команд.

Кодирование и набор команд специфичны для данного типа процессора и формируют важный слой абстракции, называемый **системой команд** (instruction set architecture, ISA). Хотя набор команд и специфичен для конкретного процессора, как правило, в этот набор входят арифметические, логические, управляющие и команды перемещения данных.

Следует отметить, что память является довольно медленным устройством по сравнению с процессором. Также память в большинстве случаев является однопортовой: за один запрос можно прочитать либо записать только одну ячейку. Из-за этого выполнение типичной команды требует нескольких тактов, поскольку у арифметических и логических операций, как правило, несколько операндов, а кроме того, обращения в память требует и сама выборка команды. Для ускорения работы с памятью применяется следующая

оптимизация: процессор содержит набор регистров, которые называются регистрами общего назначения (general-purpose registers, GPR) и все арифметические и логические операции определяются над регистрами, а не напрямую над памятью. В отличие от памяти, доступ к регистрам можно сделать многопортовым, и, таким образом, позволить многим командам работать за 1 такт.

Архитектура системы команд, при которой к памяти обращаются только выделенные для этого команды вида "прочитать/записать ячейку памяти из/в регистр" называется архитектурой с сокращенным набором команд (reduced instruction set computer, RISC). Альтернативный подход (complete instruction set computer, CISC), который исторически появился первым, допускает использование памяти в качестве одного из операндов, а также команды, выполняющие сложные составные действия, такие как работа со строками. Существенное усложнение процессора, для поддержки разнообразных типов операндов и большого набора команд, приводит к тому, что, хотя процессору и надо выполнить меньше команд чем процессору с RISC-архитектурой, выполнение команд занимает больше тактов, и поэтому процессор в целом оказывается медленнее.

Хотя существуют и другие подходы к архитектуре процессоров, описанный выше подход с наличием в процессоре регистров набором команд для выполнения операций над этими регистрами является доминирующим. Процессоры с такой архитектурой называются **регистровыми машинами**.

Процессор содержит регистры двух основных типов: регистры общего назначения, и специальные регистры. Первые, во многих процессорах, традиционно имеют имена вида Rn, где n - номер регистра. Например R1, R2, R15 и т.д. Эти регистры используются программой для прикладных нужд. Специальные регистры содержат информацию о состоянии процессора, режиме выполнения и т.д. К специальным регистрам относятся, например, указатель инструкций (традиционно называемый PC, program counter) - регистр содержащий адрес текущей исполняемой инструкции, а также один или более регистров состояния (обычно называемый SR, status register). К специальным регистрам относится также регистр стека, содержащий указатель на текущую вершину, называемый SP (stack pointer).

Каждая операция содержит код операции и операнды (которыми могут выступать регистры или адреса памяти, в зависимости от типа конкретной архитектуры).

Состояние процессора полностью определяется состоянием его регистров, содержимое которых называется **контекстом**. Возможность сохранения содержимого регистров в память, и восстановления из нее, позволяет реализовать разделение времени между задачами.

Таким образом, в первом приближении, абсолютное большинство из используемых в настоящее время процессоров, представляют собой регистровую машину с определенным для нее набором команд, которые выполняют действия над регистрами либо памятью. Количество и функции регистров зависят от конкретной модели процессора, но обычно доступно для использования от 8 до 16 регистров общего назначения плюс специальные регистры (Рис. 4).

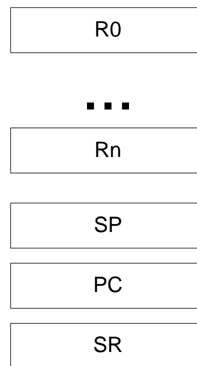

Рис. 4 Набор регистров абстрактной регистровой машины (контекст)

Производительность процессора можно определить как произведение количества тактов в единицу времени и количества команд, выполняемых за такт.

Если для каждой команды определено время ее выполнения и известна частота, на которой происходит выборка и исполнение команд, то для любого заданного набора инструкций (представленных, например в виде листинга на ассемблере) становится возможно рассчитать время его выполнения. Однако это справедливо не для всех процессоров.

Процессоры применяемые в высокопроизводительных компьютерах (в том числе в настольных ПК и серверах) используют так называмое "спекулятивное исполнение": не зависящие друг от друга по данным инструкции исполняются параллельно на имеющихся в процессоре нескольких экземплярах исполняющих устройств. Это позволяет достичь среднего количества инструкций, выполняемых за такт, больше единицы и улучшает общую производительность. К сожалению, в подобных процессорах время выполнения команд зависит от того, что процессор выполнял до этого, в том числе результат всех произошедших до этого переходов (произошел переход или нет). Поэтому определить точное время исполнения фрагмента кода уже нельзя. Из-за этого, при необходимости построения системы жесткого реального времени, следует учитывать особенности процессора, который предполагается использовать. Подробно о реализации механизма спекулятивного исполнения можно прочесть в [1] и [2].

При необходимости использования ассемблера для иллюстрации какого-либо механизма, далее будет использоваться абстрактный ассемблер, не привязанный к какому-либо существующему процессору, если только код не должен иллюстрировать специфические для некоторого процессора концепции.

2.1.2 Иерархия памяти

Скорость работы внешней памяти сильно отличается от скорости процессора. При частых обращениях в память, процессор вынужден был бы большую часть времени дожидаться ответа, так как в среднем обращение к динамической (DRAM) памяти занимает время порядка десятков наносекунд [1]. Для повышения производительности системы при доступе к памяти используются некоторые предположения, которые оперируют вероятностью повторного доступа к определённым ячейкам памяти, а именно пространственной и временной локальностью.

Под пространственной локальностью (spatial locality) подразумевается тот факт, что более

вероятно обращение к данным, которые находятся по соседним адресам в памяти. Действительно, в программировании часто используются структуры и классы, с помощью которых сложные объекты составляются из более простых, а также массивы, поэтому предположение о том, что после обращения по определенному адресу могут последовать обращения по соседним адресам не лишено оснований.

Временная локальность (temporal locality) подразумевает, что более вероятно обращение к данным, к которым уже было обращение.

Те же самые рассуждения применимы и к выборке кода, поскольку выборка происходит последовательно и программы часто используют циклы, которые многократно используют инструкции по одному и тому же адресу.

Эти свойства позволяют реализовать оптимизацию, называемую **кэш-памятью**. Идея заключается в том, что между процессором и памятью находится память меньшего размера, но более высокого быстродействия, по сравнению с основной памятью (Рис. 5).

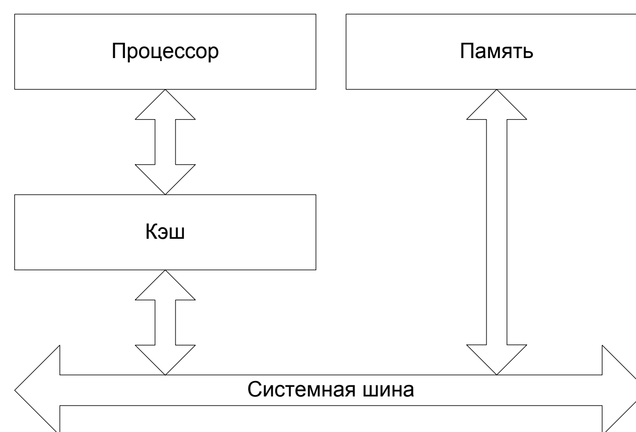

Рис. 5 Архитектура системы с кэш-памятью

Это позволяет сохранять там фрагменты кода и данных, которые наиболее часто используются процессором. Если требуемые данные находятся в кэше, запрос процессора может быть обслужен без обращения в основную память, причем все это происходит прозрачно для исполняющейся программы.

Если при обращении обнаруживается, что требуемые данные есть в кэше, это называется попаданием (cache hit), в противном случае - промахом (cache miss).

Подробное обсуждение устройства кэша выходит за рамки данной книги, посвященной только вопросам написания ПО, но, в общих чертах, кэш можно представить как массив блоков фиксированной длины, называемых **линиями** (cache line). У каждой линии есть поле состояния, содержащее информацию о том, действительны ли данные находящиеся в ней. Адрес, по которому обращается процессор разделяется на три основных части: тег, индекс и смещение (Рис. 6).

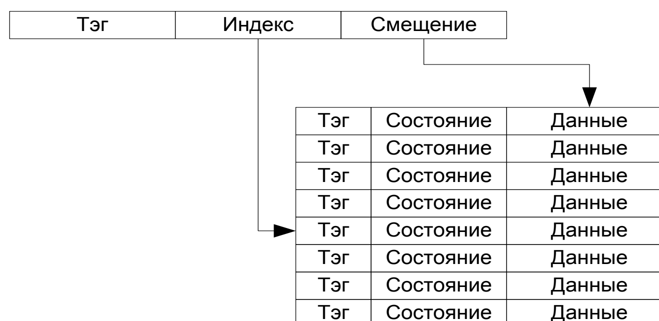

Рис. 6 Кэш-память

Когда происходит обращение к памяти, индекс из адреса используется для поиска строки в массиве. Каждая строка содержит тэг, в роли которого выступают старшие части адреса, для которого строка содержит кэшируемые данные. После нахождения строки, происходит сравнение тэга, если он совпадает, запрашиваемые данные передаются процессору. Смещение играет роль адреса внутри линии. В случае несовпадения тэга, происходит замещение строки: выгрузка данных из кэша в память и загрузка из памяти данных к которым выполнялось обращение.

Такой кэш называется кэшем прямого отображения (direct-mapped cache). Как нетрудно увидеть, при последовательных обращениях к данным, отстоящим друг от друга на размер кэша (отличается тэг, индекс совпадает) будет просиходить постоянное замещение строки без повторного использования кэшированных данных. Поэтому наиболее распространенным вариантом организации кэша является так называемый множественно-ассоциативный кэш (set-associative cache). Он призван разрешить кэширование хотя бы нескольких экземпляров данных, адреса которых различаются на размер кэша. Вместо единственной строки, соответствующей каждому индексу, используется несколько, такие строки формируют набор.

При обращениях к данным с одинаковым индексом, строки заполняются по очереди, а вытеснение в память происходит тогда, когда свободных строк в наборе не осталось. Количество строк в наборе обычно не превышает 16 (а в большинстве случаев от 4 до 8), поэтому может наблюдаться резкое падение производительности при незначительном увеличении размера данных, используемых программой, в том случае, когда начинает расти частота вытеснения строк из кэша.

Идею кэша можно развить дальше до многоуровневой архитектуры. То есть можно сделать кэш для кэша. Архитектура системы при этом приобретает следующий вид, как показано на Рис. 7.

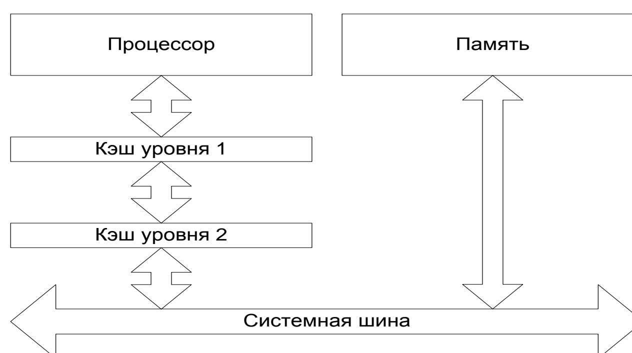

Рис. 7 Многоуровневый кэш

Чем ближе кэш к процессору, тем быстрее скорость его работы и тем меньше его объем.
Как и прочие оптимизации, направленные на улучшение средней производительности, кэш, за счет скрытого состояния, влияет на время выполнения инструкций процессором. Поэтому для систем жесткого реального времени кэширование нежелательно: невозможно заранее предсказать время выполнения обращения в память, оно зависит от того, присутствует ли в кэше запрашиваемые данные. Также это вызывает трудности с переносимостью, если архитектура кэша отличается, то при сравнимых частотах и прочих характеристиках выполнение программы может происходить с разной скоростью.
В реальных задачах часто используется компромиссный вариант: некоторые процессоры, например, процессоры серии ARM Cortex-R, предназначенные для разработки приложений, работающих в реальном времени, содержат, так называемую, тесно связанную с процессором память TCM (tightly-coupled memory). Она отображается на адресное пространство процессора по определенным адресам и отличается тем, что обращение к этому региону выполняется в обход кэш-памяти, которая применяется для ускорения доступа к опциональной внешней динамической памяти. Данные, доступ к которым должен отличаться предсказуемостью, размещаются в этой памяти и обращение к ним занимает строго определенное время.
В микроконтроллерах также используется похожий подход: внешняя память отсутствует, в качестве оперативной используется небольшой объем статической памяти расположенный непосредственно на кристалле процессора. Доступ к этой памяти быстрый, а потому она не нуждается в кэшировании.
Если процессор не поддерживает такие возможности, можно расположить приложения реального времени и их данные в некэшируемом регионе памяти.

2.1.3 Ввод-вывод

Помимо выполнения инструкций из памяти и вычислений, компьютеру необходимо взаимодействовать с внешним миром. Для этого используются различные устройства, с которыми может взаимодействовать выполняемая программа. К таким устройствам относятся таймеры, сенсоры, интерфейсные устройства для сети и различных промышленных протоколов и т.д.
С точки зрения программиста, устройство - это набор его регистров. По ходу своего выполненения программа читает и записывает эти регистры реализуя протокол работы с данным конкретным устройством и тем самым осуществуляет ввод или вывод.
Для доступа к регистрам устройств может использоваться отдельное адресное пространство (отдельное от адресного пространства памяти), в этом случае система команд процессора содержит специальные команды, отличные от команд доступа к памяти. Например, процессоры Intel x86, помимо адресного пространства памяти имеют также отдельное адресное пространство ввода-вывода, для доступа к которому используются инструкции IN и OUT.
Гораздо более распространенный подход предполагает разделение одного и того же адресного пространства между памятью и устройствами. Такой подход называется **отображением ввода-вывода на память** (memory mapped input-output). Адресное пространство процессора обычно намного больше, чем имеется в наличии оперативной

памяти, поэтому в нем можно разместить как память так и устройства.
Физическая реализация этого решения предполагает использование декодеров адреса на шине у каждого агента. Когда процессор обращается по какому-либо адресу, все агенты шины сравнивают адрес с тем, который им назначен и, если обнаружиается совпадение, (которое должно быть только одно) то это устройство и отвечает на запрос.
Отображенные на память устройства могут использоваться из языка программирования высокого уровня, такого как C, непосредственно. Регистры устройства описываются в виде структуры. Далее программа обращается по определенному заранее адресу.
В связи с тем, что устройства работают не мгновенно и намного медленнее чем процессор, после выполнения запроса к устройству в программе необходимо использовать циклы ожидания, с помощью которых процессор может дождаться, пока устройство будет готово к обмену данными.
Такой тип ввода-вывода называется программным и предполагает полное управление устройством со стороны процессора. Другое название - ввод-вывод по опросу или **полинг**.
Типичный обмен данными с устройством в программе выглядит так:

```
struct device_regs {
  uint32_t data;     // регистр для обмена данными
  uint32_t request;  // регистр запросов
  uint32_t status;   // регистр состояния устройства
};

struct device_regs *my_device = <адрес устройства>;
my_device->request = <код запроса>;
while ((my_device->status & <бит готовности устройства>) == 0);
uint32_t data = my_device->data;
```

Каждый обмен данными с устройством требует нескольких циклов шины, поскольку данные сохраняются в памяти: чтение из регистров устройства с последующим циклом сохранения. Функция процессора в данном случае заключается только в том, чтобы прочитать данные из одного места и записать в другое. Для того, чтобы избавить процессор от рутинных перекачек данных в память и обратно используется механизм **прямого доступа к памяти** (direct memory access, DMA).
Поскольку устройства имеют доступ к памяти по той же шине что и процессор, они могут работать с памятью самостоятельно. Работа программы при этом похожа на вариант с программным вводом-выводом, но процессору нужно только дождаться готовности периферийного устройства, после того как сигнал готовности получен, данные уже находятся в памяти.

```
struct device_regs {
  uint32_t data;     // регистр для обмена данными
  uint32_t request;  // регистр запросов
  uint32_t status;   // регистр состояния устройства
  uint32_t address;  // регистр адреса для DMA-обменов
};
```

```
uint32_t buffer[N];

struct device_regs *my_device = <адрес устройства>;
my_device->address = (uint32_t) buffer;
my_device->request = <код запроса>;
while ((my_device->status & <бит готовности устройства>) == 0);
// можно обращаться к прочитанным или записанным данным в буфере
```

Использование DMA требует существенного усложнения как устройств, так и процессора, в особенности, его кэша. Кэш сохраняет копии данных из памяти, в тот момент когда память перезаписывается устройством через прямой доступ, копии данных в кэше в этом процессе не участвуют, поэтому, при обращении к этим данным, процессор мог бы получить некорректные (устаревшие) данные. Для решения этой проблемы кэш реализует протоколы слежения за шиной и помечает данные, к которым обращались устройства, как недействительные. Обращение к ним процессора вызовет цикл обращения в память, минуя кэш.

В примерах выше рассматривалось только синхронное устройство, то есть такое, которое управляется запросами пользователя. Инициатором обмена данными всегда выступает программа. Существует также большой класс устройств, называемых асинхронными. К ним, в частности, относятся сетевые адаптеры. Готовность к обмену данными в этом случае возникает по приходу пакетов либо иной сетевой активности. То есть инициатива принадлежит устройству, программа не может заранее знать, когда именно придет сетевой пакет. В этом случае возникает проблема: как программе оперативно узнать о необходимости обмена данными с устройством? Если читать регистр статуса устройства слишком редко - страдает скорость реакции, если же проверять слишком часто, то программа на 90 и более процентов будет состоять из чтения состояния (полинга) регистров. Для решения этой проблемы была разработана концепция прерываний.

2.1.4 Прерывания

В идеале, для достижения наилучшей скорости реакции, хотелось бы проверять состояние устройств после каждой выполненной инструкции. Поэтому решение заключается в том, чтобы встроить логику проверки состояния устройств непосредственно в процессор и сделать ее частью цикла выборки и исполнения команд. Логически это эквивалентно полингу после каждой команды.

В отличие от полинга, при котором процессор по своей инициативе периодически читает регистры устройства (формат которого известен программе), требуется аппаратный интерфейс для устройств, чтобы они могли послать сигнал готовности процессору. То есть чтобы аппаратный полинг выглядел для всех устройств одинаковым образом.

Этот интерфейс специфичен для разных моделей процессора и требует специального устройства - **контроллера прерываний**. Контроллер прерываний - это интерфейсное устройство для сопряжения внешних устройств с разными процессорами. Со стороны процессора существует скрытый внутренний регистр, анализ состояния которого после

выполнения каждого цикла инструкции позволяет узнать о наличии сигналов со стороны устройств (Рис. 8).

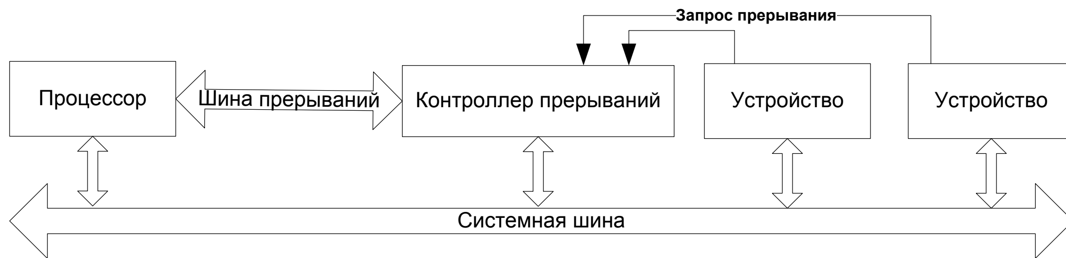

Рис. 8 Контроллер прерываний

При наличии активного запроса прерывания, процессор производит действия, эквивалентные вызову подпрограммы: помещает адрес возврата в стек и переходит к выполнению функции-обработчика прерывания, адрес которой предварительно должен быть сообщен процессору.

Часто для сохранения адреса возврата используется стек, но могут быть также специальные выделенные ресурсы специально для прерываний, в частности, специальные регистры, в которых сохраняется адрес возврата. Использование регистров предпочтительно, потому что обработка прерываний использует меньше обращений к памяти. В некоторых процессорах (например, Sun SPARC) используются **регистровые окна**, которые позволяют не сохранять контекст при вызове функции-обработчика. В этом случае обработчик прерывания использует отдельный набор регистров, переключение на который происходит в момент вызова функции-обработчика.

Работа устройства не связана с выполнением программы процессором, поэтому прерывание может прерывать пользовательский код в случайных местах.

Поскольку устройств в наличии обычно более одного, обработчику как-то надо узнать, от какого именно устройства пришел запрос. К решению этой задачи есть несколько подходов, но большинство из них связаны с **векторами прерываний**. Вектор - атрибут источника прерывания, который позволяет различать запросы от разных устройств. Разным векторам соответствуют разные функции-обработчики прерывания, так что каждый обработчик может работать только со "своим" устройством. Реализация этого механизма различается для разных процессоров, но в большинстве случаев используются два основных подхода.

Первый из них предполагает существование в памяти так называемой **таблицы векторов прерываний** (interrupt vector table, IVT). Она находится либо в фиксированном месте в памяти, о котором знает процессор, либо у последнего есть специальный регистр, который указывает на таблицу. Например, у процессоров intel x86 этот регистр называется IDTR. При возникновении прерывания, контроллер сообщает процессору вектор, после чего вектор используется как индекс в таблице для поиска обработчика, как показано на Рис. 9.

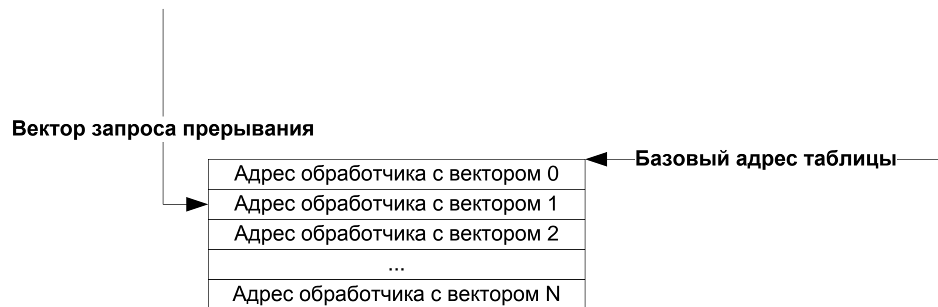

Рис. 9 Таблица векторов прерываний

При этом, в случае большого количества векторов, возможны существенные расходы памяти на таблицу. Кроме того, такой подход не очень удобен с точки зрения использования. Обработчики, выполняющие работы по обслуживанию устройства редко устанавливаются прямо в таблицу, обычно нужен некоторый инфраструктурный каркас.

По этим причинам часто используется другой вариант - использование одного вектора на все аппаратные прерывания плюс возможность узнать текущий вектор уже внутри обработчика путем чтения либо регистров процессора, либо регистров контроллера прерываний.

Этот подход лишен недостатков первого, в памяти содержится минимальное количество данных, а обработка может быть реализована различными способами уже программно: либо общим обработчиком, либо поиском в другой таблице с помощью вектора.

Допустимы также промежуточные варианты, например, у процессора может быть меньше векторов, чем у контроллера прерываний. Некоторые векторы могут вызываться напрямую, тогда как другие должны анализировать состояние контроллера, так как возможно несколько источников.

Прерывания могут возникать в произвольное время и независимо друг от друга, поэтому несколько источников вполне могут установить запрос прерывания одновременно. Возникающую при этом неопределенность: какой обработчик должен быть вызван, разрешают с помощью **приоритета прерываний**. Для каждого вектора прерываний устанавливается приоритет, статически, либо с возможностью его изменения во время работы. Обработчики прерываний получают управление в порядке приоритета, при этом, в зависимости от процессора, возможно не только определение порядка вызова при поступлении нескольких запросов, но и способность одних обработчиков прерывать другие: высокоприоритетные прерывания могут прерывать низкоприоритетные (Рис. 10).

Рис. 10 Вложение обработчиков прерываний

Приоритеты могут реализовываться как самим процессором, так и контроллером

прерываний.
Помимо гибких возможностей реагировать на события и улучшения времени реакции системы, по сравнению с использованием полинга, прерывания дают возможность реализовать в процессоре различные режимы энергосбережения. Например, система команд может содержать специальную инструкцию останова, которая приостанавливает выборку и исполнение инструкций процессором до получения прерывания.

2.1.5 Исключения

Идею проверки некоторых условий перед или после выполнения каждой инструкции можно развить дальше. Например, процессор может проверять корректность аргументов инструкций. В случае обнаружения каких-либо аномалий, могут происходить действия аналогичные прерыванию: вызов процедуры-обработчика ошибки. Такой тип прерываний называется **исключениями** или исключительными ситуациями. Если сравнивать их с внешними аппаратными прерываниями, то основное отличие заключается в том, что они являются синхронными: инструкция, выполнение которой вызывает исключение, будет вызывать это исключение всегда (с теми же параметрами), тогда как прерывания возникают независимо от действий процессора. Например деление на 0 вызывает соответствующее исключение, всякий раз, когда у команды деления в качестве делителя будет 0, при этом будет вызываться обработчик исключения.
Прерывания и исключения являются близкими понятиями, и их обработка процессором зачастую идентична, отличается только природа их источника. Эти соображения приводят к тому, что в большинстве процессоров нет различий в прерываниях и исключениях с точки зрения программиста: для исключений выделено несколько векторов в той же таблице, которая используется и для прерываний.
Предполагается, что обработчик исключения должен проанализировать причину ошибки, выявить ошибочную инструкцию (по адресу возврата, который сохраняется, как и для прерываний) и, возможно, предпринять действия по устранению ошибки. Исключения могут использоваться для сигнализации о наличии серьезных аппаратных проблем, когда процессор обнаруживает несогласованность своего внутреннего состояния, которое может произойти, например, при перегреве.
Исключения реализуются самим процессором ядром, контроллер прерываний для их обработки не используется.
Хотя, как будет показано далее, в разделе про реализацию процессором защиты памяти, в некоторых случаях исключения можно считать нормальным ходом выполнения программы. В большинстве же случаев, в отличие от прерываний, возникновение исключений является ошибкой и сигнализирует о проблемах и невозможности продолжения работы прикладной программы, а действия обработчика сводятся к тому, чтобы завершить выполнение ошибочного приложения.

2.1.6 Атомарность

Прерывания могут происходить в произвольных точках в коде, при этом выполняемая программа и обработчик прерывания могут пользоваться общими данными в памяти.

Например, обработчик прерывания от сетевого интерфейса может устанавливать переменные, указывающие на пришедший пакет, при этом в процессе работы с этими переменными может прийти следующий пакет и так далее. Нужен способ приостановить получение новых прерываний во время работы с разделяемыми данными из приложения. Для реализации такой возможности, процессор поддерживает специальные инструкции, с помощью которых можно разрешать или запрещать получение прерываний. Блок кода, выполняемый с запрещенными прерываниями называется **критической секцией**.

```
disable_interrupts();
<код критической секции>
enable_interrupts();
```

Поскольку прерывания не могут влиять на код критической секции, все действия, которые были выполнены в ней будут выглядеть для обработчиков прерываний как единое неделимое или атомарное действие.

Например, если имеется связный список, элементы которого могут добавляться и удаляться из программы и обработчиков прерываний, важно обеспечить атомарность работы со списком: несколько действий, выполняемых для вставки или удаления элемента, должны выглядеть как одно действие, которое либо не совершается вовсе, либо совершается полностью. В противном случае, список может оказаться в несогласованном состоянии и может быть разрушен.

Возможность выполнять составные операции, которые будут выглядеть как единое действие, очень важна для согласованной и непротиворечивой работы прикладных программ. В конечном итоге, любая атомарность, реализуемая компьютерной системой сводится к возможностям, предоставляемым процессором. Например, платежная система не допускает списания одних и тех же денег несколько раз пользуясь возможностями атомарного выполнения транзакций, предоставлемой системами управления базами данных (СУБД). В свою очередь, СУБД пользуется сервисами ОС, которая реализует свои механизмы через возможности предоставляемые процессором аппаратно.

Если используемый процессор допускает вложение обработчиков прерываний, сразу же возникает вопрос относительно атомарности процедуры входа в обработчик. Атомарность передачи управления обработчику обычно поддерживается аппаратно, но ОС может требоваться выполнить одновременно с этим дополнительные действия. Например, может потребоваться вести счетчик вложения прерываний, который увеличивался бы всякий раз при входе в обработчик, и уменьшался бы при выходе. Процессор ничего об этом не знает, а потому возможна ситуация, когда сразу же после входа в один обработчик, возникает более приоритетное прерывание, до выполнения первой инструкции в первом обработчике. Вложенный обработчик может ошибочно решить что он не является вложенным.

Для поддержки вложенных обработчиков некоторые процессоры запрещают прерывания при входе в обработчик, который должен сам разрешить или не разрешить вложение обработчиков (к ним относятся, например процессоры x86). Другие процессоры

поддерживают понятие приоритета прерывания и вместо запрета всех прерываний, может запрещаться выполнение только обработчиков заданного приоритета (например, ARM Cortex-M). Применяется также не очень широко распространенный подход, когда обработчик может определить, что именно он прервал, по информации, сохраняемой в стек.

Временный запрет прерываний широко используется как средство обеспечения атомарности в однопроцессорных системах. При этом необходимо помнить, что запрет прерываний никак не влияет на прямой доступ к памяти со стороны устройств, поэтому такие устройства следует программировать таким образом, чтобы используемые для прямого доступа регионы памяти более никем не использовались.

Поскольку внешние события обычно приходят к процессору в виде прерываний, для достижения высокой скорости реакции на них необходимо, чтобы критические секции в коде были как можно меньше. Критические секции, длина которых зависит от параметров времени выполнения приводят к недетерминированному выполнению - время реакции системы становится невозможно предсказать.

При использовании разделяемых данных между обработчиками прерываний следует также учитывать требования к выравниванию данных в памяти. Все процессоры, в том числе и RISC-архитектуры, позволяют обращаться к данным с указанием размера, причем некоторые процессоры (например, ARM) генерируют исключение при обращении к невыровненным данным. Может также требоваться дополнительное выравнивание стека или каких-то структур данных, которые аппаратно обрабатываются процессором. Если это требование не выполняется, чтение или запись переменной, которая не выровнена (например, чтение 4 байт выровненных на 2) может повлечь нескольких транзакций на шине, и, следовательно, такая запись не будет атомарной - если процессор допускает возникновение прерывания между такими транзакциями, то, теоретически, обработчик прерываний может увидеть промежуточное состояние ячейки памяти.

2.2 Многопроцессорные системы

Хотя традиционная архитектура предусматривает только один процессор, в некоторых случаях, бывает полезно использовать несколько процессоров. Причины такого решения могут быть самыми различными. Процессор - критически важный компонент системы. Желание повысить надежность, чтобы выход из строя процессора больше не являлся фатальной ошибкой, а также скорость, поскольку несколько задач могут выполняться параллельно, могут стать аргументом в пользу использования нескольких процессоров. Из других причин, связанных с системами реального времени, можно упомянуть скорость реакции: некоторые задачи могут иметь свой выделенный процессор и не зависеть от выполнения других задач. Кроме того, часто использование нескольких не очень производительных процессоров является экономически более предпочтительным, поскольку часто цена процессора растет быстрее, чем его производительность, то есть пять маломощных процессоров будут дешевле чем один, впятеро более быстрый.

Многопроцессорная система характеризуется двумя или более процессорами имеющими доступ к участку либо всей физической памяти (Рис. 11).

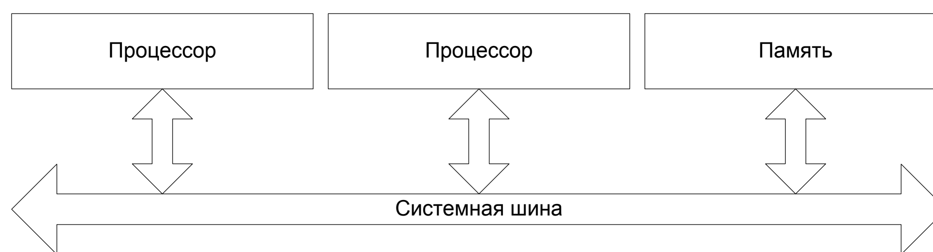

Рис. 11 Многопроцессорная система

Для предотвращения узких мест, процессоры относительно независимы и не находятся в отношениях ведущий-ведомый. Если процессоры одинаковы, они могут разделять не только данные, но и код, такая система называется **симметричной**. Традиционно с ней связывают и общую глобальную память, хотя это и не является обязательным требованием.

Если процессоры различаются, система называется **асимметричной**, поскольку, несмотря на общие данные, каждый процессор выполняет свою программу, которая не может быть разделяемой из-за отличий в системе команд.

Наличие нескольких процессоров в системе приводит к значительным изменениям в технике программирования. Однопроцессорная система, несмотря на вносимую прерываниями асинхронность, в любой момент времени выполняет только какую-то одну команду. Многопроцессорная же система обладает свойством настоящего параллелизма: несколько программ могут работать одновременно.

Претерпевает изменения также и механизм работы с памятью. Поскольку каждому процессору нужно работать с внешней относительно него памятью, в многопроцессорной системе у каждого процессора имеется свой кэш. Так как кэш хранит копии данных из памяти, при работе с одними и теми же данными, могла бы возникнуть ситуация когда данные находятся в несогласованном состоянии, вызванная тем, что процессоры не видят изменений, производимых другими процессорами. Для того, чтобы избежать такой нежелательной ситуации, процессоры реализуют аппаратный протокол, который позволяет обеспечить глобальную согласованность видения оперативной памяти [3]. Согласованное состояние кэшей называется **когерентностью**, а протокол ее обеспечения - протоколом когерентности. Вместо слежения за шиной современные процессоры используют передачу сообщений, запрашивая друг у друга данные через сеть коммуникаций. Строки кэша, вместо бинарного состояния "действительно/недействительно", содержат дополнительные биты указывающие на разделяемость данных. Подробно о механизмах и протоколах обеспечения когерентности памяти можно прочесть в [3], [4] и [5].

Для систем реального времени важен тот факт, что реализация этого протокола еще больше ухудшает предсказуемость системы, потому что время выполнения программы зависит уже не только от предыдущего кода, выполненного данным процессором, но и от кода выполняемого всеми процессорами. Кроме того, в зависимости от состояния данной ячейки памяти (находится ли она в кэше только этого процессора, или разделяется) выполнение даже простой операции чтения или записи может потребовать довольно сложных действий по согласованию этого действия между процессорами, включающих посылку сообщений и ожидание ответа.

2.2.1 Ложное разделение данных

Если процессор имеет кэш, то доступ к данным во внешней памяти происходит всегда через него, причем содержащиеся в кэше данные имеют размер гораздо больший, нежели доступные программисту элементарные типы данных, определенные для процессора. Кэш содержит линии, блоки непрерывных данных размером порядка 32 или 64 байт. В реализации алгоритмов межпроцессорной синхронизации и когеретности кэш-памяти, биты состояния распространяются именно на линию, а не на конкретные переменные, с которыми работает процессор. Из этих рассуждений следует тот факт, что в том случае, если несколько переменных, доступ к которым возможен с разных процессоров, расположены в одной кэш-линии, то обновление любой из них автоматически влияет и на другие.

Подобная ситуация называется ложным совместным использованием данных или **ложным разделением** (false sharing) и при размещении нескольких разделяемых переменных в пределах одной кэш линии, может серьезно пострадать производительность системы.

Для предотвращения такой ситуации, следует выравнивать подобные чувствительные переменные на размер кэш-линии, для чего нужно пользоваться предоставляемыми для этого возможностями компилятора.

2.2.2 Прерывания в многопроцессорных системах

В отличие от однопроцессорной системы, в многопроцессорной существует неоднозначность, какой из процессоров должен получить и обработать прерывание. Для решения этой проблемы контроллер прерываний разделяется на две части, одна часть взаимодействует с устройствами, другая часть встроена в каждый из процессоров (Рис. 12).

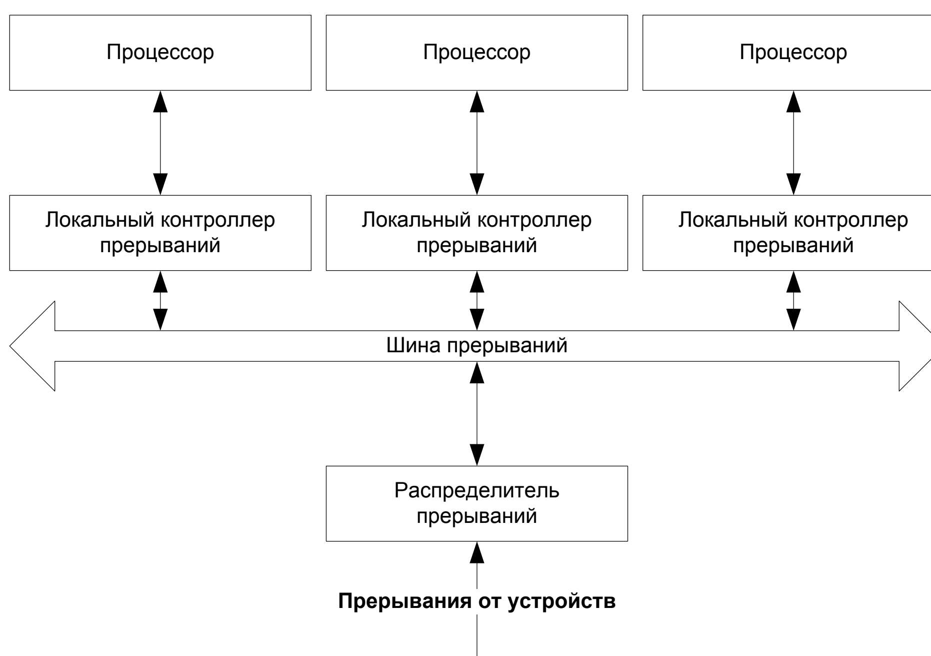

Рис. 12 Архитектура прерываний многопроцессорной системы

Распределитель имеет интерфейсы для программного взаимодействия с ним и используется для назначения обрабатывающих процессоров для источников прерываний. Все контроллеры прерываний связываются общей шиной, посредством которой они могут взаимодействовать не только с распределителем, но и друг с другом. Несмотря на наличие глобальной разделяемой памяти, одной памяти недостаточно для эффективного взаимодействия между процессорами, поэтому обычно существует механизм для обмена прерываниями между ними. В зависимости от используемого распределителя, возможны различные политики доставки прерываний, например, может использоваться циклическое планирование, когда первой прерывание с заданным вектором посылается процессору 0, следующее - процессору 1, затем 2, 3 и так далее. Возможна жесткая привязка вектора к процессору, а также доставка прерывания сразу нескольким процессорам.

2.2.3 Атомарность в многопроцессорных системах

В связи с тем, что процессоры работают независимо друг от друга, запрет прерываний влияет только на текущий процессор, и, следовательно не предотвращает возможность выполнения обработчика прерывания одновременно с кодом критической секции. Кроме того, процессорам нужно как-то синхронизировать доступ к разделяемым ресурсам. Для решения этой проблемы существуют так называемые **атомарные операции**.

Атомарные операции - специальные инструкции, которые выполняют некоторое составное действие таким образом, что это действие не может быть прерывано и никакой другой процессор (или обработчик прерывания на текущем процессоре) не может увидеть промежуточный результат. Большинство таких операций включают в себя 3 стадии, чтение, модификацию и запись нового значения. Из-за этого все операции, имеющие такую семантику называются **RMW-операциями** (Read-modify-write operations).

Простейшим примером атомарной составной RMW-операции, которая поддерживается многими процессорами является SWAP. Будучи представленной как функция C, ее прототип выглядит следующим образом:

```
int atomic_swap(volatile int*, int);
```

В качестве аргументов она принимает указатель на ячейку памяти и значение. В результате выполнения операции, указанное значение атомарно записывается по данному адресу, а функция возвращает предыдущее значение. Псевдокод команды (все содержимое функции выполянется атомарно!):

```
int atomic_swap(volatile int* ptr, int new_value)
{
  int old_value = *ptr;
  *ptr = new_value;
  return old_value;
}
```

Возможности SWAP довольно ограниченные, в частности, с ее помощью довольно трудно

реализовать счетчики, поэтому другая часто используемая операция: сложение с обменом ADD (называемая также fetch-and-add).

```
int atomic_add(volatile int*, int);
```

В качестве аргумента указывается число, которое нужно атомарно прибавить к тому, что находится в ячейке памяти. Как и предыдущая операция, возвращаемый результат является предыдущим значением ячейки.

```
int atomic_add(volatile int* ptr, int addend)
{
  int old_value = *ptr;
  *ptr = old_value + addend;
  return old_value;
}
```

Еще один пример: инструкция TEST_AND_SET, которая является несколько частным случаем операции SWAP.

```
int atomic_test_and_set(volatile int* ptr)
{
  int old_value = *ptr;
  *ptr = 1;
  return old_value;
}
```

Упомянутые атомарные операции выполняют над памятью довольно простые действия, реальные задачи часто требуют выполнения нескольких действий атомарно, например вставка элемента в двусвязный список требует модификации двух указателей. Для решения этой задачи часто используются так называемые **спин-блокировки** или спинлоки (spinlock). Идея заключается в том, что критическая секция реализуется за счет того, что только один процессор может выполнять некоторый участок кода, по аналогии с тем, как это происходит с критическими секциями в однопроцессорных системах. В качестве объекта блокировки используется ячейка памяти.
Спин-блокировка имеет два метода, acquire и release.

```
void spinlock_acquire(volatile int* lock);
void spinlock_release(volatile int* lock);
```

Несколько процессоров могут попытаться ее захватить, но только одному удастся это сделать, остальные процессоры будут находиться в активном цикле ожидания до тех пор, пока процессор захвативший блокировку явным образом ее не отпустит с помощью вызова release.

```
volatile int lock = 0;
```

```
spinlock_acquire(&lock);
<код критической секции>
spinlock_release(&lock);
```

Хотя существуют возможность реализовать спин-блокировки используя только чтение и запись памяти, (например см. алгоритм Петерсона) подобные алгоритмы плохо масштабируются на большое количество процессоров, поэтому на практике спин-блокировки реализуются с помощью атомарных операций. Например, с использованием операции SWAP, спин-блокировка могла бы быть реализованной следующим образом:

```
void spinlock_acquire(volatile int* lock)
{
  while (atomic_swap(lock, 1) == 0) ;
}

void spinlock_release(volatile int* lock)
{
  *lock = 0;
}
```

Атомарная инструкция используется для установки ячейки в условное значение "занято". Для освобождения блокировки используется запись в нее условного значения "свободно".

В отличие от однопроцессорных систем, реализация атомарных RMW-операций требует специальной аппаратной поддержки. Самый простой путь обеспечения атомарности - блокировка доступа к памяти для остальных процессоров на все время выполнения операции. Ранние процессоры использовали для этой цели блокировку системной шины, что позволяло предотвратить доступ к данным не только со стороны других процессоров, но и со стороны устройств через механизм прямого доступа к памяти. С ростом количества процессоров, это решение становилось все менее эффективно, потому что шина становилась узким местом для производительности.

Более продвинутые системы используют для реализации атомарных операций протокол когерентности кэш-памяти. Хотя общая шина может отсутствовать, через сеть межсоединений процессоры передают друг другу информацию о доступах в память, что позволяет им своевременно обновлять и поддерживать в актуальном состоянии кэш-память, а также определять, владеет ли еще какой-либо процессор данной копией данных в своем кэше. Для выполнения атомарной операции процессор вначале добивается эксклюзивного владения данными, а затем выполняет над своим кэшем RMW-операцию.

Последние разработки в области аппаратного обеспечения предусматривают обобщение атомарных операций на данные, имеющие размер более одной ячейки, формируя тем самым аппаратную транзакционную память. Подобным функционалом обладает, например, процессоры Intel Core с микроархитектурой Broadwell и старше. При таком подходе записи, составляющие транзакцию накапливаются в кэше, и могут быть

применены одной неделимой операцией. Транзакционная память удобна и позволяет реализовывать сложные атомарные структуры данных с минимальными накладными расходами. Главным минусом является то, что с ростом объема памяти, задействованной в транзакции, снижается вероятность успеха.

2.2.4 Неблокирующие алгоритмы

Хотя спин-блокировки позволяют создавать критические секции, очевидно, такое решение страдает не очень хорошей масштабируемостью. Чем больше в алгоритме критических секций, тем больше другие процессоры будут тратить времени на ожидание. Начиная с какого-то момента, блокировка станет бутылочным горлышком, поэтому особый интерес представляют алгоритмы, которые не используют такие блокировки, а пользуются только атомарными инструкциями. Эти алгоритмы делятся на три основных типа: obstruction-free, lock-free и wait-free.

Obstruction-free - самая слабая гарантия: процессор закончит выполнение алгоритма за ограниченное время, при условии, что остальные процессоры не будут обращаться к разделяемым данным [6]. Второй тип - lock-free - конечное количество процессоров закончат выполнение алгоритма за конечное время (по крайней мере 1 процессор будет продвигаться в каждый момент времени). Наконец, wait-free - конечное число процессоров закончат выполнение алгоритма за фиксированное время (все процессоры в каждый момент совершают прогресс).

Действия, которые необходимо выполнять атомарно, специфичны для каждого алгоритма, поэтому возникает вопрос, какие элементарные операции должны поддерживаться процессором, для того, чтобы на таком процессоре мог быть реализован любой неблокирующий алгоритм.

Теоретические исследования привели к тому, что любой wait-free алгоритм может быть реализован тогда и только тогда, когда существует wait-free решение так называемой **проблемы консенсуса** [7, 8]. Консенсус - это алгоритм который содержит единтвенный метод decide.

```
int decide(int);
```

Консенсус работает так: каждый процессор указывает в качестве аргумента некоторое уникальное число (например, свой номер) и получает результат. Результат должен удовлетворять следующим ограничениям:

- он должен быть равен одному из аргументов, который передан одним из процессоров (valid)
- все процессоры вызвавшие метод должны получить один и тот же результат (consistent)

Определяется также число называемое **числом консенсуса** (consensus number). Это максимальное количество процессоров, для которого решаема задача консенсуса, используя данный набор атомарных инструкций.

Оказывается, что все перечисленные выше атомарные операции имеют конечное число

консенсуса, то есть не являются универсальными. Например, легко показать, что для SWAP число консенсуса равно 2.

Простейшая инструкция, обладающая бесконечным числом консенсуса, то есть достаточно универсальная для реализации любого wait-free алгоритма называется **сравнение с обменом** (compare and swap, CAS).

CAS принимает три аргумента, указатель на ячейку памяти, ожидаемое значение и новое значение. Инструкция атомарно проверяет значение по указателю, если оно равно ожидаемому значению, ячейка устанавливается в новое значение, в противном случае значение не меняется. Функция возвращает предыдущее значение ячейки, по которому можно судить, была ли операция успешной, если вернулось предыдущее значение, это значит , что операция сработала и память была обновлена. Псевдокод на С:

```
int CAS(volatile int* ptr, int comparand, int new_value)
{
  int old_value = *ptr;
  if(old_value == comparand)
  {
    *ptr = new_value;
  }
  return old_value;
}
```

CAS очень широко применяется при разработке неблокирующих алгоритмов. Нетрудно показать, что все обсуждавшиеся выше операции реализуемы на основе CAS. Например, lock-free алгоритм для fetch_and_add мог бы быть реализован следующим образом:

```
int old_value;
int result;

do
{
  old_value = *ptr;
  result = CAS(ptr, old_value, old_value + N);
}
while(result != old_value);
```

Типичный шаблон использования CAS включает в себя цикл, называемый также CAS-loop, в котором производятся попытки атомарно обновить значение. Хотя, формально говоря, все такие алгоритмы являются lock-free, а не wait-free, на практике этого достаточно (wait-free алгоритмы возможны, но более сложны). Если CAS завершился неудачей, это значит что у кого-то другого процессора он завершился успешно, то есть тот процессор совершил прогресс, а значит, вся система в целом совершила прогресс.

Подобным образом могут быть реализованы многие операции, которые, в отличие от fetch-and-add, не имеют аппаратной поддержки ни в одном процессоре. Например условное сложение - прибавление слагаемого только в том случае, если результат не

превысит заданный порог, атомарные логические инструкции, такие как AND, OR и так далее.
Тем не менее, у CAS есть и недостатки. Один из них - так называемая проблема ABA. Довольно часто в качестве аргумента CAS используется указатель, в этом случае возможна ситуация, когда численно указатель остался тот же, но указывает уже на другой объект (такое могло случиться, если изначальный объект был удален а память освобождена, затем она повторно выделена для нового объекта). В таком случае, CAS может быть успешным в тех случаях, когда это нежелательно (когда значение менялось, но вернулось к прежнему численному значению). Для того, чтобы обойти это препятствие, некоторые процессоры, например intel x86, поддерживают **двойной CAS**. Его отличие от описанного выше в том, что аргумент может быть вдвое большей длины. Это позволяет, в случае использования указателя в качестве операнда, снабдить его дополнительной информацией, тегом или счетчиком, который находится в соседней ячейке памяти, а CAS, таким образом, обновляет сразу оба значения. В этом случае, численно равные значения, но отличающийся тег не даст операции завершиться успешно.
Хотя двойной CAS и решает проблему ABA, он не очень удобен в использовании, поэтому другие процессоры, такие как, например ARM или MIPS, реализуют другой подход, называемый LL/SC.
Процессор поддерживает вместо одного CAS пару инструкций, называемых Load Link (LL) и Store Conditional (SC). С помощью первой инструкции процессор читает ячейку памяти и помечает ее для мониторинга - отслеживания обращения к ней других процессоров. Вторая инструкция используется для сохранения обновленного значения.

```
int load_link(volatile int* ptr)
{
  int old_val = *ptr;
  start_monitoring(ptr);
}

bool store_conditional(volatile int* ptr, int new_value)
{
  if(no_accesses_to_monitored_memory())
  {
    *ptr = new_value;
    return true;
  }
  return false;
}
```

Типичный алгоритм, использующий эти инструкции, так же как и CAS, предполагает использование цикла.

```
do
{
  int old_val = load_link(ptr);
```

```
  ... модификация значения...
}
while(!store_conditional(ptr, new_val));
```

В отличие от CAS, SC-операция закончится неудачей даже в том случае, если значение, для которого осуществляется мониторинг, было заменено на численно такое же, поскольку отслеживается именно доступ к ячейке со стороны других процессоров.

2.2.5 Модель памяти

Вместе с переупорядочиванием инструкций и кэшированием, процессор применяет также и другие методы повышения быстродействия. Один из методов - оптимизация работы с памятью. Набор правил, в соответствии с которыми осуществляются такие оптимизации, называется **моделью памяти**.

Если никакие оптимизации не выполняются, тогда все процессоры обращаются к памяти в том порядке, в котором обращается программа, а все обращения в целом формируют последовательность, которая получается путем чередования запросов от всех процессоров. Такая модель называется **последовательно консистентной** (sequentially consistent).

Современные процессоры, поддерживающие спекулятивное исполнение, в большинстве случаев, применяют оптимизации, которые приводят к нарушению порядка операций и, таким образом, их модель памяти не является последовательно консистентной. Например, если обращения в память дороги из-за потенциальных проблем с когерентностью, можно сохранять записываемые значения в локальный буфер и продолжать выполнение программы параллельно с обращением в кэш. Такой буфер называется store buffer или write buffer [4]. Кроме того, процессор может реализовывать механизм предвыборки данных в кэш, которые могут понадобиться в дальнейшем.

С точки зрения самого процессора, он поддерживает последовательность обращений, то есть при попытке чтения значения, записанного во write buffer, оно будет прочитано оттуда, а не из кэша. То есть обеспечивается выполнение алгоритма таким образом, что все оптимизации происходят прозрачно. Но другие процессоры в системе не имет доступа к буферам сохранения, а значит, видят последовательность обращений такой, какой она появляется на внешней шине. Поэтому при выполнении параллельного алгоритма на нескольких процессорах, могут проявиться программно видимые последствия оптимизаций, которые требуют определенных корректировок в самом алгоритме.

Проблемы возникают главным образом в тех случаях, когда между не связанными ячейками памяти возникают неявные связи, обусловленные алгоритмом. Например, если используется некоторый флаг готовности, как показано во фрагменте ниже.

```
global_struct = malloc(...);
...
ready_flag = 1;
```

С точки зрения последовательно консистентной модели памяти, любой процессор,

который увидел значение флага равным 1, может предполагать, что переменная global_struct установлена и обращение к ней не вызовет ошибок:

```
if(ready_flag)
{
  // работа со структурой
}
```

На практике, запись флага может быть переупорядочена таким образом, что другой процессор увидит флаг равный 1 до того, как будут присвоены переменные до этого, и, следовательно, обращение может вызвать ошибку. Для избежания таких ситуаций должны использоваться **барьеры памяти** - специальные инструкции, которые вводят ограничения на то, как и какие оптимизации процессор может применять в данном участке кода, например ограничения на переупорядочивание.

Модель памяти является частью архитектуры команд и поэтому специфична для каждого процессора, так же как количество и функционал доступных барьеров.

Для систем жесткого реального времени использование процессоров с подобными оптимизациями нежелательно, однако разрабатываемое ПО, в том числе ядро ОС, должно оставаться функционально корректными при работе в любой системе, так как возможна ситуация, что оно может использоваться и в системах с менее жесткими требованиями, поэтому все алгоритмы, на которые могут влиять подобные оптимизации, должны их учитывать.

В качестве примеров можно привести барьеры, используемые в процессорах x86: инструкции LFENCE, SFENCE и MFENCE.

- LFENCE - завершение всех чтений из памяти перед выполнением следующей инструкции
- SFENCE - завершение всех записей перед выполнением следующей инструкции
- MFENCE - завершение всех обращений в память до выполнения следующей инструкции

В процессорах ARM используется другой набор инструкций DMB, DSB, ISB:

- DMB - завершение всех явных обращений в память перед следующим обращением
- DSB - завершение всех явных и неявных обращений в память перед выполнением следующей инструкции
- ISB - очистка конвейера перед выполнением следующей инструкции

Команды приведены без аргументов, что трактуется процессором как аргументы по умолчанию. Каждая из этих инструкций может содержать дополнительные опции, например, может влиять только на команды сохранения данных, но не записи (DMB ST) и тому подобное.

Поэтому, выше приведенный пример кода, должен выглядеть как:

```
global_struct = malloc(...);
...
mem_barrier(); // завершить предыдущие операции перед установкой
```

```
                 // флага готовности
ready_flag = 1;
```

То же самое касается и использования флага:

```
flag = ready_flag

mem_barrier(); // чтение данных только после завершения
               // операции чтения флага
if(flag)
{
  // работа со структурами
}
```

Особую важность этот вопрос имеет при использовании отображенных на память устройств, которые требуют определенной последовательности доступа к своим регистрам. Для их использования требуется либо отключать кэширование и оптимизации для данного региона памяти, либо использовать барьеры таким образом, чтобы не нарушить логику работы устройства.
В зависимости от алгоритма, могут применяться различные барьеры, разной степени строгости. Подробнее об этом можно прочесть в [3] и [9].

2.3 Защита

2.3.1 Режимы исполнения

Процессор может выполнять несколько задач используя разделение времени. Эти задачи могут иметь разные уровни ответственности и сложности. Например, поддержка сети и файловой системы - довольно сложные по объему задачи, которые могут быть не связаны с основным функционалом устройства (могут использоваться для мониторинга и логирования). При этом, ошибки в этих программах могут полностью нарушить работу устройства. Количество ошибок растет с ростом объема программы.
Одно из возможных решений: ввести привилегированный и непривилегированный режимы исполнения задач. Данная возможность на большинстве процессоров поддерживается аппаратно. Задаче, выполняемой в непривилегированном режиме, разрешено выполнять только те действия, которые не могут привести к нарушению работы системы, а для выполнения потенциально опасных действий она обращается к привилегированной программе.
Ограничивается использование памяти (и устройств, отображенных на память) выполнение инструкций, которые могут повлиять на выполнение привилегированной программы (например, запрет прерываний). Выполнение этих ограничений контролируется аппаратно, перед выполнением каждой инструкции. Для поддержки различных режимов выполнения, процессор должен предоставлять возможности для контролируемых запросов от непривилегированного приложения к привилегированному и обратно, по завершению запроса.
Этот подход позволяет реализовать встраиваемое приложение в виде небольшой

доверенной части, вся остальная часть системы может быть реализована как непривилегированная программа. Аварийное завершение работы такой программы не приведет к остановке всего устройства.

В качестве механизма, обеспечивающего контроль доступа, используются исключения. Предполагается, что обработчики должны быть установлены привилегированной программой, которая, в случае возникновения исключения перехватывает управление и решает, что делать далее.

Помимо прочего, разделение программы на части с разными привилегиями упрощает отладку и поддержку. Поскольку выполнение ограничений контролируются аппаратно, а исключения позволяют узнать место, где происходят нарушения, возникающие ошибки могут быть быстро локализованы и исправлены.

2.3.2 Переключения между режимами.

Для того, чтобы непривилегированная программа могла работать, требуется способ, взаимодействия с оборудованием, для чего используется привилегированная программа, которой нужно передать управление контролируемым образом.

Способы переключения между режимами зависят от процессора и его архитектуры, но чаще всего используется механизм исключений, как и для контроля ошибок. Непривилегированная программа не может напрямую вызывать код привилегированной, вместо этого используется специальная инструкция, называемая системным вызовом или **программной ловушкой** (software trap). Она позволяет вызвать установленный заранее обработчик, притом что его адрес устанавливается привилегированной программой. Таким образом, непривилегированная программа может передать управление только в контролируемые точки привилегированной. Обработчик может проанализировать запрос и выполнить или отклонить его. Передача аргументов может происходить как через память, так и через регистры. В качестве примера инструкции используемой для системных вызовов можно привести инструкцию SVC, процессоров ARM.

При ее выполнении, происходит переключение в привилегированный режим с сохранением в специальных регистрах текущего состояния процессора (адрес возврата и регистра состояния CPSR) и начинает выполняется обработчик, предназначенный специально для этой инструкции, который имеет свое выделенное место в таблице прерываний (Рис. 13).

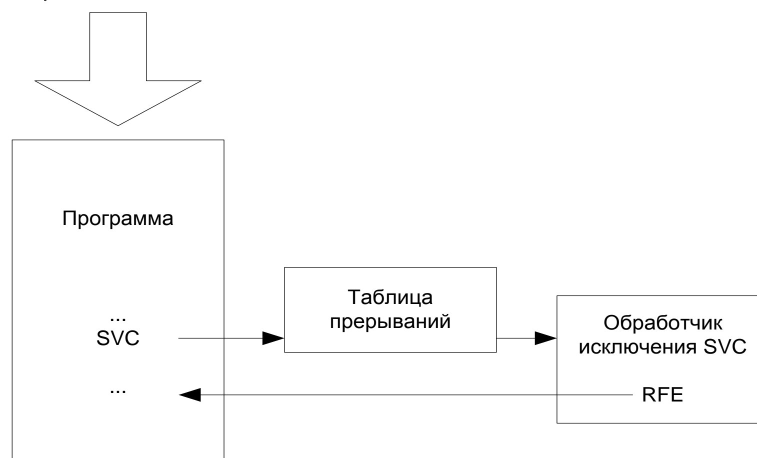

Рис. 13 Механизм системного вызова ARM

После обработки, функция-обработчик должна выполнить инструкцию LDM или RFE, чтобы восстановить сохраненные регистры состояния процессора и тем самым переключить режим обратно в непривилегированный.
Привилегии текущего кода обычно хранятся в специальных регистрах процессора, поэтому понижение привилегий может возникать либо как часть процедуры переключения контекста, либо может быть достигнуто записью в эти специальные регистры (некоторые процессоры поддерживают такую возможность).
Обработчики прерываний должны взаимодействовать с устройствами, поэтому они, как правило, выполняются в привилегированном режиме процессора.
Некоторые процессоры поддерживают больше чем два уровня привилегий. Например, intel x86 поддерживает 4 уровня. Это позволяет еще более точно проводить границы между различными частями программ, но очень редко используется на практике из-за трудностей портирования на процессоры, имеющие меньшее число привилегий.

2.3.3 Права доступа к памяти

Привилегированная программа находится в памяти, так же как и непривилегированная, поэтому доступ к памяти также должен быть защищенным для того чтобы предотвратить возможное повреждение данных со стороны непривилегированных программ в результате ошибок в них.
Для этого могут использоваться различные техники, в самом простом случае процессор содержит дополнительные регистры, в которых указывается базовый адрес и граница доступа к памяти для непривилегированного приложения (Рис. 14).

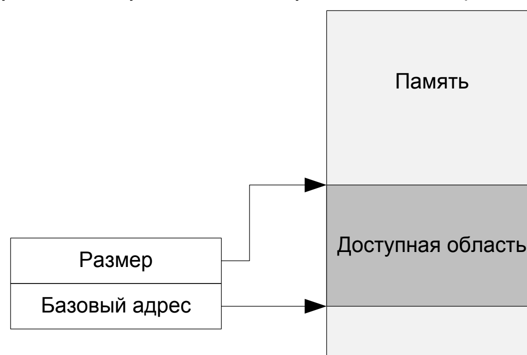

Рис. 14 Контроль доступа к памяти с использованием регистров региона

Во время выполнения непривилегированной программы любые обращения в память за пределами указанных границ вызывают исключение, что и обеспечивает необходимую защиту. Сами регистры, управляющие доступом к памяти, доступны только для привилегированной программы. При переключении программ возможно также переключение содержимого этого регистра, то есть программы будут изолированы не только от привилегированной части но и друг от друга. Главный недостаток этого подхода - возможен только один регион памяти, куда может быть разрешен доступ со стороны непривилегированного приложения.
Развитие этого метода - **устройство защиты памяти** (memory protection unit, MPU). MPU тесно связан с ядром процессора и содержит набор регистров, описывающих несколько регионов и права доступа к ним. Количество регионов обычно невелико, в пределах 8-16,

но этого достаточно для большинства приложений, поскольку позволяет установить различные права для кода, данных и прочего.

Высокопроизводительные процессоры могут содержать **устройство управления памятью** - (memory management unit, MMU). MMU - гораздо более продвинутое устройство, по сравнению с MPU. Идея заключается в отделении виртуального адресного пространства памяти, с которым работают программы, от физического адресного пространства. Память разбивается на набор блоков, фиксированного размера, называемых **страницами**. MMU задает отображение страниц виртуального адресного пространства на физические страницы.

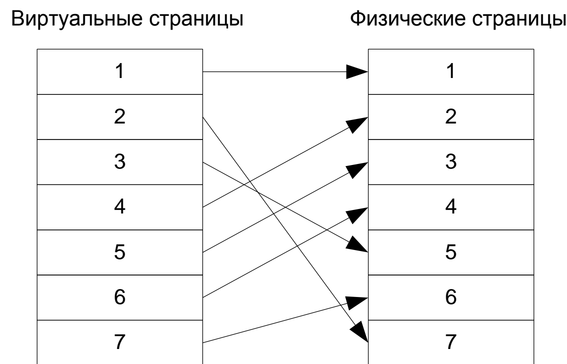

Рис. 15 Виртуальное и физическое адресное пространство

Помимо задания этого отображения, для каждой страницы задаются параметры доступа, что позволяет очень гибко настраивать адресное пространство приложений. Основное назначение MMU - предоставить каждой программе изолированное адресное пространство и реализовать так называемую **виртуальную память** - возможность использования большего объема физической памяти, чем имеется в компьютере.

Помимо прав доступа, для каждой страницы указывается ее наличие или отсутствие. попытки обращения по адресу, который находится в странице, помеченной как отсутствующая вызывает исключение, поэтому ОС оставляет в памяти только наиболее часто используемые приложениями страницы, а все остальные выгружать на внешние носители, такие как диски. При обращении к отсутствующей странице, обращение перехватывается, страница загружается с диска и выполнение возобновляется. Этот механизм называется **свопингом** (swapping).

В системах реального времени свопинг не используется из-за неопределенности, которую он вносит в процесс выполнения программы, поэтому используется только часть возможностей MMU, связанные с настройкой защиты и отображением адресов.

Физическая реализация MMU включает в себя буфер трансляции, называемый **буфером TLB** (Translation Look aside Buffer). Любой адрес, по которому обращается программа, это может быть явное обращение к данным или выборка инструкций, проходит процедуру трансляции - поиска записи в TLB, которая по данному витуальному адресу определяет физический и далее этот физический адрес используется для доступа к памяти. Структура буфера TLB похожа на структуру кэша и по ключу (виртуальному адресу) позволяет получить значение - физический адрес (Рис. 16).

Состояние и флаги	Виртуальный адрес страницы	Физический адрес страницы
...	...	...
Состояние и флаги	Виртуальный адрес страницы	Физический адрес страницы

Рис. 16 Структура TLB

Поскольку поиск в TLB должен выполняться быстро, и, по возможности, параллельно по всему его содержимому, объем TLB ограничен несколькими десятками записей, которые не описывают адресное пространство полностью. Если запись транслирующая даный адрес не найдена в TLB, начинается процедура его заполнения, которая реализуется с помощью двух основных способов.

В первом случае (и наиболее распространенном) адресное пространство описывается с помощью расположенных в памяти таблиц, которые должны быть подготовлены системным ПО. Процессор содержит специальный регистр, который хранит указатели на таблицу трансляции. При неудачном обращении к TLB, процессор аппаратно находит в памяти эти таблицы, по адресу обращения находит в них нужную запись и загружает ее в TLB. Весь этот процесс происходит прозрачно для пользователя, а буфер TLB скрыт от него по аналогии с кэшем - им полностью управляет процессор (Рис. 17).

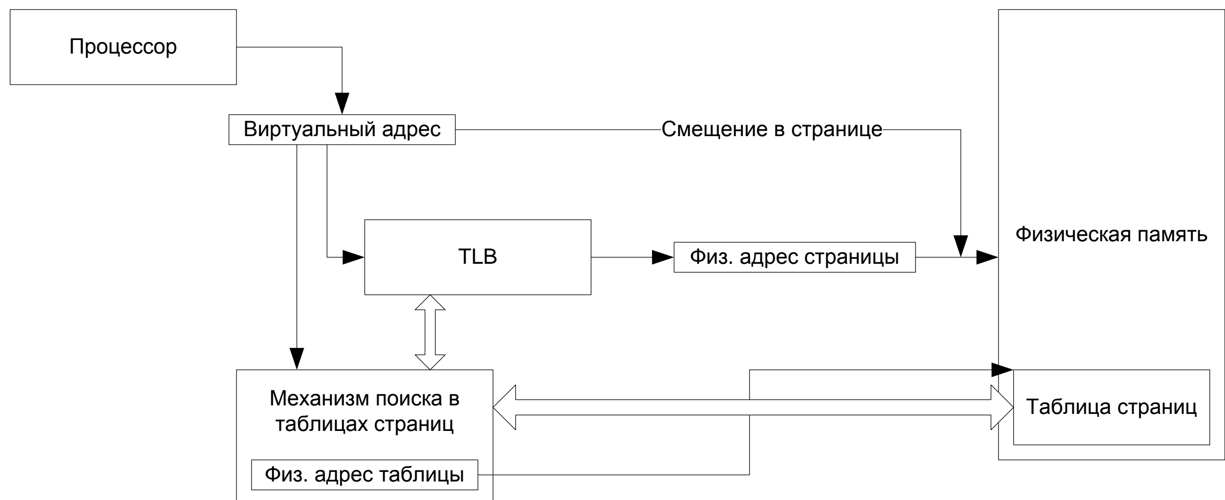

Рис. 17 Механизм работы TLB

Часть адреса является индексом в таблице и физический адрес формируется как показано на (Рис. 18)

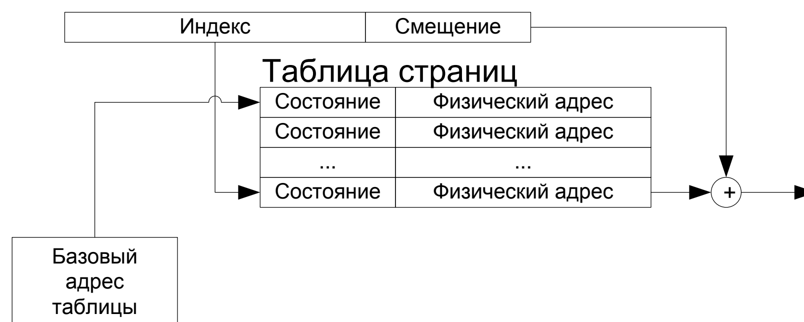

Рис. 18 Формирование физического адреса страницы с помощью таблицы страниц

Разумеется, таблица сама должна постоянно присутствовать в памяти, чтобы сделать любую трансляцию возможной. При небольшом объеме памяти, таблица страниц может

занимать существенную ее часть. Например таблица описывающая 4Гб памяти в виде страниц по 4Кб, содержит 1048576 страниц, при размере записи в таблице в 4 байта, получается 4Мб только на таблицу страниц. Для сокращения этого размера используется подход с многоуровневыми таблицами. Адрес разбивается на несколько частей, например на 3, как показано на (Рис. 19).

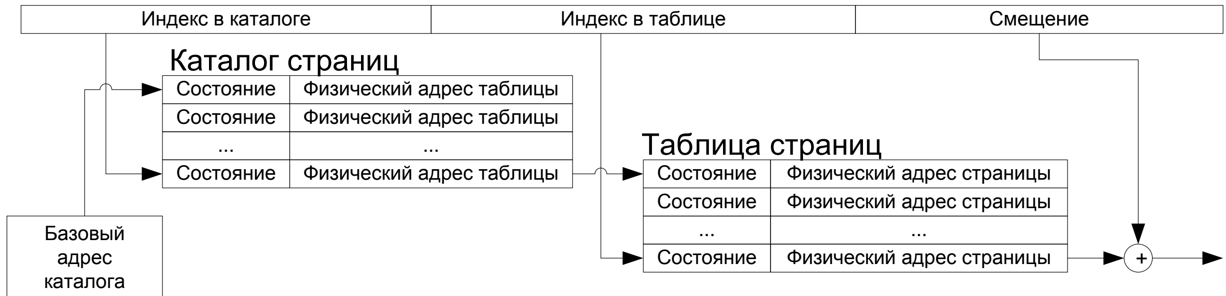

Рис. 19 Многоуровневая таблица страниц

Верхняя часть адреса используется как индекс в таблице верхнего уровня, называемой каталогом страниц. Средняя часть - индекс в таблице страниц, базовый адрес которого определяется с помощью записи каталога. Такой подход позволяет помечать как отсутствующие не только страницы, но и таблицы, через записи каталога страниц. Если в системе, к примеру 256Мб физической памяти, для этого потребуется 65536 4-килобайтных страниц. Если разделить поровну оставшиеся после 12-битного смещения в странице 20 бит, на каталог и таблицу, получается что таблица страниц содержит 1024 записи, и, следоваетельно, для описания 65536 страниц необходимо 64 таблицы и записи каталога. Поскольку виртуальное адресное пространство всегда должно быть определно полностью, каталог должен присутствовать всегда, поэтому как минимум 4Кб на него потребуются в любом случае. Плюс к тому, каждая из 64 таблиц, занимает также по 4Кб. В итоге получается 260Кб, вместо 4Мб, что является довольно ощутимой экономией.

В случае большого адресного пространства 64-битных систем, количество уровней трансляции может быть более трех, потому что адресное пространство столь огромно, что для полного его описания нужны тысячи терабайт только для таблицы страниц!

Применяются также подходы связанные с увеличением размера страницы, и, как следствие, уменьшением размера каталогов и таблиц. Например, процессоры ARM позволяют использовать страницы размером в 4Кб, 64Кб, 1Мб и 16Мб. Как нетрудно подсчитать, в последнем случае таблица содержит всего 256 4-байтных записей, то есть занимает 1Кб. Но такая большая гранулярность отображения и установки прав не всегда приемлема для приложений, поэтому требуются компромиссы.

Полностью аппаратный подход к трансляции страниц хорош прозрачностью: подготовив таблицы страниц ПО не нужно заботиться о TLB, всё выполняется процессором автоматически. Минус заключается в том, что из-за того, что процессор знает о таблицах, их формат фиксирован и предопределен, поэтому даже в том случае, когда адреса трансляции связаны некоторой функцией (например - 1 в 1, виртуальный адрес равен физическому), все равно требуется подготовка и заполнение всех таблиц, в том числе многоуровневых. Второй минус является следствием первого: из-за трансляции с

помощью таблиц, процессор в случайные моменты времени (в зависимости от размера TLB и его содержимого) обращается в память и выполняет довольно много обращений, связанных с поиском записи в многоуровневых таблицах. Поэтому в системах реального времени должны применяться специальные меры при использования MMU, такие как, например, создание в TLB невыгружаемых записей описывающих регионы, где расположена программа с требованиями жесткого реального времени, если процессор позволяет такие записи создавать.

Второй подход к трансляции предполагает программную обработку исключения отсутствия страницы. В этом случае ПО может самостоятельно определять структуры таблиц и контролировать выполнение трансляции. Этот подход лишен недостатков первого, но приводит к усложнению ПО, а также страдает от более низкой производительности.

При использовании программной трансляции TLB делается доступным для программиста с помощью соответствующих команд загрузки в него данных. На ПО возлагается ответственность за то, чтобы TLB всегда содержал отображение виртуального адреса на физический для всех регионов, к которым может обращаться программа. В случае, если искомое отображение в TLB отсутствует, возникает исключение, которое должно загрузить в TLB требуемое программой отображение, при этом обработчик исключения либо должен находиться в регионе, который уже был описан в TLB до включения трансляции адресов, либо должен находиться в регионе памяти, для которого не выполняется трансляция адресов.

Существует также промежуточный вариант между MPU и MMU, называемый FMT - fixed mapping translation. В этом случае, чип содержит жестко заданные диапазоны доступа с разными правами к оперативной памяти. Границы диапазонов можно задавать как и в случае MPU с помощью набора выделенных для этого регистров. Никаких таблиц трансляции не используется, но в отличие от MPU, виртуальное и физическое адресное пространства разделены. К примеру, все адресное пространство может быть разделено на 2 или более диапазонов таким образом, что начиная с адреса 0x80000000 отображается та же самая физическая память, что и по адресу 0, но при обращении по адресам в нижнем диапазоне учитывается информация в граничных регистрах, которые не позволяют обращения за пределами региона, а при обращениях в верхних адресах эти проверки не выполняются.

В случае многопроцессорной или многоядерной системы каждый процессор имеет свой блок защиты памяти или трансляции адреса, а также TLB. Если операционной системой используется одна таблица страниц и выполнение задач привело к изменению таблицы, необходимо с помощью прерываний уведомить остальные процессоры о необходимости обновить буферы TLB, чтобы поддерживать глобальную согласованность данных. Возможны варианты с аппаратной поддержкой когерентности TLB, но они менее распространены.

2.3.4 Инвертированные таблицы страниц

Прямой доступ к TLB в случае использования программной трансляции, дает возможность

реализовать хранение информации об отображении виртуальных адресов на физические произвольным образом. Например, если правило отображения задается некоторой простой функцией, скажем, от 0 до 0x80000000 - прямое отображение виртуальных адресов на физические, с 0x80000000 и далее также отобразить с 0 страницы. Разумеется, правила могут быть и более сложными. В этом случае, логика трансляции может быть полностью реализована внутри обработчика исключения отсутствия страницы без необходимости хранения какой-либо дополнительной информации в таблицах.

В настольных компьютерах и серверах отображение между виртуальными и физическими адресами непрерывно меняется в результате выделения, освбождения памяти и свопинга, поэтому хранение таблицы отображения - единственный разумный выход, но во встраиваемых системах закон отображения может быть известен заранее, поэтому использование программного управления TLB может быть выгодно, к тому же программная трансляция отличается предсказуемым и детерминированным поведением.

Для систем с большим адресным пространством может использоваться так называемая инвертированная таблица страниц - хранение записей описывающих не виртуальные, а физические страницы (откуда и происходит название). В таком случае, виртуальный адрес уже не может использоваться для поиска трансляции и приходится пользоваться различными вариантами хэширования. То есть обработчик страничного исключения должен найти в структуре данных, хранящей записи о физических страницах элемент, которому соответствует виртуальный адрес, вызвавший исключение. Это, разумеется, менее детерминировано, а также обладает меньшей производительностью, чем поиск в таблицах.

2.4 Типичный микроконтроллер

Во встроенных системах часто используются микроконтроллеры. Микроконтроллер это микросхема, которая содержит на одном кристалле процессорное ядро, флеш-память, оперативная память и периферию. После разработки ПО на инструментальном ПК, оно загружается во флеш-память. После подачи питания на контроллер, выполнение кода начинается с определенного адреса во флеш-памяти. Устройства и флеш отображаются на адресное простаранство оперативной памяти.

В качестве периферийных устройств часто содержатся коммуникационные контроллеры RS-232, CAN, USB, Ethernet. Кроме того, часто присутствуют АЦП/ЦАП и порты ввода-вывода, который позволяет считывать устанавливать пины контроллера в бинарные значения соответствующие определенному напряжению.

Среди различных периферийных устройств особое значение имеют таймеры. Как правило, это устройства содержащие счетчик, при достижении которым запрограммированного значения процессору подается сигнал прерывания. Это устройство позволяет встроенным системам учитывать время и является внутренним источником прерываний, по которым может происходить анализ других датчиков.

В более производительных системах может использоваться внешняя динамическая память, а также MPU и MMU. На кристалле контроллера могут содержаться также несколько процессорных ядер, формирующие симметричную или асимметричную многопроцессорную систему.

Во время старта управление должно получить системное ПО, которое настраивает аппаратуру и далее передает управление приложению.

2.5 Резюме

Несмотря на все различия, абсолютное большинство микропроцессоров, используемых во встроенных системах, построены по одним и тем же принципам, поэтому, вместо обсуждения конкретного устройства, можно рассматривать абстрактный микропроцессор, который взаимодействует с памятью и устройствами ввода-вывода.

Для реализации реакции на непредсказуемые внешние события, как правило применяются аппаратные прерывания, синхронизация которых с приложением выполняется с помощью механизмов запрета прерываний. Для повышения производительности используется кэширование памяти, которое приводит также и к снижению предсказуемости, поэтому этот факт необходимо учитывать в системах реального времени.

Многопроцессорные системы, несмотря на повышение надежности и производительности, привносят дополнительные сложности для программиста с точки зрения синхронизации. Особенности таких систем, такие как механизмы синхронизации кэшей также должны учитываться при проектировании систем реального времени.

Для повышения надежности и разделения встроенного ПО на независимые компоненты используются разделение режимов исполнения и аппаратные механизмы защиты памяти, такие как MPU и MMU.

[1] Hennessy, Patterson; "Computer Architecture. A quantitative approach" Fourth edition 2007
[2] Shen, Lipasti; "Modern Processor Design. Fundamentals of superscalar processors." 2005
[3] Paul McKenney; "Memory Barriers: a Hardware View for Software Hackers"
[4] Daniel Sorin, Mark Hill, David Wood; "A Primer on Memory Consistency and Cache Coherence" 2011
[5] David Culler et al.; "Parallel Computer Architecture: A Hardware/Software approach" 1998
[6] Maurice Herlihy et al.; "Obstruction-Free Synchronization: Double-Ended Queues as an Example"
[7] Maurice Herlihy; "Wait-Free Synchronization" 1991
[8] Maurice Herlihy, Nir Shavit; "The Art of multiprocessor programming" 2008
[9] Anthony Williams; "C++ concurrency in action. Practical multithreading." 2012

3 Подходы к проектированию встроенного ПО

Темы главы:

- Архитектура встроенного ПО на основе циклов
- Использование прерываний в качестве платформы
- ОСРВ как каркас для разработки приложений
- Обзор архитектур ОСРВ

3.1 Требования

В отличие от настольных систем, которые являются универсальными платформами для запуска разнообразных прикладных приложений, в области встроенных систем ПО проектируется индивидуально для каждого устройства. Это связано с необходимостью идеального соответствия между аппаратным и программным обеспечением, для достижения высоких показателей производительности и времени автономной работы.

Разработка встроенного ПО начинается с анализа требований и разработки спецификации требований. От этих требований зависит, каким образом должно быть разработано ПО и какие функции на него возлагаются. Например, в случае использования низкопроизводительного процессора (с целью удешевления устройства или снижения энергопотребления), использование ОС или даже языка программирования высокого уровня может быть сопряжено со слишком большими накладными расходами. В качестве примера можно привести 8- и 16-битные микроконтроллеры с размером оперативной памяти, не превышающим 1Кб. В этом случае, встроенное ПО может полностью разрабатываться в рамках одного проекта без использования сторонних инфраструктурных компонентов. Другой пример - требования высокой надежности и полного покрытия тестами всех возможных вариантов исполнения кода зачастую исключают использование прерываний, что также вносит существенные изменения в процесс проектирования и разработки ПО, а также использования сторонних компонентов, таких как ОС.

Несмотря на некоторые отличия, в целом, для встроенных систем и встроенного ПО справедливы все те же критерии качества что и для любого другого ПО: поддерживаемость, расширяемость, производительность и так далее.

3.2 Реализация управляющей логики в приложении

3.2.1 Главный цикл

Одним из самых простых подходов к разработке встраиваемого ПО является использование так называемого главного цикла или суперцикла. Если задачей системы является реагирование на некоторые внешние события, то все ПО можно представить как бесконечный цикл в котором анализируются входные данные и генерируются соответствующие выходные данные. Например, цикл, обрабатывающий данные, поступающие от нескольких сенсоров мог бы выглядеть так:

```
do
{
```

```
  if (sensor1_ready(...))
  {
    process_data_from_sensor1(...);
  }
  ...

  if (sensorN_ready(...))
  {
    process_data_from_sensorN(...);
  }
}
while (1);
```

Цикл состоит из последовательных опросов датчиков и генерации выходных данных. Устройства опрашиваются через регистры, то есть прерывания при таком подходе не используются, следовательно, скорость реакции на асинхронные запросы ограничена частотой выполнения цикла.

Несмотря на свою простоту, метод обладает таким достоинством как хорошая предсказуемость. Анализ состояния сенсоров происходит с фиксированной частотой, которую легко измерить. Из-за отсутствия каких бы то ни было асинхронных вызовов данный метод отличается хорошей анализируемостью. Если количество проверяемых датчиков невелико и одна итерация цикла не занимает слишком много времени, обеспечивается также хорошее время реакции. Но подход с циклом страдает также и от ряда недостатков. Несмотря на то, что метод практически идеален для простых встроенных систем, по мере роста сложности ПО его становится все труднее поддерживать. При росте количества анализиуемых датчиков возрастает время работы цикла и смещается частота опроса. То же самое происходит и при росте сложности функций обработки, вызываемых в качестве реакции на информацию от сенсоров. Все это требует полного цикла тестирования при внесении любых изменений. Другим важным вопросом является энергосбережение. Поскольку не используется никаких механизмов остановки процессора, опрос происходит постоянно, если же события присходят нечасто, например не чаще раза в секунду, подход с главным циклом впустую расходует вычислительные ресурсы процессора.

Хотя разработка ведется на языке высокого уровня и формально говоря цикл является кроссплатформенным, на практике различные устройства обладают различным программным интерфейсом, поэтому без дополнительных усилий и программных прослоек для абстрагирования устройств код получается непереносимым.

Среди других минусов можно отметить отсутствие всякой гибкости ПО. Ветви цикла являются относительно независимыми, поэтому может возникнуть необходимость выполнения некоторых участков цикла на выделенных для этого процессорах. В этом случае, код для каждого из них нужно проектировать индивидуально и практически с нуля.

Все эти соображения приводят к тому что решение об использовании цикла принимается

только тогда, когда либо объем выполняемой в нем работы невелик, например, в случае очень ресурсоограниченного оборудования. либо в случае, когда предсказуемость стоит на первом месте и всеми остальными параметрами системы можно ради этого пожертвовать.

3.2.2 Циклический планировщик

Попытки выделить приоритетные и неприоритетные задачи, улучшить время реакции в случае расширения, а также упростить реализацию конечных автоматов привели к модификации подхода с главным циклом называемым циклическим планировщиком. В литературе он может упоминаться под разными названиями, например планировщик сопрограмм или прото-потоков [1] .

Подход с главным циклом страдает от своей статической природы, все ветви цикла выполняются последовательно. Один из подходов его улучшению заключается в том, чтобы функции-обработчики событий после выполнения возвращали управление не в главный цикл, а в некоторый планировщик, который мог бы выбирать и запускать функции обработки в зависимости от наличия событий, а также от приоритета самих обработчиков.

Пример такого цикла показан ниже:

```
int state = <initial_state>;

do
{
  switch (state)
  {
    case 1:
    process_data_from_sensor1(...);
    break;

    case 2:
    process_data_from_sensor2(...);
    break;

  ...

    case N:
    process_data_from_sensorN(...);
    break;
  }

  state = schedule();
}
while (1);
```

Функция планировщика может быть как простой, вроде опроса датчика и выяснения, какой их них нужно обрабатывать следующим, так и довольно сложным, учитывающим приоритеты и специфику поступления входных сигналов.
Возможны также модификации этого подхода с использованием не фиксированного заранее количества функций-обработчиков. Помимо планировщика, возможна также передача управления между функциями-обработчиками напрямую с использованием явного указания следующего состояния и continue, поэтому данный метод удобен для реализации конечного автомата.
По сравнению с главным циклом, циклический планировщик может уменьшить время реакции за счет подбора различной частоты опроса различных устройств.
Но вместе с тем, подход страдает от тех же недостатков что и подход с главным циклом: требует очень дисциплинированного программирования и документирования. Все функции-обработчики событий должны быть небольшого размера, для достижения приемлемой скорости реакции. Хотя, в отличие от цикла, одна функция-обработчик может быть разбита на несколько, выполнение такой декомпозиции не всегда тривиально. Как и предыдущий случай, не используются прерывания, а значит, при невысокой частоте поступления входных событий, процессор будет расходовать много дополнительной энергии. Имеются проблемы также и с поддержкой: коммуникация между обработчиками выполняется через некоторый глобальных контекст или глобальные переменные, поэтому по мере роста сложности, растет и время поиска ошибок.

3.2.3 Синхронизированный цикл

В случае невысокой частоты поступления событий оба предыдущие подхода страдают от расхода энергии на постоянную прокрутку цикла. Поэтому логичным развитием этих подходов является так называемый синхронизированный цикл.
Суть этого подхода в использовании прерываний для ограничения частоты цикла.

```
do
{
   ...
   delay(100);
   state = schedule();
}
while(1);
```

Реализация функции delay может вычислять значение, на которое необходимо запрограммировать таймер для получения прерывания через заданный промежуток времени, а затем останавливать процессор соответствующей инструкцией. Например, процессоры ARM имеют инструкцию WFI (wait for interrupt), которая переводит процессор в состояние пониженого энергопотребления до получения прерывания.
Из-за того, что часто точное время ожидания неизвестно и требуется остановить процессор до получения любого прерывания, используется модифицированный подход.

Обработчики прерываний сами устанавливают флаги готовности своих устройств (для этого могут использоваться атомарные инструкции), а цикл обработки останавливается до получения прерываний после обработки текущих событий.

```
do
{
  ...
  stop();
  state = schedule();
}
while (1);
```

На первый взгляд кажется, что достаточно чтобы функция stop содержала только инструкцию, аналогичную WFI, но на деле оказывается, что все не так просто. Решение о том, что работы больше нет и необходимо остановить процессор, принимается тогда, когда нет активных запросов от прерываний. Но между анализом состояния и фактическим исполнением инструкции WFI всегда есть некоторый временной зазор.

```
void stop(void)
{
  if(nothing_to_do)
  /*Если прерывание пришло в этот момент, оно будет пропущено*/
    wait_for_interrupts();
}
```

Если прерывание придет в момент между проверкой и остановкой процессора, оно будет пропущено и время реакции на него возрастет до времени прихода следующего прерывания.
Если запретить прерывания, проверить условие, а затем разрешить и остановить процессор, то зазор все равно есть: между разрешением прерывания и командой остановки. Эта проблема не решается без специальной аппаратной поддержки. Эта поддержка заключается в том, что инструкция WFI выводит процессор из состояния ожидания не только тогда, когда прерывания разрешены, но и тогда, когда запрещены [6]. Вызов обработчика при этом откладывается до момента разрешения прерываний. Правильный вариант функции stop, при условии что поддерживается описанная семантика команды WFI.

```
void stop(void)
{
  disable_interrupts();
  if(nothing_to_do)
    wait_for_interrupts();
  enable_interrupts();
}
```

Обработчики прерываний в таком дизайне предельно просты, энергопотребление существенно ниже чем у простого суперцикла. Недостатки все те же - скорость реакции ограничена скоростью работы самой длинной функции-обработчика. Из-за наличия прерываний и необходимости атомарной установки флагов в них вносится элемент неопределенности и определенные трудности в тестировании.

3.2.4 Прерывания

Решения с циклами и разнообразными их модификациями довольно просты, обладают хорошей предсказуемостью и тестируемостью. Многие встроенные решения вполне успешно используют этот подход. Однако главный цикл и его вариации становятся неприемлемым вариантом в случае, если приоритет - время реакции системы. Из-за того, что любая ветвь цикла должна доработать до конца, высокоприоритетные функции не могут начать выполняться сразу же после получения соответствующего сигнала. Если у процессора уже есть механизм вызова обработчиков в ответ на прерывания, почему бы не использовать это в качестве каркаса для постройки встроенных приложений? В этом случае, код обработки события располгается в обработчике соответствующего прерывания, а главная программа может содержать бесконечный пустой цикл [2].

```
void handler1(void)
{
  // обработка события 1
}

void handler2(void)
{
  // обработка события 2
}

int main(void)
{
  do{ wait_for_interrupt(); }  while(1);
}
```

Для того, чтобы этот метод работал, уже потребуется некоторый специфический для процессора слой, который подготавливал бы таблицу прерываний и устанавливал в нее соответствующие функции-обработчики, возможно, снабжая их обертками для корректного завершения прерывания.

Некоторые процессоры, например ARM Cortex-M, позволяют разрабатывать обработчики прерываний на C. В момент, когда возникает прерывание, процессор сохраняет в качестве возвращаемого адреса особое значение, которое не может являться обратным адресом (находящееся на несколько байт ниже границы адресного диапазона, например 0xFFFFFFFD). Если происходит переход на этот "адрес", процессор трактует его особым образом и запускает процедуру выхода из прерывания, поэтому используя стандартные

функции C можно добиться их работы в качестве обработчиков прерываний первого уровня, без каких-либо оберток.

Другие процессоры, например x86, имеют специальную инструкцию IRET, которой должен завершаться обработчик. Компилятор не генерирует эти инструкции в коде возврата из функций, а потому для прерываний всегда необходимы написанные на ассемблере обертки, из которых уже вызываются функции на C.

Если предпринять меры для того, чтобы обработчики были вытесняемыми (реализовать сохранение и восстановление контекста при входе и выходе из каждого обработчика), это позволяет достичь наилучшей скорости реакции: для самого приоритетного обработчика прерывания скорость реакции становится фиксированной).

Хотя при простых процедурах обработки прерывания этот метод может дать хорошие результаты, он слишком тесно интегрирован с аппаратурой. Не все процессоры предоставляют возможности настройки приоритета прерываний, поэтому распределение приоритетов обработчиков может не совпадать с приоритетами реальных задач. Но главная проблема заключается в сложности расширения. При необходимости, например, обеспечить некоторую общую процедуру обработки, скажем, для данных поступающих из разных источников, выясняется что не так просто это реализовать. Другой пример - необходимость по получению данных (наприме, сетевых пакетов) помещать их в очередь, которая будет обрабатываться асинхронно. Кроме того, количество векторов прерываний ограничено, на многих процессорах нельзя программно инициировать аппаратные прерывания, что может привести к необходимости разработки сложных протоколов синхронизации и взаимодействия между обработчиками. При переносе на другую аппаратную платформу может потребоваться существенная модификация.

Подводя итоги, можно сказать, что в случае, если процессор предоставляет возможности по настройке приоритетов прерываний и в самих обработчиках не планируется выполнять длительные работы, то решение с использованием прерываний в качестве "движка" приложения может дать неплохие результаты по времени реакции и производительности, особенно в свете того, что с помощью настройки распределения прерываний по процессорам, можно добиться эффективной работы в том числе и в многопроцессорной системе. Чем более ограниченные аппаратные возможности, а также чем сложнее выполняемая в обработчиках логика, тем этот метод становится все менее привлекательным.

3.3 Использование ОСРВ

Варианты, использующие цикл или прерывания могут быть приемлемым вариантом для несложных приложений. Главный архитектурный недостаток этих решений - смешение управляющей и прикладной логики. С этой позиции ОС является своеобразным фреймворком, который берет на себя абстракцию оборудования и управляющие механизмы.

3.3.1 Акторы

Можно обобщить использование прерываний с помощью реализации подобного функционала программно. Это позволило бы обойти аппаратные ограничения вроде

количества приоритетов, возможностей их настройки, а также позволило бы активировать запросы прерываний программно и позволило реализовать кросс-платформенные интерфейсы.

Предполагается, что аппаратные прерывания активируют некоторый планировщик или диспетчер, который в дальнейшем отдает время одному из программных обработчиков, которые показаны на Рис. 20 как акторы.

Рис. 20 Акторы

В коде это выглядит следующим образом: каждый актор представлен объектом типа actor_t, во время инициализации системы с этими объектами связывается функция-обработчик, приоритет и другие параметры. Обработчик прерывания активирует акторы с помощью предоставляемой специально для этого функции actor_activate. Важным моментом является то, что выполнение актора происходит не синхронно с выполнением обработчика прерывания, функция просто переводит актор в состояние готовности к работе. Далее, после каждого обработчика прерывания должен запускаться планировщик. Его вызов может выполняться как часть инфраструктурной обертки для обработчиков. Планировщик выбирает один из активных акторов и передает ему управление после выхода из всех обработчиков прерываний.

```
static actor_t actor1, actor2;

void handler1(void* arg)
{
  // обработка события 1
}

void interrupt_handler1(void)
{
  // активация обработчика события из обработчика прерывания
  actor_activate(&actor1);
}

int main(void)
{
  // инициализация обработчика
```

```
  actor_init(&actor1, priority, handler1, arg);

  do{ wait_for_interrupt(); }  while(1);
}
```

В отличие от настоящих аппаратных прерываний, обработчики и планировщик реализованы программно, поэтому могут быть реализованы переносимым образом, а также не зависят от аппаратуры и ее возможностей. Возможна гибкая настройка приоритетов и неограниченное количество обработчиков. Кроме того, акторы могут активировать друг друга, что позволяет хорошо декомпозировать функциональность.

Из-за наличия вытеснения обеспечивается хорошее время реакции, а также оптимальное энергопотребление. Некоторые акторы могут быть выделены для выполнения работы, которая может инициироваться из нескольких обработчиков.

Поскольку каждый актор в случае активации начинает работать сначала, у него нету никакого сохраняемого состояния, а поэтому они, как и обработчикив прерываний, могут использовать общий стек [7, 8].

В простейшем случае, обмен данными между акторами возможен через глобальную память, так же, как и в случае циклов. В более продвинутом варианте каждый актор может иметь выделенную для него очередь событий, помещение сообщений в которую происходит как часть процедуры активации. Вся система в крупном масштабе при этом приобретает вид нескольких очередей событий, данные в которые помещаются из прерываний либо акторов, обработчик-актор вызывается на каждое сообщение из очереди до тех пор, пока очередь не опустеет.

К сожалению, для акторов не существует устоявшегося названия, поэтому часто они называются one-shot tasks, run-to-completion tasks, event service routines и так далее.

Подобный же подход применяется как паттерн проектирования при решении некоторых задач даже в настольных системах. Например программирование графического интерфейса (GUI) состоит в написании обработчиков, которые вызываются в ответ на действия пользователя. Инфраструктура, вызывающая сами обработчики при этом скрыта от прикладного программиста.

Модель акторов также является перспективным направлением в исследованиях распределенных систем, в случае коммуникации акторов только через очереди сообщений, можно отказаться от глобальной разделяемой памяти, что открывает путь для разработки систем содержащих сотни и тысячи процессоров, обменивающихся сообщениями.

Основным недостатком, если сравнивать с использованием прерываний, является тот факт, что из-за наличия диспетчера производительность может оказаться существенно ниже. Всякий раз, когда активируется обработчик, его контекст должен быть проинициализирован заново, на что также уходит время.

3.3.2 Потоки

Акторы - достаточно мощный механизм для построения даже сложных встроенных

систем, но есть и альтернативный подход. По ряду причин использование одного разделяемого стека для всех акторов может быть нежелательно. Это затрудняет динамический анализ, сколько стека нужно каждому из них. Оперативная память может быть физически разделена на банки и использование общего стека может быть затруднено. Поэтому дальнейшее естественное усовершенствование заключается в том, чтобы предоставить каждому актору свой собственный стек. Поскольку в этом случае акторы могут работать уже в произвольном порядке, можно также избавиться от накладных расходов на активацию, контекст может сохраняться в стеке и восстанавливаться каждый раз из той же точки, создавая иллюзию, что актор выполняется бесконечно. Рассуждая таким образом, можно прийти к идее потока. **Поток** - последовательность инструкций с которой сопоставлен аппаратный контекст, включающий регистры процессора и стек. Поток можно также определить как виртуальный процессор, который имеет сохраняемый в памяти контекст [3]. Инфраструктура поддержки потоков должна по очереди загружать контексты потоков из памяти в регистры процессора, обеспечивая тем самым иллюзию параллельного исполнения потоков таким образом, как если бы каждый из них выполнялся на выделенном процессоре. В отличие от акторов, для потока создается иллюзия, что он бесконечен, что отражается на принципах прикладного программирования: обычно поток представляет собой бесконечный цикл ожидания и обработки какого-либо события.

```
void thread(void* arg)
{
  while(1)
  {
    wait_for_event(...);
    ... //обработка события
  }
}
```

Также важным моментом является сама концепция явного описания в коде участков, которые допускают параллельное выполнение. Кроме того, в отличие от предыдущих рассмотренных варантов, потоки могут планироваться независимо друг от друга, что дает возможность синхронизации доступа к разделяемым ресурсам путем приостановки потоков. Также они могут распределяться по нескольким процессорам в многопроцессорной системе.

Потоки, хотя и являются в чем-то компромиссным вариантом: более сложны для анализа, медленнее, чем использование прерываний, но вместе с тем, лишены большей части серьезных недостатков. Они реализуемы программно, хорошо изолированы друг от друга в смысле отдельных стеков и контекстов, переносимы, вытесняемы и обеспечивают хорошую скорость реакции, а также предоставляют удобную и независимую от оборудования абстракцию. Кроме того, не так требовательны к питанию как вариации с циклами. Все эти соображения приводят к тому что в настоящее время концепция потока является центральной в абсолютном большинстве ОС, как встроенных так и общего

назначения. Потоки можно рассматривать как универсальную абстракцию, которая допускает реализацию и всех других обсуждавшихся подходов. Например, один из потоков может содержать главный цикл.

Завершая рассмотрение различных механизмов, обеспечивающих выполнение кода, можно ввести следующую классификацию:

Все варианты с циклами можно рассматривать как одну последовательность команд выполянемых на одном стеке (1 задача 1 стек). Использование прерываний и акторов предполагает существование N независимых последовательностей инструкций, которые могут вытеснять друг друга, но выполняются на едином стеке (N задач 1 стек). Наконец, потоки обеспечивают существование нескольких независимых последовательностей инструкций, каждая из которых имеет собственный стек (N задач N стеков).

Сравнение рассмотренных методов приведено в таблице ниже.

	Время реакции	Поддерживаемость	Переносимость	Энерго-эффективность	Простота реализации
Главный цикл	-	-	-	-	+
Цикл. планировщик	-	-	-	-	+
Синхр. цикл	-	-	-	+	+
Прерывания	+	-	-	+	+
Акторы	+	+	+	+	-
Потоки	+	+	+	+	-

3.3.3 Процессы

Все вышеперечисленные способы работают с общей глобальной памятью, поэтому ошибки даже в не очень важных для работы устройства функциях могут привести к аварийному завершению работы всей системы. В случае циклов или прерываний довольно трудно изолировать важные части приложения от не столько важных, потоки же, из-за своей полной изолированности друг от друга за счет отдельного стека, могут быть объединены в группы. Причем с каждой группой могут быть сопоставлены права доступа к памяти и прочие привилеги, что позволит защитить выполнение отних потоков от некорректных действий других. Естественно, для этого необходима аппаратная поддержка, которая обсуждалась в предыдущей главе. Один или более потоков, а также набор сопоставленных с ними ресурсов, таких как адресное пространство, называется процессом. В ранних ОС, таких как первые варианты UNIX, каждый процесс содержал только один поток, такой процесс называется однопоточным. В противоположность этому, современные ОС позволяют создание несклькиx потоков внутри одного процесса, такой процесс называется многопоточным.

Для большинства систем характерно планирование потоков сквозным образом: все потоки от всех процессов планируются в соответствии с приоритетом, процессы приоритета не имеют, а выступают только контейнером общих ресурсов, как показано на Рис. 21, где потоки исполняющиеся в разных процессах показаны разным цветом.

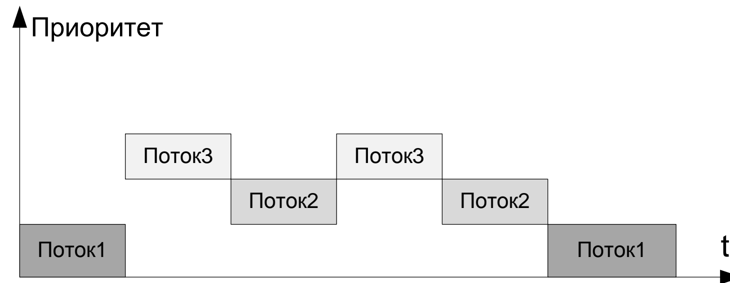

Рис. 21 Сквозное планирование потоков

Альтернативный подход, применяющийся в ответственых системах - циклическое планирование процессов. На этапе проектирования системы составляется расписание, согласно которому определенные процессы получают процессорное время, а потоки планируются уже внутри временного промежутка, предназначенного для их процесса (Рис. 22) [4].

Рис. 22 Циклическое планирование процессов

Изоляция требует применения довольно продвинутых процессоров и применяется только в довольно сложных и функциональных встроенных системах, а также в системах, имеющих повышенные требования к безопасности. Возрастает также сложность прикладного программирования. Из-за отстутвия общей глобальной памяти, процессы не могут обмениваться данными, если ОС не предоставляет для этого соответствующих сервисов, обеспечивающих межпроцессное взаимодействие (inter process communication, IPC).

3.3.4 Задачи ОС

Решения основанные на циклах и прерываниях как правило проектируются и реализуются с нуля при разработке встроенного ПО. Более сложные решения, такие как акторы или потоки, потребовали бы дублирования большого объема кода, при разработке решения каждый раз полностью с нуля. Так как потоки могут быть реализованы переносимым образом для разных процессоров, такие реализации могут распространяться как отдельный универсальный продукт, который можно использовать для решения широкого круга задач. Из описанного становится понятна задача встроенной ОС: предоставление

инфраструктуры для разработки приложений. Эта инфраструктура включает реализацию потоков, планировщика, средств коммуникации между потоками, а также опционально изоляцию групп потоков друг от друга. То есть ОС определяет модель приложения и правила написания последних, позволяя отделить приложение от конкретного аппаратного обеспечения и писать более абстрактный, поддерживаемый и переносимый код.

Поскольку предоставляемые приложениям функции должны быть универсальными, для возможности разработки переносимых приложений, ОС также выполняет функции абстракции аппаратуры и унификации.

Все упомянутые интерфейсы должны быть реализованы таким образом, чтобы допускать функционирование в реальном времени, то есть с использованием детерминированных алгоритмов. ОС, которая удовлетворяет этим требованиям, называется ОСРВ. Термин ОСРВ иногда используется в качестве синонима "быстрой" ОС, что является абсолютно неверным. Основная характеристика ОСРВ - предсказуемость времени реакции. Зачастую, ради достижения предсказуемости, используются специальные алгоритмы, которые, демонстрируя лучшие временные характеристики, имеют, тем не менее, худшую производительность, по сравнению с той, которую можно было бы получить, если бы задача ограничения времени реакции не стояла.

Другой распространенный миф связан с убеждением, что использование в системе реального времени ОСРВ автоматически делает любое прикладное ПО соответствующим критериям работы в реальном времени. Встроенная система состоит из многих компонентов, и предсказуемость ее работы складывается как результат предсказуемой работы всех составляющих, вплоть до аппаратного обеспечения. Реагирует на входные сигналы именно приложение, поэтому, если оно реализовано так, что время реакции не обладает свойством предсказуемости, то ОСРВ спасти ситуацию не сможет. Применение ОСРВ гарантирует лишь то, что ОС работает предсказуемым и детерминированным образом, что дает возможность реализовать приложения, способные работать в реальном времени, а не гарантирует это свойство для любого приложения.

3.3.5 Архитектуры ОС

В большинстве случаев, если при создании устройства используются микроконтроллеры, приложение пользуется сервисами ОСРВ напрямую. То есть ОСРВ представляет собой библиотеку, статически компонуемую с приложением и использующуюся для реализации потоков и прочих сервисов, так называемую libOS (Рис. 23). В этом случае граница между ОС и приложением практически стирается. ОСРВ выступает в роли компонента инкапсулирующего различия между платформами, что позволяет приложению не изменяться при переносе на другие аппаратные платформы. При этом периферийные устройства используются напрямую, то есть без использования ОС, что требует проектирования абстрагирующих устройства слоёв уже на стороне приложения, либо пользоваться сторонними компонентами.

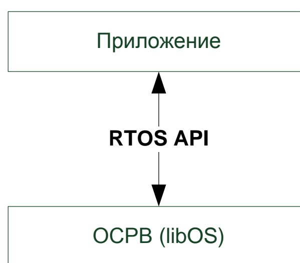

Рис. 23 Архитектура libOS

Как правило, размер библиотеки, реализующей функции ОС в этом варианте исчисляется единицами и десятками килобайт. Подобные ОС часто называются ядрами (RTOS kernels), чтобы подчеркнуть тот факт, что ОС не является полноценной платформой для реализации приложений, а лишь предоставляет минимальный набор абстракций.

Некоторые ОС с такой архитектурой предоставляют базовые сервисы для реализации изоляции. Поскольку контроллеры отличаются очень малым объемом оперативной памяти, который в лучшем случае составляет несколько десятков килобайт, реализация нескольких изолированных процессов сопряжена с большими накладными расходами, поэтому часто процессов всего два: в одном работают защищенные пользовательские потоки имеющие непосредственный доступ к аппаратуре, во втором находится все остальное, например реализации файловых систем или коммуникационных протоколов. Такая архитектура позволяет встроенной системе оставаться работоспособной в случае некорректных действий со стороны тяжеловесных системных сервисов, таких как файловая система, а также повышает устойчивость к атакам извне. Разработка такой системы похожа на разработку системы без защиты - приложение вместе с ядром ОС компонуется в единый образ, который загружается в микроконтроллер. Настройки прав доступа к регионам памяти выполняются уже на этапе старта системы.

При наличии довольно продвинутого аппаратного обеспечения, можно развить эту идею далее и позволить ядру ОС предоставлять системные сервисы, такие как файловая система или TCP/IP всем приложениям. Сами сервисы реализуются в ядре ОС, которая поддерживает несколько изолированных процессов. Последние могут обращаться к ядру ОС за выполнением требуемой работы, например записи или чтения файлов [5].

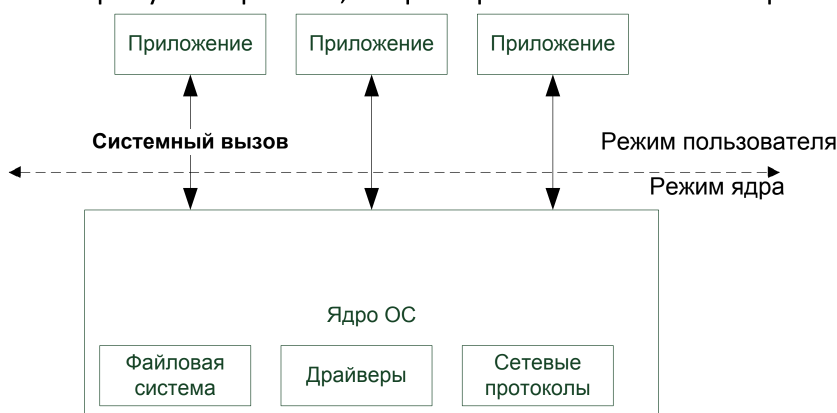

Рис. 24 Монолитная архитектура ядра

Такая архитектура является монолитной (Рис. 24). Несмотря на хорошую производительность, за счет того, что все компоненты ядра могут непосредственно обращаться друг к другу, она страдает от проблем, связанных с надежностью и

стабильностью. Полный набор системных сервисов может быть довольно тяжеловесным, и весь этот функционал работает в привилегированном режиме процессора и имеет доступ без ограничений к любым устрйоствам и к памяти любых пользовательских процессов. При наличии ошибок в реализации довольно объемного ядра страдает стабильность системы в целом.

Попытки добиться большей изоляции частей системы друг от друга для достижения высокой надежности привели к идее микроядра. В этом случае ядро предоставляет только минимальный набор абстракций для создания изолированных процессов, а также механизм IPC. Прикладные сервисы, такие как сеть и файловая система могут быть реализованы как отдельные процессы, которые обращаются к другим процессам через механизм IPC для выполнения той или иной работы (Рис. 25).

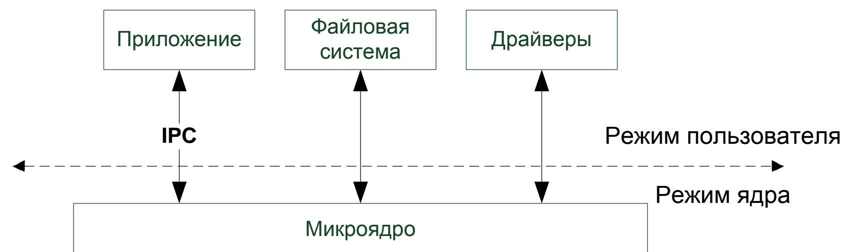

Рис. 25 Микроядерная архитектура

Следует особо подчеркнуть, что понятие микроядро, в своем традиционом значении, никак не связано с размерами кода, выполняющегося в привилегированном режиме процессора. Микроядро прежде всего обозначает именно архитектуру ядра, когда на ядро возлагается, в основном, только поддержка исполняемых сущностей, таких как потоки и организация межпроцессного взаимодействия, позволяющая все остальные части ядра выполнять в виде изолированных процессов. Если же в системе нет процессов-серверов и при своей работе пользовательские потоки не обмениваются сообщениями с другими, то такое ядро не является микроядром в классическом понимании последнего.

Наконец, существуют также различные промежуточные варианты. В качестве примера можно привети гибридные ядра, которые находятся между монолитными и микроядрами. В этом случае критичные к производительности сервисы реализаются внутри ядра, как в случае монолитных систем, а менее требовательные или те, к которым предъявляются требования повышенной надежности - как процессы-серверы.

Другим примером является экзоядро. В случае экзоядра, ядро системы, выполняющееся в привилегированном режиме процессора, отвечает только за выделение аппаратных ресурсов, таких как области памяти или время процессора. Все остальные сервисы ядра реализуются как libOS на стороне пользователя, внутри пользовательского процесса.

Одна система может сочетать признаки всех типов, поэтому разделение по типам не является строгим.

3.4 Резюме

Существует много подходов к реализации встроенного ПО. Для простых устройств целесообразно применение того или иного варианта цикла, которые обеспечивают хороший компромисс между простотой реализации приложения и временем реакции. В более сложных системах оправдано применение более продвинутых техник, наиболее популярной из которых является использование потоков. Несмотря на некоторую сложность реализации, они обеспечивают хорошее время реакции, энергоэффективность, а также возможности расширяемости и поддержки приложений, что позволяет сокращать время выхода на рынок устройств. Особенно это актуально в случае поддержки линейки устройств имеющих пересекающиеся функции, когда важно оебспечить повторное использование кода.

Для реализации потоков и прочей инфраструктуры часто используются ОСРВ в виде внешних компонентов. В большинстве случаев достаточно функционала, предоставляемого ОСРВ имеющих минимальный функционал и архитектуру libOS. При наличии повышенных требований к надежности и безопасности может использоваться изоляция и процессы. Монолитная архитектура обеспечивает хорошую производительность, но обладает в целом невысокой надежностью из-за большого количества кода, который имеет привилегии ядра. Микроядро - попытка добиться высокой степени изоляции не только пользовательских приложений, но и системных сервисов. К сожалению, механизмы межпроцессного взаимодействия требуют существенных накладных расходов, поэтому несмотря на высокую степень надежности, такая архитектура отличается невысокой производительностью.

[1] Philip Laplante; "Real-time systems design and analysis." Third edition 2004
[2] Jim Cooling; "Real time Operating systems"
[3] Anderson, Dahlin; "Operating systems. Principles and practice" 2011
[4] Wind River VxWorks 653 Platform 2.3 whitepaper
[5] Andrew Tanenbaum, "Modern Operating systems" 2nd edition. Chapter 1.
[6] ARMv8-M Reference Manual B2.1.5 p.50 2015
[7] Baker, "A Stack-Based Resource Allocation Policy for Realtime Processes", 1991
[8] Karsten Walther et al. "Using Preemption in Event Driven Systems with a Single Stack"

4 Абстракция оборудования и переносимость

Темы главы:

- Подходы к разработке кроссплатформенного встроенного ПО
- Использование специальных команд процессора в языке С
- Инициализация после запуска
- Абстракция программных и аппаратных прерываний
- Переносимая реализация переключения контекста
- Интерфейсы для взаимодействия процессоров в многопроцессорной системе
- Использование механизмов разделения привилегий и защиты памяти

4.1 Проблема переносимости

Встроенные системы работают на большом разнообразии устройств. Из-за того, что проектирование, разработка и тестирование ПО являются довольно затратными по времени процессами, хотелось бы разрабатывать ПО без привязки к конкретной аппаратной платформе. К тому же, довольно часто разрабатываемое ПО должно использоваться одновременно на нескольких различных устройствах, например, в случае разаботки линейки продуктов, которые имеют некоторый общий функционал, который реализуется программно, на различной аппаратуре. Кроме того, по мере развития проекта также может потребоваться переход на другую аппаратную платформу. Поэтому, с точки зрения снижения рисков и сокращения выхода продукции на рынок, нужно иметь возможность разрабатывать ПО один раз, а затем переносить его на другие аппаратные платформы. Подобные рассуждения приводят к тому, что желательным свойством ПО является **переносимость**. Переносимость - свойство ПО, позволяющее ему работать в другом аппаратном или программном окружении без изменений. Это достигается с помощью повышения уровня абстракции - ПО разрабатывается на основе универсальных интерфейсов, которые скрывают различия платформ. Разработка переносимого прикладного кода требует усилий от программиста в том смысле, что он должен придерживаться этого интерфейса. Если была использована специфика какого-то конкретного процессора, переносимость теряется, хотя код формально остается корректным.

Хотя под переносимостью может подразумеваться не только переносимость между аппаратными платформами, но и, например, приложений между разными ОС, в данной главе будет рассматриваться только аппаратная переносимость, так как именно этот вид переносимости наиболее важен при разработке ОС.

Несмотря на то, что все процессоры похожи между собой в том смысле, что реализуют принцип последовательного исполнения кода и имеют сходные наборы инструкций, они не являются совместимыми. Бинарный код одного процессора в общем случае нельзя запустить на другом. Ассемблер также не является переносимым, так как каждый процессор имеет свой собственный набор инструкций. Аналогичная ситуация и с периферийными устройствами: хотя все устройства одного типа сходны между собой по функциям, например таймеры или контроллеры прерываний, они имеют различный набор регистров и форматов этих регистров, что затрудняет использование кода

написанного для одного устройства с другим устройством.
Более или менее сложные встроенные системы требуют довольно объемной инфраструктуры, например поддержки потоков или акторов, которые требуют знания подробностей об используемом аппаратном обеспечении для выполнения, например такого действия как переключение контекста. Поэтому одной из функций ОС является абстракция оборудования, которая позволяет разрабатывать переносимые или кроссплатформенные приложения.

4.2 Универсальный слой абстракции

Ранние компьютерные системы программировались в машинных кодах или на языке ассемблера, поэтому, поскольку последний меняется от процессора к процессору, для каждой платформы приходилось разрабатывать код с нуля. С распространением языков высокого уровня, таких как С, появилась возможность абстрагироваться от конкретных процессоров и поручить перевод высокоуровневой программы на машинный язык или ассемблер самому компьютеру с помощью компиляторов.
Если с прикладными программами все получилось относительно неплохо, то с системным ПО возник ряд проблем. В отличие от более или менее общей концепции исполнения прикладного кода, процессоры довольно сильно различаются в обработке прерываний, исключений и тому подобного. Более того, обработка этих событий отличается в зависимости от окружения, в котором работает программа, например в случае использования поддержки различных уровней привилегий или MMU, поэтому потребовалось бы отдельное портирование не только на процессорную архитектуру, но и на режимы работы процессора, что трудно назвать эффективным. Если в язык высокого уровня ввести все эти объекты, а также специфичные для каждого процессора функции для работы с ними, то получится, что даже написанный на языке высокого уровня код получается непереносимым. Кроме того, это существенно усложнило бы компилятор, притом что в подавляющем большинстве случаев эти возможности не использовались бы, потому что основная доля разрабатываемого ПО носит все-таки прикладной характер и не занимается обработкой прерываний.
Хотя некоторые компиляторы содержали необходимые расширения, например Borland Turbo C содержал специальные ключевые слова позволявшие писать обработчики прерываний на С для распространенных на тот момент процессоров x86, а компилятор генерировал специфический код такой функции [1], в целом, по вышеуказанным причинам, такой подход не прижился. Для таких случаев в языке обычно предусмотрены возможности для взаимодействия с кодом, написанным на ассемблере. Хотя этот подход решает проблему использования специфических возможностей процессора, он не решает проблему переносимости, поскольку каждый процессор реализует свой собственный аппаратный интерфейс который не имеет выражения в самом высокоуровневом языке.
С другой стороны, очевидно, если бы в мире существовал только один процессор, то проблемы переносимости не существовало бы. Поэтому, идея абстракции оборудования заключается в том, чтобы разработать высокоуровневый (выраженный на языке высокого уровня) интерфейс некоторого абстрактного процессора, таким образом, чтобы для всех платформ, которые планируется поддерживать, этот интерфейс можно было бы

реализовать (с привлечением компонентов, написанных на ассемблере). Тогда все прикладное и системное ПО можно было бы писать для этой абстрактной машины, что позволило бы использовать его на всех платформах, для которого существует реализация этого универсального интерфейса (Рис. 26). Совокупность всех компонентов которые реализуют этот интерфейс называются уровнем или **слоем абстракции оборудования** (hardware abstraction layer, HAL).

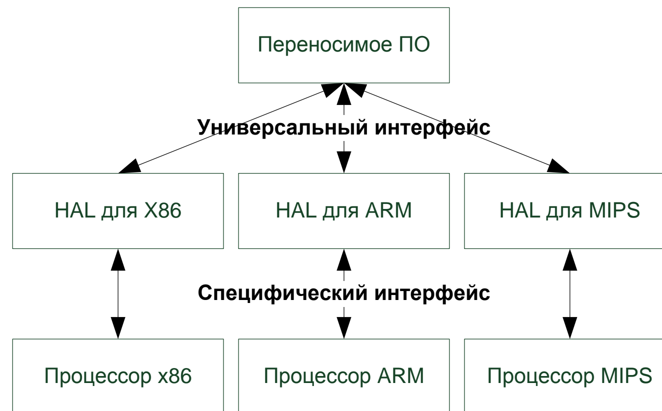

Рис. 26 Слой абстракции оборудования

Традиционно для разработки встроенного ПО используется язык С, поэтому примеры, рассматриваемые в данной главе будут приводиться с использованием этого языка. Он обладает хорошей производительностью, а также не содержит встроенных в язык возможностей вроде сборки мусора, которые затрудняют разработку ПО реального времени. Кроме того, С предоставляет удобные возможности для использования ассемблера.

4.2.1 Структура HAL

Традиционный способ реализации HAL для ОС: определение платформенно-зависимых функций и механизмов конкретного ядра с последующей их реализацией для каждой поддерживаемой платформы. Этот метод эффективен только в том случае, если функционал ядра остается относительно постоянным. Если функциональность ядра может изменяться, например, в результате конфигурирования, то использование во всех случаях полного модуля HAL с максимальной функциональностью сопряжено с накладными расходами. Как уже было показано, в области встроенных систем существует большое разнообразие подходов к тому, как вообще может разрабатываться приложение, хотелось бы решить проблему переносимости в более общем виде, так, чтобы все описаные подходы могли быть использованы с использованием одного слоя абстракции оборудования. Это позволяет сделать HAL самостоятельной сущностью, разрабатываемой отдельно от конкретного ядра ОС, что, помимо прочего, позволит организовать его независимое тестирование. Поэтому HAL можно также определить как набор программных компонентов, реализация которых специфична для аппаратной платформы, притом что в конкретных применениях набор этих компонентом может отличаться.

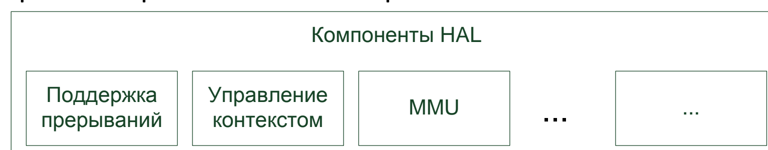

Рис. 27 Компоненты HAL

Под компонентом в данном случае подразумевается независимо компилируемая и используемая сущность, обладающая интерфейсом. В языке С такой сущностью является пара из заголовочного файла описывающего интерфейс и реализации интерфейса в одном или более файлах исходных текстов.

Этот набор не является фиксированным, в зависимости от функционала ОС, а также от требований приложений, какие-то компоненты могут добавляться или удаляться. Например, работа с MMU специфична для процессора, формат записей каталогов и таблиц страниц отличается от процессора к процессору, однако многие встроенные приложения не пользуются MMU и поэтому всё, связанное с этим устройством, желательно удалить из HAL. Другой пример: многие процессоры предоставляют возможности управления кэшированием, для большинства прикладных и даже системных программ доступ к этим возможностям не нужен, в то же время при разработке чувствительных к производительности программных продуктов может быть оправдано проектирование и реализация кроссплатформенного интерфейса управления кэшированием и тому подобное.

Проектирование HAL сложно тем, что требуется, опираясь на существующее аппаратное обеспечение, соответствовать в том числе и архитектурам, которые появятся в будущем. Поэтому, при проектировании следует придерживаться принципа сокрытия нежелательных свойств какого-либо функционала, стараясь при этом не скрыть за абстракциями полезные свойства, которые могут появиться. Иными словами, излишние обобщения редко идут на пользу интерфейсу.

Если какие-то возможности не поддерживаются аппаратно, HAL должен реализовать их программно. Поскольку HAL абстрагирует конкретное аппаратное окружение, из этого, в частности, следует то, что для одного и того же процессора может существовать несколько вариантов HAL. Например, если для достижения максимальной производительности не используются возможности защиты памяти и поддержки процессов, то поддержка этих возможностей потребует разработки отдельного HAL, потому что наличие разделения привилегий влияет практически на все низкоуровневые аспекты работы процессора, от инициализации до обработки прерываний, как будет показано далее.

HAL может содержать множество компонентов, каждый из которых посвящен какой-то аппаратной функции, имеющей высокоуровневое представление. Рассмотреть все из них вряд ли возможно, поэтому рассматриваемый набор будет ограничен компонентами, которых достаточно для реализации функционала, описываемого в следующих главах, то есть ядра ОСРВ с поддержкой многопроцессорности, прерываний, исключений, защиты памяти и разделения режимов процессора. Будут рассмотрены следующие темы:

- Переносимая реализация специфических команд процессора, таких как атомарные инструкции
- Инициализация системы
- Абстракция прерываний и механизмов управления ими
- Обработка исключений

- Поддержка непривилегированных приложений и режимов работы процессора
- Защита памяти с помощью MPU и MMU
- Поддержка многопроцессорных систем и соответствующей низкоуровневой синхронизации

Некоторые компоненты рассматриваются несколько раз в различных контекстах. Например, инициализация HAL с поддержкой MMU несколько отличается от инициализации HAL для микроконтроллеров без защиты памяти.

В качестве иллюстрации будут использоваться компоненты, входящие в HAL FX-RTOS для процессоров MIPS M4, ARM Cortex M, ARM Cortex A, Intel x86. Хотя, по возможности, используется язык C, при разработке HAL сложно обойтись без ассемблера. Ассемблерный код для каждого процессора снабжается подробными комментариями, но для детального понимания рекомендуется обратиться к описанию системы команд каждого из процессоров, ссылки на соответствующие документы можно найти в списке литературы [2], [3], [4], [5].

4.2.2 Специальные команды процессора

В связи с невозможностью выразить специфические команды процессора в языке C один из первых вопросов, которые возникают при разработке кроссплатформенных компонентов: каким образом использовать атомарные операции и команды для доступа к устройствам. Многие компиляторы поддерживают высокоуровневый интерфейс для подобных инструкций (называемых intrinsics), а также ассемблерные вставки, которые можно использовать внутри функций на C. Однако оба этих способа плохо подходят для разработки кроссплатформенного ПО, потому что сильно зависят от компилятора и его возможностей. К тому же, само использование ассемблера делает код непригодным для запуска на другой архитектуре, поэтому наиболее разумным подходом является использование внешних ассемблерных файлов в которых были бы реализованы требуемые функции.

В качестве примера можно рассмотреть реализацию атомарных инструкций и функций для управления прерываниями. Предполагается, что каждый процессор должен содержать команду сравнения с обменом CAS (или способ ее реализовать), сложения с обменом, и атомарный обмен между ячейками памяти. Для управления прерываниями используется две функции, разрешение и запрет прерываний.

```
unsigned int atomic_swap(volatile unsigned int* p, unsigned int newval);
unsigned int atomic_add(volatile unsigned int* p, unsigned int addend);
unsigned int atomic_cas(volatile unsigned int* p, unsigned int old, unsigned int new);

void intr_enable(void);
void intr_disable(void);
```

С атомарными инструкциями связан один тонкий момент, касающийся типа значения. У разных процессоров разрядность может различаться, поэтому никакой из стандартных размерных типов не подходит на роль универсального переносимого типа. Один из вариантов - ввести отдельный тип для атомарных значений, другой подход заключается в том чтобы положиться на компилятор и предположить, что процессор поддерживает

атомарный доступ к памяти для стандартного беззнакового типа, размер которого меняется от процессора к процессору. Для всех рассматриваемых архитектур это правило выполняется, поэтому предоложение можно считать рабочим. Еще один неочевидный нюанс: пригодность этих функций для работы с указателями. Необходимость атомарного доступа к указателям требуется едва ли не чаще, чем атомарный доступ к целочисленным переменным. Для большинства рассматриваемых процессоров размерность указателя равна размерности целочисленного типа данных (32 бита), поэтому эти же функции могут быть использованы и для указателей. В то же время, существуют архитектуры, например intel x64, в которых размерность данных по-умолчанию (типа unsigned int) отличается от размерности указателя: типы данных, как и в 32-битных архитектурах, остаются 32-битными, а указатели имеют вдвое больший размер. Для корректной работы может потребоваться дополнительный набор функций, работающих с указателями: atomic_cas_ptr, atomic_swap_ptr и т.д.

Пример реализации CAS для архитектуры x86, где такая команда поддерживается процессором:

```
atomic_cas:
  movl    4(%esp),  %ecx
  movl    8(%esp),  %eax
  movl    12(%esp), %edx
  lock cmpxchg %edx, (%ecx)
  ret
```

Первые три команды читают аргументы, которые вызывающий код помещает на стек. Правила, согласно которым происходит передача аргументов между функцями называется **конвенцией вызовов** (call convention) и определяется компилятором для данной процессорной архитектуры. В случае x86 аргументы передаются через стек.

После чтения аргументов вызывается команда CAS, которая в архитектуре intel имеет название cmpxchg и получает свои аргументы в регистрах. Возвращаемый результат после выполнения команды находится в регистре eax, который определен конвенцией как место возврата результата из функций, то есть никаких действий по возврату аргумента не требуется и можно сразу возвращать управление вызывающей функции с помощью команды ret.

Дл архитектур, содержащих вместо CAS пару LL/SC реализация функции более сложна и содержит цикл.

```
atomic_cas:           ; Согласно конвенции вызовов указатель находится в R0,
                      ; компаранд в R1, новое значение в R2
  ...
retry:
  dmb                 ; Барьер для завершения всех предыдущих операций с памятью
  ldrex r3, [r0]      ; Чтение значения в r3 и мониторинг ячейки памяти
  teq r3, r1          ; Сравнение прочитанного значения с ожидаемым
  strexeq r4,r2,[r0]  ; Попытка замены значения при успешном сравнении (результат в R4)
  movne r4, #0        ; Если прочитанное значение не совпадает с ожидаемым оно
  movne r1, r3        ; возвращается как результат
  teq r4, #0          ; Проверка успешности STREX
  moveq r0, r1        ; Возврат ожидаемого значения (это означает успешность CAS)
```

```
  bne retry          ; STREX закончился неудачей - переход в начало
  dmb
  ...
  bx lr
```

В отличие от x86, в ARM конвенция вызовов предполагает передачу аргументов через регистры, поэтому в момент вызова функции с вышеописанной сигнатурой, R0 будет содержать указатель на атомарную переменную, R1 - ожидаемое значение переменной (компаранд), а R2 - новое значение, которое будет записано в перменную если ее текущее значение равно ожидаемому.

После пролога функции, сохранения регистров, которые не должны меняться при вызове, а также барьера DMB, начинается основной цикл. Командой ldrex (аналог LL) читается текущее значение по указателю. Поскольку возвращаемое значение используется в качестве индикатора успеха CAS, в том случае, если команда сохранения была неуспешна, но текущее значение переменной совпадает с ожидаемым, необходимо повторить цикл чтобы избавиться от ложной трактовки возврата старого значения как успешного завершения CAS. Для этого командой TEQ сравнивается прочитанное значение и компаранд. Далее, в том случае, если они равны, происходит попытка выполнить сохранение командой strex (аналог SC). Суффикс eq говорит о том, что команда будет выполнена только в том случае, если TEQ был успешен (прочитанное значение и компаранд были равны). Результат выполнения strex помещается в регистр R4. В том случае, если компаранд и прочитанное ldrex значения были не равны, выполнение функции необходимо прервать. Для этого выполняются две следующие команды mov, первая из них устанавливает R4 в значение 0, как если бы strex завершился успешно, а вторая устанавливает прочитанное из памяти значение в качестве возвращаемого. Далее происходит проверка регистра R4. Единственная возможность, когда это сравнение будет неудачно - если strex выполнился и завершился неудачей. В этом случае выполняется команда перехода и происходит следующая итерация цикла. В противном случае, если strex завершился успешно, либо если выполнение функции прервано из-за несовпадения прочитанного и ожидаемого значения, прочитанное значение устанавливается в качестве результата функции, после чего происходит возврат.

Большинство функций состоят и вовсе из 1-2 ассемблерных команд, например упомянутые функции управления прерываниями для ARM и x86 будут использовать соответствующие команды, CPSID I и CLI соответственно.

```
intr_disable:
     cli
     ret

intr_disable:
     cpsid i
     bx lr
```

Подобным образом указанный интерфейс может быть реализован для любого процессора.

В завершение этой темы можно сказать, что интерфейс устройств и процессора может

содержать функции двух типов: интерфейс предназначенный для использования приложениями и интерфейс предназначенный для использования в пределах одной платформы. Например, предполагается что любой контроллер прерываний имеет возможность запрещать определенные источники (векторы). Эти функции должны иметь специальное название и поддерживаться во всех платформах для всех процессоров. С другой стороны, конкретный контроллер прерываний может иметь специфические дополнительные функции, которые не могут использоваться приложениями, но могут использоваться HAL, которые предполагают использование именно такого контроллера прерываний.

4.3 Инициализация

4.3.1 Системы с одним процессором и микроконтроллеры

Встроенные системы используют два основных типа инициализации: с использованием первичного загрузчика и без такового. В первом случае используется отдельный загрузчик, который выполняет инициализацию оборудования, а затем передает управление приложению или ОС, образ которых считывается либо из встроенной энергонезависимой памяти, либо из внешних носителей. Системы с ограниченными ресурсами как правило не пользуются загрузчиками, в этом случае и код инициализации оборудования и ОС и приложение находятся в едином бинарном образе.

После включения питания, типичный микроконтроллер начинает выполнять программу с определенного адреса из энергонезависимой памяти. Этот код должен выполнить первичную инициализацию: настройку частот, параметров питания для периферии и так далее. Все эти действия сильно специфичны не только для определенного типа процессора или производителя, но также и для конкретной модели микроконтроллера. Производителем обычно поставляются библиотеки, которые содержат необходимые функции инициализации, а также код, обеспечивающий инициализацию образа приложения и библиотеки поддержки языка и запуск функции main.

С другой стороны, использование ОС или какой-либо иной инфраструктуры обычно предполагает выполнение трех стадий инициализации:

- Инициализацию инфраструктуры
- Инициализацию пользовательского приложения, под чем подразумевается создание пользовательских потоков и прочих прикладных сущностей
- Запуск ОС, которая передает управление пользовательскому приложению в соответствии с используемой политикой планирования

Самый простой подход - последовательно вызывать функции, предназначенные для той или иной стадии инициализации.

```
void main(void)
{
  hardware_init();     // Поставляется производителем устройства
  os_init();           // Поставляется производителем ОСРВ
  application_init();  // Разрабатывается пользователем
  os_start();          // Поставляется производителем ОСРВ
}
```

Данный подход, несмотря на свою простоту, обладает также некоторыми недостатками. В зависимости от используемого процессора, начальный стек, на котором работает функция main, может потребовать изменения. Например, процессоры ARM Cortex-M по умолчанию запускаются с использованием стека, который используется для прерываний. Процессор же поддерживает аппаратно еще один стек, который должен использоваться для многопоточных приложений, поэтому предполагается, что в процессе инициализации стек будет переключен.

Переключение стека нежелательно в пределах одной С-функции, так как может привести к неопределенному поведению из-за наличия в стеке временных переменных, поэтому процедуру инициализации можно немного модифицировать: после выполнения аппаратной инициализации, просиходит вызов процедуры запуска ОС, которая при необходимости устанавливает правильный стек, а затем выполняет инициализацию ОС с последующим вызовом пользовательского приложения.

```
void main(void)
{
  hardware_init();
  kernel_entry();
}

void kernel_entry(void)
{
  os_init();
  app_init();
  os_start();
}
```

При необходимости, функция kernel_entry реализуется на ассемблере и производит все необходимые низкоуровневые манипуляции с оборудованием и стеками. Таким образом, из пользовательского приложения удаляется знание о платформе, в момент вызова функции app_init система всегда находится в корректном состоянии и известном окружении (ОС и оборудование инициализированы и готовы к работе).

При использовании варианта с загрузчиком общая структура инциализации остается такой же, точка входа в приложение устанавливается на функцию main, загрузчик передает в нее управление и далее все происходит так же как и в случае, описанном выше.

Хотя реализация может показаться простой и даже тривиальной, при использовании многопроцессорных систем или систем с разделением привилегий модуль инициализации существенно усложняется.

4.3.2 Инициализация многопроцессорной системы

При переходе к многопроцессорному случаю, инициализация должна гарантировать то же самое - в момент вызова пользовательской функции ОС и аппаратура проинициализированы и готовы к работе. Под готовностью аппаратуры подразумевается в том числе и готовность всех процессоров, поэтому инициализация и запуск такой системы полностью возлагается на HAL. Очевидно, размножение рассмотренного выше кода на все процессоры не приведет к желаемому эффекту, так как пользователь ожидает

однократного вызова функции инициализации приложения. Кроме того, многие компоненты ОС, работающие с аппаратными устройствами также должны быть вызваны однократно, потому что устройство может присутствовать в единственном экземпляре. Поэтому функция инициализации ОС разделяется на две: os_init_once - вызываемая однократно, и os_init_each, вызываемая на каждом из процессоров.

В отличие от однопроцессорного случая, возможны различные сценарии запуска многопроцессорной системы. Поскольку сразу после запуска состояние оперативной памяти может быть не определено, невозможно как-то скоординировать действия нескольких процессоров, если они все запустятся одновременно, поэтому в типичном мультипроцессоре первым стартует один из процессоров, называемый главным (primary processor), который, после проведения первичной инициализации, должен запустить остальные процессоры, называемые второстепенными (secondary processors).

Второстепенные ядра после завершения своей части иницилизации сигнализируют главному процессору и переходят в режим ожидания. Главный процессор дождавшись завершения иницилизации всех остальных процессоров вызывает приложение. К моменту вызова приложения система полностью инициализирована, а все второстепенные процессоры находятся в режиме ожидания (Рис. 28).

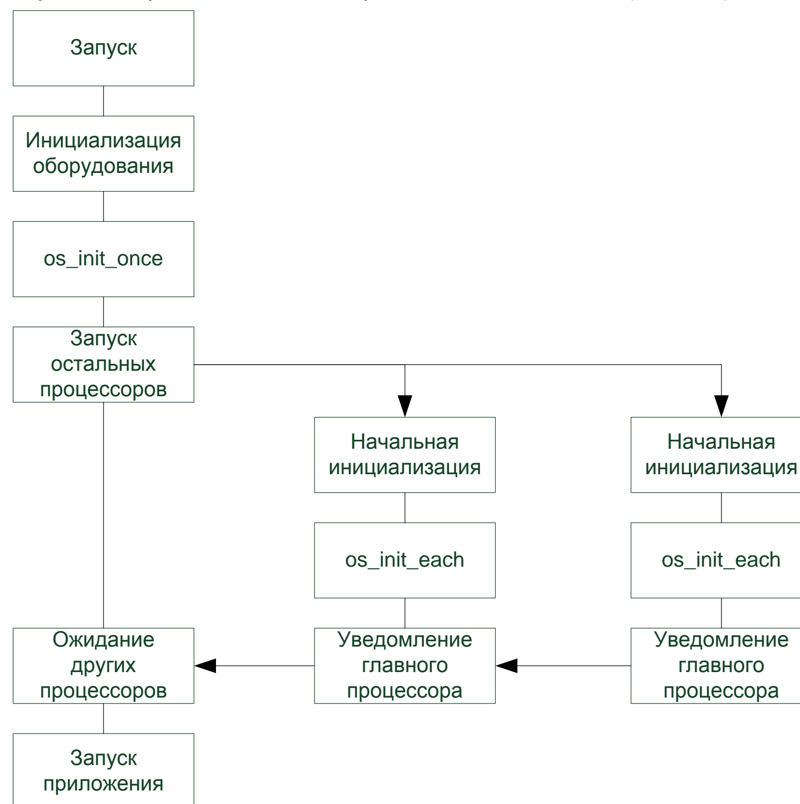

Рис. 28 Инициализация многопроцессорной системы

Также могут использоваться различные вариации этой схемы [6]. С точки зрения пользовательского приложения интерфейс остается прежним.

4.4 Модель прерываний

Реализация высокоуровневного интерфейса для специфических инструкций процессора предполагает вызовы функций, реализованных на ассемблере, из высокоуровневого кода. Задача обработки прерываний является примером обратного взаимодействия: из

низкоуровневного, специфического для процессора кода, требуется вызывать высокоуровневый обработчик. Применяемые при этом подходы схожи.

4.4.1 Аппаратные прерывания

Эффективная работа с прерываниями - краеугольный камень систем реального времени. Для того, чтобы скорость реакции не страдала от накладных расходов, абстракция прерываний должна быть реализована настолько эффективно, насколько это возможно. До сих пор рассматривались только синхронные функции. Прерывания существенно отличаются от них, так как происходят независимо от действий программы. Если операционная система поддерживает потоки, то, так как прерывания могут происходить во время выполнения любого потока, его стек должен быть достаточного размера для обработки всех возможных прерываний включая вложенные. Это ведет к большому перерасходу памяти, в случае большого количества потоков. Поэтому, в большинстве операционных систем, включая ОС общего назначения, используется выделенный стек прерываний. Поскольку это решение оптимально независимо от используемого процессора, использование отдельного стека для обработчиков прерываний можно включить в описание интерфейса прерываний HAL. Некоторые процессоры, например ARM Cortex-M аппаратно поддерживают два стека, один из которых используется для прерываний, и, таким образом, требуемое поведение реализуется аппаратно. В других процессорах переключение стека должно быть реализовано программно.

В большинстве процессоров обработчик прерывания требует использования специальных инструкций для возврата в прерванную программу. Подход с использованием написанной на ассемблере специальной функции, которая содержала бы нужную команду, в данном случае не работает. Дело в том, что данная команда использует скрытое состояние, то, что было помещено процессором на стек в момент передачи управления обработчику. Компилятор может очищать временные переменные, расположенные на стеке, после выполнения всего содержимого функции, а значит, никогда нельзя гарантировать, что вершина стека содержит данные именно в том формате, который нужен инструкции возврата. По этой причине, первичный обработчик, которому передает управление процессор, обычно реализуется на ассемблере и является специфичным для процессора. Этот обработчик используется для выполнения всех необходимых действий при входе в обработчик, таких, как переключение стека, если оно не поддерживается аппаратно, а затем вызывается высокоуровневый обработчик (называемый также вторичным), написанный на С. Этот обработчик использует стандартную конвенцию вызова и возвращается в первичный обработчик так же, как и любая другая функция. Далее первичный обработчик использует специальные инструкции для завершения прерывания. Поскольку первичный обработчик реализован на ассемблере, он сохраняет полный контроль над использованием стека, поэтому описанная выше проблема не возникает.

Процессор находит первичные обработчики с помощью таблицы прерываний, поэтому другой важный вопрос заключается в том, кто и как заполняет эту таблицу. Поскольку структура таблицы специфична для процессора, было бы логично расположить ее как внутреннюю структуру компонента HAL, посвященного прерываниям, однако такой подход не всегда оправдан. Дело в том, что многие процессоры предъявляют требования

к расположению таблицы прерываний в памяти, а ОС, если она реализована в виде библиотеки, не может влиять на размещение в памяти своих структур данных, поскольку эта процедура выполняется как часть компоновки при создании бинарного образа. Чтобы не усложнять использование ОС, поставляя вместе с ней файлы настройки компоновщика, таблица обычно размещается на стороне пользовательского приложения и ее размещением в памяти управляет также пользователь. HAL, в свою очередь, предоставляет первичные обработчики прерываний, которые должны быть установлены пользователем в таблицу (Рис. 29).

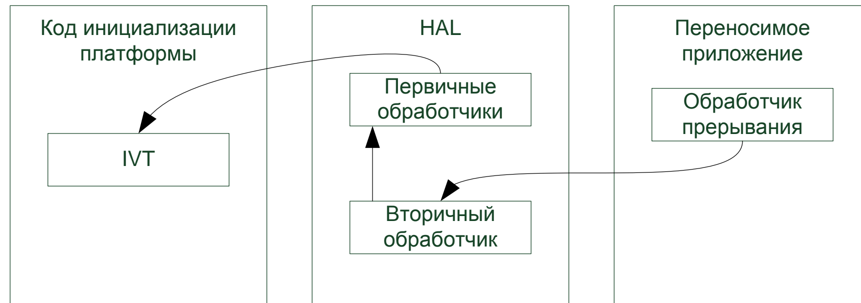

Рис. 29 Первичные и вторичные обработчики

Это дает также дополнительную гибкость в том смысле, что пользователь может и не пользоваться сервисами ОС для некоторых разработчиков, становив в них свои собственные первичные обработчики определенные приложением. Кроме того, некоторые процессоры могут содержать более тысячи векторов прерываний, поскольку HAL не может знать, сколько именно векторов используется в данном устройстве, в случае размещения таблицы в HAL потребовалось бы либо вводить конфигурационные опции для задания ее размера, что усложнило бы использование ОС, либо заполнять ее всю, что привело бы к неоправданным расходам памяти на таблицу.

Переходя к проектированию интерфейса, можно определить набор функций первичных обработчиков, которые предназначены для установки пользователем в таблицу прерываний. Поскольку таблица прерываний специфична для процессора, набор этих функций также специфичен. Например, если процессор использует всего один вектор для аппаратных прерываний (ARMv7A), либо имеется возможность узнать свой текущий вектор уже находясь внутри обработчика, функция также одна. Если же процессор предполагает множество векторов прерываний и отсутствует возможность узнать вектор, кроме как с помощью вызова уникального обработчика для каждого из них, имеет смысл использовать набор из функций.

Определяется также внешняя функция, которую должен предоставить пользователь (interrupt_handler). Эта функция вызывается в ответ на аппаратные прерывания, причем независимо для используемого процессора для нее гарантируется выполнение на отдельном стеке, а также возможность ее написания на высокоуровневом языке без необходимости использования специфических инструкций.

Во время выполнения обработчика прерывания разрешены, но все векторы с приоритетом менее текущего замаскированы.

Для некоторых процессоров, например для Cortex-M, реализация модуля прерываний тривиальна, потому что описанное поведение по большей части реализуется аппаратно.

```
intr_entry:
  push  {lr}
  ...              ; сохранение не сохраняемой автоматически части контекста
  bl interrupt_handler
  ...              ; восстановление сохраненной части контекста
  pop   {pc}
```

При выполнении приложения основным стеком является так называемый процессный стек (process stack pointer, PSP). При получении прерывания, процессор частично сохраняет контекст в процессный стек, переключается на стек прерываний и начинает выполнение обработчика на новом стеке. При этом автоматически отслеживается вложение, если во время выполнения обработчика возникло еще одно прерывание, стек не будет переключен так как процессор уже использует стек прерываний. Более того, из-за поддержки процессором механизма выхода из прерывания без специальных инструкций, даже первичный обработчик может быть написан на C. Функция получения вектора также реализуется процессором аппаратно. Процессор содержит регистр IPSR, содержимое которого соответствует текущему вектору прерывания.

Когда процессор поддерживает все нужные функции аппаратно, реализовывать HAL довольно просто. Можно рассмотреть не столь тривиальный вариант: процессоры MIPS. В этом случае процессор не имеет регистра аналогичного IPSR, а также не переключает стек автоматически, для реализации описанного интерфейса все эти действия приходится реализовывать программно. Различать обработчики по векторам можно только с помощью установки отдельного обработчика для каждого вектора, поэтому вместо одной функции, как в случае ARM, используется набор из функций-заглушек. Каждая из них помещает свой вектор в регистр k0 и выполняет переход на общую часть первичного обработчика.

```
intr_entry_N:
    addiu k0, zero, <N>
    j common_interrupt_handler
    nop
```

Регистры k0 и k1 выделены для использования операционной системой, поэтому их содержимое не нужно сохранять, они не являются частью контекста прикладной программы. Общая часть обработчика выглядит следующим образом:

```
common_interrupt_handler:
   1. addiu       sp, sp, -HAL_CONTEXT_SZ    ; Отведение места под контекст в стеке
   2. ...                                    ; Сохранение контекста в стек
   3. addu        a0, zero, k0               ; Сохранение текущего вектора в A0
   4. la          t0, intr_nesting           ; Загрузка адреса переменной в T0
   5. lw          t1, (t0)                   ; Чтение счетчика вложения прерываний
   6. addiu       t2, t1, 1                  ; Увеличение счетчика
   7. sw          t2, (t0)                   ; Сохранение результата
   8. bne         t1, zero, skip_stack_switch; Пропустить переключение стека если >0
   9. nop
   10. addu       s8, zero, sp               ; Сохранить текущий указатель стека в S8
   11. la         sp, intr_stack             ; Переключение на стек прерываний
   12. addiu      sp, sp, (HAL_INTR_STACK_SIZE-4)
```

```
13. skip_stack_switch:
14. ...                                        ; Размаскирование прерываний
15. jal        interrupt_handler               ; Вызов вторичного обработчика
16. nop
17. di                                         ; Запрет прерываний
18. la         t0, intr_nesting                ; Уменьшение счетчика вложения
19. lw         t1, (t0)
20. addiu      t1, t1, -1
21. sw         t1, (t0)
22. movz       sp, s8, t1                      ; Переключение стека обратно
23. ...                                        ; Восстановление контекста
24. eret                                       ; Возврат из обработчика
```

Она начинается с сохранения контекста (многоточием показано сохранение и загрузка регистров с помощью команд lw и sw). Далее, переданный вектор сохраняется в регистре a0 и происходит инкремент счетчика вложения прерываний (строки 3-7). Поскольку процессор аппартно запрещает прерывания при входе в обработчик этот процесс происходит атомарно относительно других прерываний. Если текущий обработчик прервал другой обработчик (счетчик вложения более 1) переключение стека не происходит и выполнение продолжается на том же стеке, в противном случае стек переключается на стек прерываний (строки 8-12). Далее маскируются все прерывания с приоритетом меньше, чем у текущего, а также происходит глобальное разрешение прерываний, с этого момента становится возможно вложение обработчиков. Затем происходит вызов C-функции. Поскольку в строке 3 в регистре a0 был сохранен вектор и этот же регистр используется для передачи аргументов в функции, вектор будет передан в качестве аргумента. Наконец, после возврата из обработчика, происходит декремент счетчика вложения, переключение стека обратно, при достижении счетчиком вложения нуля (строки 18-22), после чего из стека восстанавливается контекст и происходит возврат в прерванный код с помощью специальной инструкции eret.

В многопроцессорной системе обработчики прерываний выполняются на каждом процессоре независимо. Основное отличие заключается в том, что используемые глобальные структуры данных, такие как стек прерываний, присутствуют не в единственном экземпляре, а выделяются для каждого процессора индивидуально, поэтому прежде чем их использовать, каждый процессор должен выяснить свой номер в системе и использовать его как индекс соответствующей структуры данных.

Хотя отсчет времени ведется с помощью аппаратных таймеров, а таймер является обычным аппаратным прерыванием, на первый взгляд нет необходимости выносить источник периодических прерываний в отдельный модуль. Тем не менее, некоторые практические соображения говорят в пользу обратного. В большинстве случаев таймер действительно будет являться обычным аппаратным прерыванием, но из этого правила бывают исключения. Некоторые процессоры, особенно те, которые ориентированы на использование в многопроцессорных конфигурациях содержат внутренний встроенный в архитектуру таймер, при этом его реализация и вектор прерывания может относиться не к аппаратным прерываниям, а к исключениям. Поэтому целесообразно, по аналогии с общей функцией обработки аппаратных прерываний, определить также выделенную

функцию, которая будет периодически вызываться и обеспечивать отсчет времени.

4.4.2 Исключения

По аналогии с аппаратными прерываниями, обработчики исключений требуют использования специальных команд и зачастую работают в специально для них предназначенных режимах процессора, поэтому их поддержка устроена так же как и поддержка прерываний. Существует первичный обработчик, который должен быть установлен в таблицу прерываний пользователем, этот обработчик, после выполнения необходимых низкоуровневых действий вызывает вторичный обработчик написанный на C. Так как исключения, в отличие от прерываний, обычно не предполагают вложения, отдельного стека для этого не используется.

Каждый процессор имеет свой набор исключений, поэтому в задачу HAL входит также унификация исключений между различными процессорами. Так как HAL определяет некоторую абстрактную платформу, для этой платформы определяется также свой собственный набор исключений, а фактические исключения, имеющиеся на конкретном процессоре должны каким-то образом связываться с исключениями определяемыми HAL. Для этого отображения можно использовать табличные методы. Например, HAL может определять 3 исключения: неверная операция (illegal operation), ошибка доступа к памяти (memory error), точка останова (breakpoint). Все исключения поддерживаемые процессором транслируются с помощью таблицы в один из этих трех типов и затем передается в качестве идентификатора исключения.

Библиотка поддержки языка может использовать передаваемый контекст, который специфичен для каждой процессорной архитектуры, для того чтобы выполнить раскрутку стека.

4.4.3 Переключение контекста

От операционной системы требуется реализовать возможность сосуществования на одном процессоре множества параллельных потоков. Поэтому первый вопрос заключается в том, что если у каждого потока есть состояние в виде регистров и стека процессора, то где должно храниться состояние неактивных потоков, если состояние активного в данный момент хранится, собственно, в регистрах процессора? Поскольку контекст процессора является уникальным для данной архитектуры, реализация переключения контекста является одной из задач HAL.

В случае использования концепции потоков, каждый из них имеет собственный контекст, в состав которого входит текущая вершина стека, счетчик инструкций и, возможно, аргумент, передаваемый в функцию потока. Контекст можно определить как структуру, содержащую регистры процессора, которые, в свою очередь, содержат перечисленную выше информацию, поэтому функция инициализации нового контекста могла бы выглядеть так:

```
void cpu_context_create(
      cpu_context_t* context, void* stack, void (*)(void*) entry, void* arg);
```

Эта функция не выполняет никаких действий с оборудованием, а только инициализирует поля структуры, соответствующие определенным регистрам в соответствии с их назначением. Например, в случае ARM, стеку соответствует регистр R13, указателю инструкций - R15, а аргументу функции, согласно конвенции вызова - R0.
Структура контекста не используется сама по себе, а инкапсулируется в некоторый объемлющий объект, такой как поток (Рис. 30).

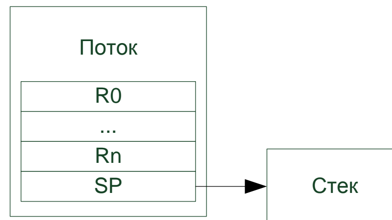

Рис. 30 Хранение контекста в объекте потока

Переключение контекста заключается в том, чтобы выгрузить текущее состояние регистров процессора в такую структуру, после чего загрузить в него состояние из другой структуры.

```
void cpu_context_switch(cpu_context_t* new, cpu_context_t* old);
```

Функция, разумеется, должна выполняться атомарно и ее реализация представляет наибольший интерес. На первый взгляд кажется, что лучше всего реализовать ее синхронным образом. В этом случае переключение контекста будет выглядеть так, как показано на Рис. 31:

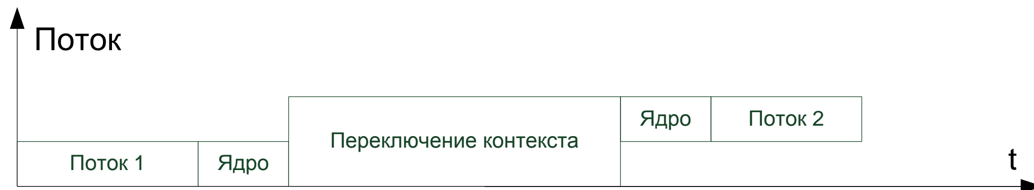

Рис. 31 Переключение контекста в результате синхронного вызова

Приложение вызывает функцию ядра, внутри которой вызывается функция переключения.
При такой реализации возникает некоторая трудность с прерываниями. Переключать контекст внутри прерываний нельзя, поскольку обработчики могут быть вложенными, но, тем не менее, в результате обработки прерывания могут потребоваться переключения контекста. Такое может произойти в том случае, если обработчик прерывания переводит в состояние готовности к выполнению поток с приоритетом выше чем у текущего потока, который был прерван прерыванием. Нужно как-то организовать асинхронный вызов функций ядра, занимающихся планированием и переключением контекста (функций планировщика), который произошел бы после выхода из всех вложенных прерываний. Это может быть достигнуто с помощью счетчика вложения прерываний, имеющего логику "кто сделал последний декремент и достиг нуля, тот вызывает планировщик".
Данный алгоритм, показанный на Рис. 32, гарантирует вызов планировщика после выхода

из всех вложенных обработчиков.

Рис. 32 Переключение контекста после обработки прерываний

Происходит как бы инжекция вызова планировщика во все обработчики прерываний, которые, с точки зрения потока, выглядят так, как будто эти вызовы производит синхронно сам поток.

Хотя данная реализация имеет право на жизнь и успешно применяется в ряде коммерческих ОСРВ, нельзя не отметить несколько существенных проблем.

Во-первых, поскольку работа планировщика происходит на стеке одного из потоков, каждый поток в системе, помимо основного рабочего пространства стека, должен также предусматривать на стеке место, достаточное для вызова планировщика.

Во-вторых, функция синхронного переключения контекста хотя и не является сложной алгоритмически, тем не менее является чрезвычайно сложно поддерживаемой и трудной в разработке. Эта функция работает так, что войдя в нее в одном потоке, "выход" происходит уже в другом, с учетом смены стека это подразумевает, что в пределах одной функции имеется точка, за которой вместе с контекстом "переключаются" также и все локальные переменные. Ну и, разумеется, в настолько низкоуровневом коде существенно ограничены возможности отладки. Все это вместе приводит к тому, что написанный текст программы является лишь вершиной айсберга и без досконального знания подводной части поддерживать этот код затруднительно.

В-третьих, во время переключения контекста могут происходить новые прерывания и, если планировщик вызывается безусловно в момент выхода из всех обработчиков, то операцию переключения нужно защищать от параллельного выполнения самой себя. При этом нужно учитывать тот факт, что вход в критическую область выполняется в одном потоке, а выход из нее - уже в другом. Если происходит переключение на потоки, которые уже выполнялись ранее, то вход в них осуществляется в той же самой точке синхронного переключения контекста и выход из критической секции планировщика произойдет на более поздних этапах выполнения этой функции, но что делать, если поток ранее еще не выполнялся и это первое переключение на него? В этом случае выполнение будет идти по сценарию показанному на Рис. 33. Здесь также возможны различные варианты: либо любой поток должен начинаться с функции-заглушки, которая должна выйти из критической секции, либо нужно воссоздавать на стеке нового потока ситуацию, как будто этот поток был остановлен в функции переключения контекста.

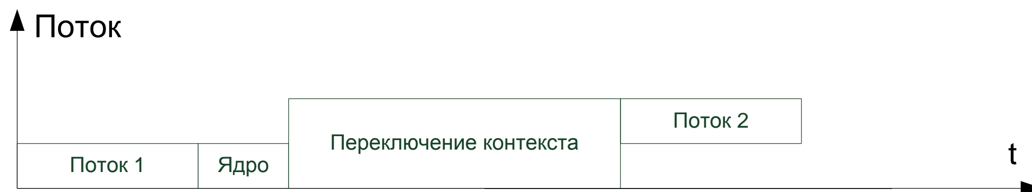

Рис. 33 Запуск нового потока в результате синхронного вызова планировщика

И, наконец, в-четвертых, если вызов планировщика происходит в результате прерывания, то частично контекст уже сохранен при входе в обработчик, и при синхронном вызове происходит частичное дублирование контекста на стеке потока, что отрицательным образом сказывается на производительности.

Существует другой подход к этой проблеме - **отложенное переключение контекста** и асинхронная функция переключения. Стек прерываний существует как раз для того, чтобы не отводить в стеке каждого потока память, достаточную для выполнения всех вложенных прерываний, поэтому, если рассматривать планировщик как низкоприоритетное прерывание, то решаются сразу все проблемы. описанные выше. Если аппаратные прерывания делают что-то, что может потребовать переключения контекста, они устанавливают запрос на прерывание "планировщик", которое, будучи наименее приоритетным автоматически выполнится после выхода из всех обработчиков, как показано на Рис. 34.

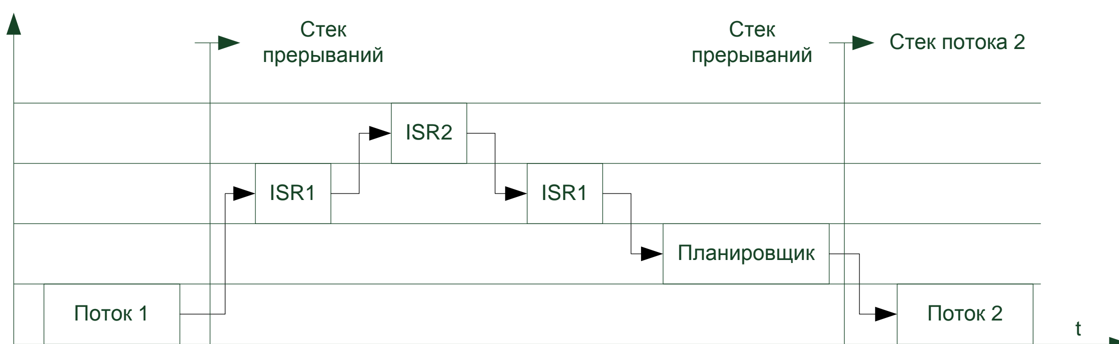

Рис. 34 Отложенное переключение контекста

Поскольку планировщик теперь работает на выделенном стеке прерываний, нет нужды использовать низкоуровневую магию при переключении контекста, достаточно переключить указатель стека прерванного потока, откуда будет восстановлен контекст после выхода из прерывания.

При инициализации нового потока, на его стеке создается структура, соответствующая контексту, без необходимости сложной синхронизации с блокировками планировщика. Такая же структура возникает всякий раз, когда поток прерывается обработчиком прерывания или планировщиком. То есть контекст целиком содержится на стеке, поэтому внутри структуры контекста хранится только указатель на стек. Такой подход к хранению контекста показан на Рис. 35.

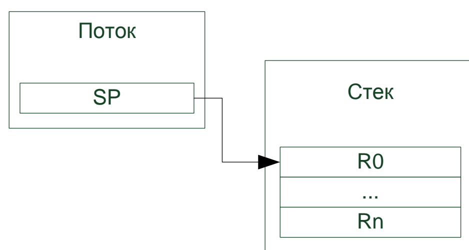

Рис. 35 Хранение контекста потока в стеке

Детально процесс отложенного переключения контекста показан на Рис. 36 (переменная, хранящая указатель на текущий стек показана как SP).

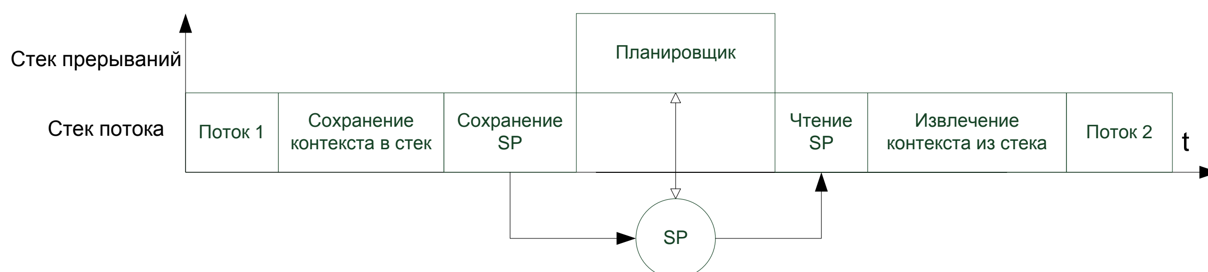

Рис. 36 Процесс переключения контекста

После сохранения контекста текущего потока обработчиком прерывания в стек происходит переключение на стек прерываний. Ключевой особенностью является сохранение указателя на сохраненный контекст в некоторой глобальной переменной, так что обработчики прерываний работающие на выделенном стеке имеют к этой переменной доступ и могут ее изменять. После завершения обработки прерывания, восстановление контекста начинается с чтения этой переменной и восстановения контекста из того стека, на который она указывает, поэтому с помощью переключения этой переменной можно выбирать тот или иной поток на который нужно переключиться.

При трактовке планироващика как прерывания, некоторую трудность представляют синхронные вызовы из потока. Как раз этот момент выглядел наиболее естественно в случае синхронного вызова. Но, поскольку прерывания всегда имеют приоритет выше чем потоки, запрос на прервание из самого потока можно с некоторыми оговорками трактовать как вызов функции, поскольку передача управления обработчику должна произойти немедленно. В отличие от аппаратных прерываний, которые возникают в результате работы аппаратуры, прерывание соответствующее планировщику должно инициироваться программно. Подробности реализации программных прерываний будут рассмотрены в следующем разделе.

Поскольку программные прерывания и реализация с их помощью планировщика обладают рядом преимуществ по сравнению с альтернативами, логично включить программные прерывания в слой абстракции окружения и считать что этот полезный функционал поддерживается всегда, но может быть реализован программно, в случае отсутствия аппаратной поддержки.

4.4.4 Программные прерывания

Хотя программные прерывания являются очень полезным функционалом, путь к этой идее был довольно долгим, поэтому многие процессоры лишены таких возможностей. В данном разделе будут рассмотрены программные прерывания для случая, когда они поддерживаются аппаратно. Вариант с полностью програмной реализацией будет рассмотрен далее в разделе, посвященном маске прерываний, так как в программной реализации эти компоненты очень тесно связаны.

Существует два основных вида программных прерываний - синхронные и асинхронные. Синхронные обычно представлены специальной командой, выполнение которой приводит к вызову обработчика. В качестве примера можно привести команды INT процессоров семейства x86 или SVC для процессоров ARM. Они имеют много общего с исключениями, только роль исключительной ситуации играет сам факт выполнения этой инструкции. В остальном синхронные программные прерывания подобны вызовам функций.

Асинхронные программные прерывания устроены более сложно, в этом случае процессор должен иметь откладывать запуск обработчика до тех пор, пока для этого не возникнут условия. Эти условия обычно связаны с приоретизацией прерываний, если запрос был установлен из обработчика более высокого приоритета, вызов обработчика программного прерывания откладывается до выхода из него. В случае же запроса из менее приоритетной сущности, например из потока, асинхронные прерывания работают так же как и синхронные - вызов обработчика происходит немедленно.

Именно такое поведение и требуется для прерывания планировщика, при его запросах из других прерываний вызов должен быть отложен, при запросе же из потоков, которые предполагаются наименее приоритетными сущностями в системе, вызов должен происходить так же как и в случае синхронных вызовов.

Для обработки программных прерываний, по аналогии с аппаратными, используется внешняя функция, которая должна быть предоставлена приложением. Так же как и аппаратные прерывания, эта функция выполняется на стеке прерываний. С точки зрения программиста, отличие аппаратных и программных прерываний заключается только в способе выполнения запроса.

Такие процессоры как MIPS и ARM Cortex-M поддерживают асинхронные программные прерывания аппаратно, поэтому описанные выше функции просто отображаются на соответствующий аппаратный интерфейс.

```
request_swi:           ; Процессор поддерживает 2 вектора прерываний, они передаются
                       ; как аргумент 1 или 2 в регистре A0
  mfc0 t1, _CP0_CAUSE ; Чтение регистра сопроцессора CP0 содержащего биты запроса
  ins t1, a0, 8, 2     ; Установка запроса программного прерывания
  mtc0 t1, _CP0_CAUSE ; Обновление регистра сопроцессора
  sync 0               ; Барьер
  jr ra                ; Возврат из функции
  nop
```

Для программных прерываний предусмотрено два вектора. Для нужд ОС используется только один из них, обработчик которого похож на рассматривавшийся ранее обработчик

аппаратных прерываний. Поскольку переключение контекста может происходить только как результат работы аппаратного прерывания, сохранение и восстановление глобальной переменной указывающей на стек прерванного потока происходит только в этом обработчике:

```
swi_entry:
   1. ...                         ; Сброс запроса программного прерывания
   2. ...                         ; Сохранение контекста
   3. la        k0, swi_stack_frame ; Сохранение текущего стека в переменную
   4. sw        sp, (k0)
   5. la        sp, intr_stack      ; Переключение на стек прерываний
   6. addiu     sp, sp, (HAL_INTR_STACK_SIZE-4)
   7. ...                         ; Увеличение счетчика вложения прерываний
   8. ...                         ; Размаскирование прерываний
   9. jal       swi_handler         ; Вызов вторичного обработчика
   10. nop
   11. di
   12. ...                        ; Уменьшение счетчика вложения прерываний
   13. la       k0, swi_stack_frame ; Чтение указателя стека из переменной
   14. lw       sp, (k0)
   15. ...                        ; Восстановление контекста из стека
```

Обработка в целом аналогична обработке аппаратных прерываний за исключением двух моментов. Во-первых стек, куда был сохранен контекст, сохраняется не в регистре, а в переменной (строки 3-4), на выходе из обработчика стек, на который будет произведено переключение также читается из переменной, то есть он может быть переключен в функции-обработчике. Во-вторых, поскольку приоритеты прерываний настроены таким образом, что данное прерывание имеет наименьший приоритет, оно не может прервать другие обработчики, то есть не может быть вложенным, поэтому переключение стека должно происходит безусловно.

Подобным образом программные прерывания реализуются и в других процессорах. Например процессоры ARM Cortex-M, по аналогии с MIPS, имеют встроенное в архитектуру и систему команд асинхронное програмное прерывание PendSV, которое работает примерно так же, только процессор выполняет аппаратно еще больше работы, в частности, сам переключает стеки. Более того, этот процессор содержит даже аппаратно реализованный регистр, указывающий на стек прерванного потока. Пример реализации обработчика программного прерывания для этих процессоров приведен ниже:

```
swi_entry:                  ; В момент входа в прерывание стек уже переключен
  mrs   r0, psp             ; Чтение указателя стека в R0
  stmdb r0!, {r4-r11}       ; Сохранение контекста в стек потока
  msr   psp, r0             ; Обновление указателя стека потока
  push  {lr}
  bl    swi_handler         ; Вызов вторичного обработчика (он может переключить PSP)
  pop   {lr}
  mrs   r0, psp             ; Чтение регистра PSP и загрузка контекста из нового стека
  ldmia r0!, {r4-r11}
  msr   psp, r0
  bx lr
```

Возможны также другие подходы. Например процессоры ARM Cortex-A не имеют

программных прерываний встроенных в архитектуру, но зато такие прерывания реализует контроллер прерываний. Он имеет специальный регистр, запись в который позволяет запрашивать прерывания с определенным вектором.

Поскольку архитектура ARM предполагает существование одного вектора на все аппаратные и программные прерывания, используется общая функция низкоуровневого входа в прерывание, а далее по вектору выясняется, какой именно обработчик нужно вызывать.

```
void arm_intr_handler(void)
{
  const unsigned int vector = get_vector();
  intr_enable();
  switch(vector)
  {
    ...
    case N: swi_handler(); break;
  }
  intr_disable();
  // Обновление состояния контроллера прерываний
}
```

Важным вопросом является также обеспечение правильной работы синхронного вызова. Если процессор не гарантирует немедленного перехода к обработчику сразу после получения запроса, может использоваться следующий шаблон: перед установкой запроса устанавливается некоторый флаг в глобальной переменной, который должен безусловно сбрасываться обработчиком программного прерывания. После установки флага и запроса прерывания размещается цикл активного ожидания сброса этого флага, хотя, как правило, такой цикл не требуется.

4.4.5 Фрейм прерывания

Поскольку функции обработчику программных прерываний необходимо манипулировать переменной, в которой сохраняется указатель на контекст прерванного потока, для этого должно быть соответствующее высокоуровневое API. Хотя для большинства процессоров этот указатель является переменной, с которой можно работать по обычным правилам языка C, в некоторых случаях, например для процессоров Cortex-M, вместо этой переменной используется аппаратно реализованный регистр PSP.

Сохраненный в стеке контекст процессора называется кадром или **фреймом прерывания**, он является своеобразным видом адреса возврата, сохраняемым вместе со всеми остальными регистрами. Сохраняемая на стеке структура имеет вид

```
typedef struct _intr_frame_t
{
// регистры
}
intr_frame_t;
```

Она специфична для каждого типа процессора. Функции работы с фреймом прерывания позволяют создавать и модифицировать контекст процессора кроссплатформенным

образом. Интерфейс обычно включает функции выделения нового фрейма относительно заданного указателя, которая дополнительно производит его первичную инициализацию, а также функций для модификации фрейма. Поскольку состав структуры фрейма для каждого процессора уникален, используются условные значения полей для модификации, например адрес исполнения и аргумент.
Пример реализации функции выделения фрейма для ARM:

```
intr_frame_t* intr_frame_alloc(intr_frame_t* base)
{
  intr_frame_t* frame = base - 1;
  frame->CPSR = (CPSR_MODE_SYSTEM | CPSR_A_BIT | CPSR_F_BIT);
  return frame;
}
```

Относительно указанного базового адреса выделяется новый фрейм, который инициализируется значениями по умолчанию для управляющих регистров (предполагается, что память для фрейма была предварительно заполнена нулями вызывающим кодом).

```
void intr_frame_modify(intr_frame_t* frame, int reg, uintptr_t val)
{
  switch(reg)
  {
    case KER_FRAME_ENTRY: frame->r15 = val; break;
    case KER_FRAME_ARG0: frame->r0 = val; break;
    default: break;
  }
}
```

Для модификации фрейма используется два условных значения, соответствующих адресу функции и ее аргумента.
Функция выделения и модификации контекста может использоваться не только для потоков. Например, если используется модель вытесняющих акторов, при возникновени прерывания, если выясняется, что текущий актор должен быть вытеснен, из обработчика прерывания запрашивается программное прерывание, которое выделяет на стеке новый фрейм, сответствующий новому актору и передает ему управление. С точки зрения прерванного актора это выглядит так, как будто новый актор является прерыванием которое его прервало.
Так как все состояние процессора инкапсулировано в структуру фрейма, на основе этого интерфейса можно реализовать универсальный механизм отложенного переключения контекста. Структура контекста в этом случае содержит указатель на фрейм.

```
typedef struct _cpu_context_t
{
  intr_frame_t* frame;
}
cpu_context_t;
```

Инициализация контекста заключается в выделении нового фрейма и установки точки входа и аргумента.

```
void cpu_context_create(
      cpu_context_t* context, void* stack, void (*entry)(void*), void* arg)
{
  hal_intr_frame_t* frame = hal_intr_frame_alloc( stack );
  hal_intr_frame_modify(frame, KER_FRAME_ENTRY, entry);
  hal_intr_frame_modify(frame, KER_FRAME_ARG0, arg);
  context->frame = frame;
}
```

Переключение контекста, которое должно выполняться только в обработчике программного прерывания, заключается в переключении указателя текущего фрейма. После выхода из программного или аппаратного прерывания, новый контекст будет загружен уже из нового фрейма, и, таким образом, продолжится выполнение уже другого потока.

Этот код остается универсальным и его можно применять для разных процессоров, поэтому создается впечатление, что на самом деле переключение контекста относится к переносимой части, а HAL должен реализовывать лишь работу с фреймами. Это утверждение справедливо лишь до тех пор, пока не используются различные режимы работы процессора и защита памяти, в этом случае могут потребоваться специфические реализации модуля переключения контекста для каждого из процессоров. Эти специальные случаи будут рассмотрены далее.

4.4.6 Уровень маски прерываний

Для корректной работы ПО необходима возможность блокировки прерываний на время работы с разделяемыми структурами данных. При этом оборудование довольно сильно отличается в аппаратном плане: разные процессоры имеют разные контроллеры прерываний и возможности управления ими.

Хотя у всех процессоров есть возможность блокировать аппаратные прерывания (команды вида "разрешить-запретить"), требуется также и возможность блокировать программные. В том случае, если они поддерживаются аппаратно, это обычно не представляет проблем, но в том случае, когда программные прерывания не поддерживаются и реализуются проагрммно, способ их маскировки также должен быть программно реализован в HAL.

Если идти по пути простого вывода низкоуровневых интерфейсов устройств в C, то получится, что необходимы несколько не очень удобных для использования интерфейсов: для управления аппаратными прерываниями, программными прерываниями, а также контроллером прерываний, так как в некоторых архитектурах программные прерывания (а значит и возможности для их маскировки) реализуются не непосредственно ядром процессора, а контроллером прерываний. Введение в интерфейс HAL контроллера прерываний не всегда оправдано, так как в некоторых архитектурах он вполне может отсутствовать.

Учитывая тот факт, что прерывания обладают приоритетом (это необходимо для корректной и предсказуемой реализации вложенных прерываний), можно прийти к идее управления маскированием прерываний с помощью управления приоритетом. Идея эта заключается в том, что вместо бинарного флага прерывания, который позволяет

маскировать/размаскировать все прерывания какого-то типа, вводится некоторое число, которое указывает "насколько маскировать". В роли этого числа часто выступает приоритет прерывания, повышение приоритета процессора до уровня N приводит к тому что все прерывания, приоритет которых меньше чем N оказываются замаскированными. В предельном случае, когда уровень принимает значения "максимум" и 0 всё вырождается в классический флаг запрета прерываний.
Значение приоритета по отношению к асинхронным событиям называется в разных ОС по разному. Одна из первых реализаций такого подхода была предложена в ОС BSD, где функции установки приоритета сокращенно назывались SPL (set priority level). В данной книге название SPL будет также использоваться для названия **приоритета по отношению к асинхронным событиям** (system priority level), чтобы исключить неоднозначность и подчеркнуть его отличие от приоритетов как потоков, так и прерываний.
Уровень SPL, так же как и флаг прерывания, является атрибутом процессора. Код с большим SPL может прерывать код с меньшим, но не наоборот. Интерфейс для управления SPL включает функции для его повышения и понижения:

```
spl_t raise_spl(spl_t new_spl);
void lower_spl(spl_t new_spl);
```

Таким образом, в рамках одного API реализуется логика управления как процессорным флагом запрета прерываний, так и логика управления контроллером прерываний. Количество приоритетов, как и количество уровней SPL получается зависимым от используемого процессора, поэтому с точки зрения универсальности важно определить такие значения типа spl_t, которые были бы допустимы для всех поддерживаемых процессоров. Во-первых, должны поддерживаться как минимум два уровня, соответствующие всем запрещенным и разрешенным прерываниям. Их можно назвать SYNC-уровнем и LOW-уровнем соответственно. Приоритеты распределяются таким образом, чтобы программные прерывания обладали наименьшим приоритетом. Это соответствует логике работы программных прерываний для переключения контекста, которая обсуждалась выше. Поскольку программные прерывания используются для реализации планировщика, соответствующий им уровень можно называть уровнем диспетчеризации DISPATCH. Его использование необходимо в тех случаях, когда хотелось бы предотвратить вытеснение и переключение контекста, но таким образом, чтобы аппаратные прерывания оставались разрешенными.
Помимо этих трех уровней, могут поддерживаться также и дополнительные соответствующие приоритетам определенных векторов. В случае необходимости маскировки отдельных уровней приоритета можно предусмотреть специальные переносимые интерфейсы для выяснения количества уровней и их соответствия векторам.
В простейшей реализации, допустимо существование всего двух приоритетов, SYNC и LOW. В этом случае логика работы рассмотренных двух функций эквивалентна логике работы функций разрешения и запрета прерываний. В случае Cortex M эти функции напрямую отображаются на аппаратные возможности. Существует специальный регистр

BASEPRI, который хранит текущий приоритет относительно приоритетов прерываний. С помощью его модификации можно управлять, какие прерывания смогут прерывать текущий выполняемый код:

```
spl_t raise_spl(spl_t new_spl)
{
  spl_t prev = prio_to_spl(get_basepri());
  set_basepri(spl_to_prio(new_spl));
  return prev;
}

void lower_spl(spl_t new_spl)
{
  set_basepri(spl_to_prio(new_spl));
}
```

В том случае, если процессор не поддерживает ни SPL ни программные прерывания аппаратно, их реализация целиком возлагается на HAL. В качестве примера можно рассмотреть x86. Для реализации программных прерываний предусмотрена глобальная переменная, хранящая активные запросы программных прерываний. Текущее значение SPL также хранится в глобальной переменной.

При возникновении прерывания, определяется уровень SPL того кода, который был прерван (по значению глобальной переменной). Поскольку в момент выполнения начального обработчика другие прерывания запрещены, доступ к переменной безопасен. Затем вычисляется SPL текущего прерывания по его вектору и маска в контроллере прерываний устанавливается таким образом, чтобы отразить новый уровень. После глобального разрешения прерываний и их обработки проверяется уровень SPL прерванного кода, если он был больше либо равен DISPATCH, это значит, что программные прерывания были запрещены и происходит выход из обработчика с возвратом прежнего уровня SPL. В том же случае, если прерванный SPL был ниже, чем DISPATCH, проверяется, нет ли ожидающих запросов программных прерываний, и если таковые есть - для них вызывается функция-обработчик. Если выполнение обработчика прерывается другими прерываниями, они увидят прерванный урвоень SPL=DISPATCH и не вызовут обработчик еще раз. Таким образом, после каждого обработчика проверяется состояние программных прерываний со ступенчатым понижением уровня SPL.

В случае синхронного запроса программных прерываний из приложения все то же самое происходит синхронным образом: вызывается обертка, которая обеспечивает переключение стека и повышение уровня SPL до DISPATCH, затем, уже на стеке прерываний вызывается обработчик. Пример обработки аппаратного прерывания в процессорах x86.

```
extern int swi_pending;
spl_t current_spl;

void internal_intr_handler(intr_frame_t* frame)
{
  spl_t prev_spl = current_spl;
  spl_t new_spl = <определение нового SPL по текущему вектору>
```

```
  current_spl = new_spl;

  // Установка маски в контроллере прерываний, соответствующей SPL
  pic_update(new_spl);

  // Разрешение прерываний и вызов обработчика
  ...

  // Понижение SPL до уровня DISPATCH
  pic_update(DISPATCH);
  current_spl = DISPATCH;

  if(prev < DISPATCH)
  {
    while (swi_pending)
    {
      swi_pending = 0;
      intr_enable();
      // вызов обработчика программных прерываний
      ...
      intr_disable();
    }
  }

  pic_update(prev_spl);
}
```

Таким образом можно реализовать и более чем один вектор программных прерываний с различными приоритетами.

4.5 Взаимодействие между процессорами

Взаимодействие процессоров в многопроцессорной системе также сильно специфично для архитектуры и типа процессора, поэтому задача эффективной абстракции этих механизмов также возлагается на HAL. Также следует разделять API для симметричных и асимметричных многопроцессорных систем. Будет рассмотрен симметричный случай из-за его большей распространенности.

В первую очередь у ОС должна быть возможность узнать текущий процессор, а также общее количество процессоров в симметричной системе.

```
unsigned int get_cpu_count(void);
unsigned int get_current_cpu(void);
```

Первая функция возвращает количество процессоров, которые присутствуют в текущей конфигурации, то есть целое положительное число от 1 до некоторого максимума. Поскольку многим приложениям (и ОС в том числе) требуется знать максимальное количество процессоров еще на этапе компиляции, для правильного распределения памяти под структуры данных, выделенных для каждого из процессоров, вводится специальный макрос, который и указывает этот максимум - HAL_MP_CPU_MAX.

Как правило, эти функции реализуются на ассемблере. Например, выяснение собственного номера для процессоров ARM происходит с помощью чтения регистров системного сопроцессора CP15.

```
get_current_cpu:
  mrc  p15, 0, r0, c0, c0, 5
  and  r0, r0, $0xf
  bx   lr
```

Так как основным способом общения между процессорами в симметричной системе является обмен прерываниями (inter-processor interrupt, IPI), должна быть соответствующая функция и в высокоуровневом интерфейсе.

```
void request_ipi(unsigned int processor, spl_t level);
```

В качестве аргументов она принимает номер процессора, которому нужно послать межпроцессорное прерывание, а также уровень SPL этого прерывания. В большинстве случаев необходимо запрашивать обработчики программных прерываний на других процессорах, то есть прерывания уровня DISPATCH.

Для эффективного использования кроссплатформенного ПО на однопроцессорных системах может быть предусмотрена реализация модуля поддержки многопроцессорности в виде заглушки. В этом случае все функции отображаются на константы.

```
#define CPU_MAX 1
#define get_current_cpu() 0
#define get_cpu_count() 1
```

В случае одного процессора межпроцессорные прерывания эквиваленты отправке прерываний самому себе, то есть обычным программным прерываниям, рассмотренным ранее. Рассмотренный интерфейс является довольно низкоуровневым, поэтому обычно на его основе делаются уже более продвинутые конструкции.

4.5.1 Спинлоки

Если в однопроцессорных конфигурациях можно обойтись запретом прерываний как основным механизмом обеспечения атомарности и синхронизации, то в многопроцессорной системе уже не обойтись без атомарных инструкций и реализуемых на их основе примитивов. Основным низкоуровневым механизмом синхронизации в многопроцессорных системах являются **спинлоки**, с помощью которых реализуется взаимное исключение. Простейший спинлок на основе операции атомарного обмена мог бы выглядеть так:

```
while(atomic_swap(&lock, 1) == 0) ;
```

Поскольку атомарные инструкции довольно тяжеловесны и требуют особой обработки на шине, можно применить также такую оптимизацию, что в цикле ожидания выполняется обычная, а не атомарная инструкция, что снижает нагрузку на шину.

Хотя спинлоки могут быть реализованы переносимым образом на основании атомарных операций, предоставляемых процессором, многие процессоры имеют аппаратные

механизмы предназначенные для оптимизации спинлоков и поэтому, чтобы не отказываться от этих возможностей, имеет смысл реализовать спинлоки как один из интерфейсов, предоставляемых HAL. Несмотря на то, что, как правило, критические секции защищаемые с помощью спинлоков, имеют небольшой размер, спинлоки в современных ядрах ОСРВ используются повсеместно, поэтому суммарное время, проведенное в активном ожидании, может быть значительным. Особенно это проявляется тогда, когда система с большим числом процессоров использует глобальные спинлоки, защищающие некоторый общесистемный ресурс.

В качестве примеров аппаратных оптимизаций спинлоков можно привести инструкцию pause, появившуюся в процессорах Intel Pentium 4. Эти процессоры могли выполнять два потока на одном физическом процессоре (технология HyperThreading), и инструкция использовалась для подсказки процессору, что текущий код является циклом ожидания и можно отдать приоритет другому потоку. Инструкция должна использоваться в качестве содержимого цикла ожидания:

```
while(...)
{
  pause();
}
```

В процессе выполнения цикла ожидания процессор не выполняет никакой полезной работы, поэтому впоследствии появились средства позволяющие приостанавливать выполнение цикла до тех пор, пока не появятся основания считать, что спинлок свободен. В частности, процессоры ARM имеют пару инструкций SEV/WFE (Send EVent, Wait For Event), предназначенные для использования в циклах ожидания. Инструкция WFE, если она поддерживается, приостанавливает выполнение кода до выполнения на каком-либо другом процессоре инструкции SEV. Цикл ожидания в этом случае будет выглядеть следующим образом:

```
spinlock_acquire:
lock:
   1. ldrex r1, [r0]         ; Чтение переменной спинлока
   2. cmd r1, #LOCKED
   3. wfeeq                  ; Спинлок занят, приостановка процессора
   4. beq lock               ; Переход к следующей итерации цикла
   5. mov r1, #LOCKED
   6. strex r2, r1, [r0]     ; Попытка захватить спинлок
   7. cmp r2, #0x0
   8. bne lock               ; Попытка завершилась неудачно, повторение цикла
   9. dmb                    ; Барьер
   10. bx lr

spinlock_release:
   1. dmb
   2. mov r1, #UNLOCKED      ; Освобождение спинлока
   3. str r1, [r0]
   4. dsb                    ; Барьер для завершения предыдущей операции
   5. sev                    ; Пробуждение всех процессоров ожидающих на WFE
   6. bx lr
```

В процессорах x86 также со временем были введены инструкции MONITOR/MWAIT, которые могут использоваться в циклах ожидания.

Со спинлоками и их использованием в многозадачных ОС связана также еще одна проблема. Спинлоки используются для защиты данных разделямых между процессорами, причем каждый процессор работает независимо от других и может выполнять потоки, планирование которых не связано с выполнением кода на других процессорах. Поэтому может возникнуть ситуация, когда сразу после захвата спинлока поток вытесняется. При этом спинлок остается заблокированным и другие процессоры при попытке его захвата будут ждать не просто в течении времени выполнения критической секции, но также и все время, пока вытесненный поток-владелец спинлока вновь не получит процессорное время и не освободит спинлок. Понятно, что это время может быть довольно долгим по сравнению с временем выполнения критической секции, защищаемой спинлоком.

Выходом из этой ситуации является запрет на использование спинлоков в вытесняемом коде. Поскольку вытеснение возможно в любой момент на уровне SPL соответствующем прикладным программам, запрет использования спинлоков в вытесняемом коде эквивалентен утверджению, что использование спинлоков запрещено на уровнях SPL ниже DISPATCH.

Дискуссионным вопросом является необходимость запрета прерываний во время выполнения критической секции. Если сразу после захвата спинлока на уровне DISPATCH поток был прерван обработчиком прерывания (возможно даже несколькими вложенными обработчиками) то другие процессоры опять же будут вынуждены ждать гораздо больше, чем длина критической секции. Хотя, в отличие от вытеснения потока, обработка прерываний завершится за относительно небольшое время, желательно избегать также и этих задержек, то есть на время захвата спинлока следует запрещать прерывания для минимизации потенциальных задержек и времени активного ожидания на других процессорах.

До сих пор рассматривались только проблемы вытеснения потока захватившего спинлок на другом процессоре, в то же время, если такое событие произошло на одном и том же процессоре, последствия могут быть гораздо более печальными. Если обработчик прерывания прерывает код, который только что выполнил захват спинлока, и сам пытается захватить тот же спинлок, возникает тупиковая ситуация или **взаимоблокировка**: обработчик прерывания не может завершиться, пока поток не освободит спинлок, а поток не может получить управление, пока не завершится обработка прерывания. Дальнейшая работа системы невозможна. Поэтому, зачастую вводятся дополнительные требования, помимо запрета на использование спинлоков в вытесняемом коде. Либо должны использоваться соглашения, позволяющие избежать одновременного использования спинлоков потоками и обработчиками прерываний, либо захват спинлока должен атомарно приводить и к запрету прерываний, чтобы исключить попытки захвата ими спинлоков. Компромиссный вариант заключается в том, чтобы позволить самому коду, захватывающему спинлок, указать, до какого уровня необходимо повышать SPL, поэтому интерфейс спинлоков может выглядеть так:

```
void spinlock_acquire_at_level(spinlock_t* spinlock, spl_t spl);
void spinlock_release(spinlock_t* spinlock);
```

Так как изначальный SPL должен быть восстановлен после освобождения спинлока, он должен сохраняться в структуре самого спинлока:

```
typedef struct _spinlock_t
{
  volatile unsigned int spinlock;
  spl_t old_spl;
}
spinlock_t;
```

Реализация без использования оптимизаций для ARM:

```
void spinlock_acquire_at_level(spinlock_t* spinlock, spl_t spl)
{
  do
  {
    const spl_t caller_spl = raise_spl(spl);
    if(atomic_swap(&(spinlock->spinlock), 1) == 0)
    {
      spinlock->old_spl = caller_spl;
      break;
    }
    lower_spl(caller_spl);
    wfe_instruction_wrapper();
  }
  while(1);
  dmb_instruction_wrapper();
}

void spinlock_release(spinlock_t* spinlock)
{
  const spl_t old_spl = spinlock->old_spl;
  dmb_instruction_wrapper();
  spinlock->spinlock =  0;
  dsb_instruction_wrapper();
  sev_instruction_wrapper();
  lower_spl(old_spl);
}
```

Помимо спинлоков могут использоваться также и другие низкоуровневые примитивы. Например семафоры или барьеры, которые могут быть представлены не просто бинарным значением занято-свободно, а счетчиком, с возможностью ожидания конкретного значения. Для использования возможных аппаратных оптимизаций все эти интерфейсы также имеет смысл реализовывать как интерфейсы HAL.

4.5.2 Спинлоки с очередью

При освобождении классического спинлока, рассмотренного ранее, молчаливо предполагается, что после его освобождения, ожидающие потоки получат управление в некотором справедливом порядке. На практике порядок особождения очень сильно зависит от аппаратной реализации атомарных операций, и в случае наличия нескольких ожидателей, в принципе не определен порядок получения ими спинлока. Возможна даже ситуация, называемя голоданием, когда отпустивший спинлок поток вновь начинает его ждать, а тот, кто его захватил, снова передает его тому кто его только что освободил. То

есть владение спинлока будет постоянно переходить между двумя потоками, а все остальные ожидатели могут вообще никогда не закончить ожидание. Очевидно, такая система не отличается предсказуемостью работы. Кроме того, частый доступ к разделяемой памяти увеличивает нагрузку на шину и снижает производительность.

Для решения этой проблемы были изобретены спинлоки с очередью (queued spinlocks), называемые также MCS-спинлоками (по именам изобретателей Mellor-Crummey и Scott) [7]. Они обеспечивают определенную независимую от оборудования дисциплину обработки очереди ожидания (по мере освобождения спинлока, ожидатели получают его по принципу FIFO), а кроме того, переменная, используемая для ожидания, не является разделяемой, что снижает нагрузку на шину и в целом способствует улучшению производительности. Платой за это является существенно более сложная реализация, по сравнению с обычными спинлоками.

Идея заключается в том, что каждый процессор создает локальную структуру, использующуся для ожидания, а сама структура спинлока используется только для определения предыдущего владельца.

```
typedef struct _spinlock_node_t // Локальная структура на стеке каждого потока
{
  struct _spinlock_node_t* next;
  volatile unsigned int locked;
}
spinlock_node_t

typedef struct _spinlock_t
{
  spinlock_node_t* lock;
}
spinlock_t;

void spinlock_acquire(spinlock_t* spinlock, spinlock_node_t* node)
{
  node->next = NULL;
  spinlock_node_t predecessor = atomic_swap_ptr(&spinlock->lock, node);
  if(predecessor != NULL)
  {
    node->locked = 1;
    predecessor->next = node;
    while(node->locked) ;
  }
}

void spinlock_release(spinlock_t* spinlock, spinlock_node_t* node)
{
  if(node->next == NULL)
  {
    if(atomic_cas_ptr(&spinlock->lock, node, NULL) == node)
      return;
    else
      while(node->next == NULL) ;
  }
  node->next->locked = 0;
}
```

Фактически, глобальная переменная используется как голова односвязного списка, в

который выстраиваются процессоры с помощью операции атомарного обмена (Рис. 37). Для того чтобы освободить спинлок, поскольку ожидание происходит на локальной переменной, процессор-владелец должен выполнить запись не в глобальную переменную, а в локальную структуру следующего по порядку ожидателя. Поскольку между заполнением этой структуры и обновлением глобальной переменной проходит некоторое время, может получиться так, что при освобождении потребуется подождать, пока локальная структура не будет заполнена.

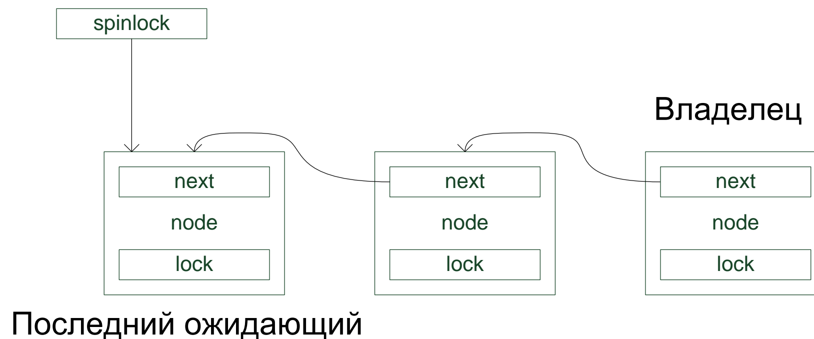

Рис. 37 Спинлоки с очередью

Поскольку спинлоки с очередью требуют усложнения интерфейса, в дальнейшем в примерах кода будут использоваться обычные спинлоки, но следует учитывать, что в нагруженных системах следует отдавать предпочтение спинлокам с очередью.

4.6 Разделение привилегий

До сих пор интерфейсы HAL предполагали использование в приложениях, не предполагающих разделения привилегий, любому потоку были доступны любые вызовы. Для простых встроенных систем это не представляет особой проблемы, потому что, как правило, у пользователя нет возможности запускать в системе код, который не был предусмотрен изначально. Тем не менее, как уже описывалось выше, по мере роста сложности ПО возникает необходимость изолировать выполнение ответственных функций таким образом, чтобы второстепенные компоненты системы не могли на них повлиять. Для такого разделения используются специальные аппаратные возможности предоставляемые процессором. Этот функционал сильно различается от процессора к процессору и включает в себя несколько механизмов, среди которых возможности по обращению непривилегированного кода к привилегированному, а также возврат из привилегированного в непривилегированный после обработки запроса, возможности защиты памяти и так далее. В задачу HAL входит обобщение, абстракция и предоставление высокоуровневых интерфейсов для этих возможностей.
В отличие от прерываний, которые работают сходным образом на всех процессорах, возможности защиты различаются более существенно. В частности, до этого обсуждалось только два режима, привилегированный и непривилегированный, но многие процесоры предоставляют больше уровней разграничения, например процессоры intel x86 предоставляют четыре таких уровня. Поскольку невозможно реализовать больше уровней защиты, чем процессор поддерживает аппаратно, для максимальной универсальности HAL использует только два уровня.
Хотя все состояние процессора связанное с привилегиями обычно содержится в одном из

регистров, идея предоставления возможности чтения и записи этого регистра не очень удачна. Дело в том, что доступ к этому регистру (он имеет различные названия у разных процессоров) часто программно ограничен. Например, руководство по разработке ПО для процессоров семейства AVR32 утверждает, что явная запись в регистр хранящий текущее состояние процессора приводит к неопределенному поведению [8]. Процессоры intel x86 вообще не содержат в непосредственно доступных регистрах информации о привилегиях. В процессорах ARM Cortex-M регистр, хранящий привилегии доступен, но для обработчиков прерываний его содержимое игнорируется (они всегда являются привилегированными), то есть процессоры часто содержат скрытое состояние, недоступное для прямой модификации со стороны ПО, и привилегии часто являются частью этого скрытого состояния.

Возникает вопрос, если запись в регистр привилегий не всегда возможна, то каким образом вообще выполняется управление привлегиями? Идея реализации непривилегированного режима исходит из наблюдения, что если все процессоры поддерживают прерывания и прерывания могут происходить в любой момент, то процессор должен, как минимум, поддерживать переход в привилегированый режим по прерыванию и возврат из него обратно в тот режим, в котором он находился на момент получения прерывания. То есть состояние процессора, в том числе скрытое, должно быть частью контекста.

HAL должен предоставлять возможности создания непривилегированных потоков, которые выполняются с ограниченными возможностями модификации памяти, а также должен абстрагировать обращения непривилегированных потоков к ядру. Поскольку привилегии являются атрибутом потока, в том случае, если они поддерживаются, должна быть специальная реализация фрейма прерываний, который, возможно, включает в себя дополнительную информацию о привилегиях. Кроме того, в функции модификации контекста intr_frame_modify, можно добавить возможности для изменения точки входа в пользовательский режим и аргумента. Использование этих аргументов должно заменить структуру фрейма таким образом, как если был прерван непривилегированный код, то есть возврат в такой фрейм приведет также и к переключению режима. В связи с наличием во фреймах прерываний дополнительной информации, могут потребоваться специальные реализации модуля прерываний. Его интерфейс остается переносимым, но процедуры входа и выхода могут учитывать возможность изменения при этом текущего уровня привилегий.

При этом в системе возможно сосуществование потоков, которые выполняются в привилегированном режиме и непривилегированных потоков (называемых также пользовательскими), которые выполняются с пониженными привилегиями.

Традиционно, привилегированный режим называется **режимом ядра** (kernel mode), а непривилегированный - **режимом пользователя** (user mode). Эти названия пришли из ОС общего назначения, которые выполняют в привилегированном режим ядро ОС, а пользовательские приложения - в непривилегированном. В мире встроенного ПО это разделение является довольно размытым, так как возможно выполнение отдельных пользовательских приложений в привилегированном режиме процессора.

После того, как непривилегированный поток создан, он лишен возможности прямых

обращений к функциям HAL, поскольку они выполняются в привилегированном режиме, за всеми потенциально опасными действиями такой код должен обращаться к ядру системы с помощью механизма **системных вызовов**.

К счастью, среди разработчиков процессоров существует относительный консенсус отнносительно того, как именно выполняется системный вызов. Обычно, для этого используются синхронные программные прерывания или ловушки. Непривилегированный поток выполняет специальную инструкцию и тем самым передает управление привилегированной процедуре которая может проанализировать запрос и выполнить требуемые действия.

Так как ловушки и синхронные программные прерывания являются подвидом прерываний, то, если не считать модулей, которые подключаются при наличии привилегированного режима (то есть они либо есть, либо нет без необходимости написания нескольких вариантов) все изменения главным образом сосредоточены в модулях прерываний.

В некоторых процессорах, например в x86, со временем были разработаны расширения позволяющие переключать режимы без обращения в память с помощью специальных инструкций SYSENTER/SYSEXIT. Такой подход похож на прерывания в RISC: данные помещаются в регистры и затем считываются оттуда при выполнении возврата. Использование этих инструкций может улучшить производительность приложений, так как несложные системные вызовы могут быть выполнены без необходимости сохранения в памяти большого количества данных.

4.6.1 Стек ядра

Самый простой вариант, как может быть реализована поддержка процессов: системный вызов используется для того чтобы повысить потоку привилегии и далее он выполняет код ОС так же, как и привилегированный поток. Некоторые ОС реализуют такой тип защиты, однако подобный подход может быть полезным только при отладке. Многие структуры данных, включая информацию о привилегиях, хранятся на стеке потока, поэтому любой поток может изменить структуры данных ОС на стеке и тем самым вызвать крах. Кроме того, такой дизайн имеет много проблем с безопасностью: путем модификации стека и аргументов передаваемых через него можно влиять на выполнение кода ОС, а через него и на другие процессы, что расходится с концепцией процессов, как механизма защиты и изоляции.

Решением данной проблемы является выделение непривилегированному потоку двух стеков. Потоки выполняемые в привилегированном режиме используют как и ранее один стек, потоки же выполняемые в пользовательском режиме имеют два стека, один используется в непривилегированном режиме, а второй используется в то время, когда поток выполняется в режиме ядра, во время обработки системного вызова, он называется стеком ядра потока. Прерывания обрабатываются на отдельном стеке как и ранее. Таким образом в любой момент времени выполнение кода может происходит на одном из трех стеков: стек прерываний, стек ядра текущего потока и пользовательский стек текущего потока (Рис. 38).

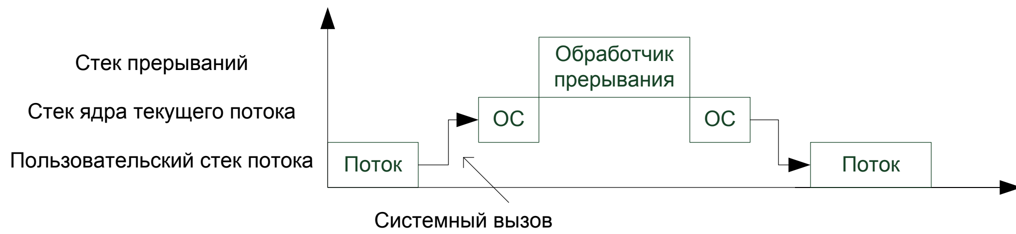

Рис. 38 Стек ядра для потока

Так как необходимо обеспечить выполнение кода системного вызова на стеке ядра текущего потока, переключение стека обычно реализуется аппаратно. В момент выполнения инструкции системного вызова процессор атомарно переходит в режим ядра и переключает стек (откуда этот стек берется, зависит от конкретного процессора).

Атомарность нужна для того, чтобы исключить сценарий, когда точно в момент между переключением режима и переключением стека произойдет аппаратное прерывание. Обработчик может решить, что раз выполнение происходит уже в режиме ядра, то контекст можно сохранить на текущий стек и тем самым дать пользовательским потокам доступ к потенциально критической информации.

Из наличия стека ядра следуют дополнительные отличия в работе компонента прерываний и исключений для HAL, поддерживающего процессы и разделение привилегий. В тот момент, когда поток выполняется на своем пользовательском стеке может произойти прерывание и первичный обработчик HAL должен сохранить контекст в стек ядра прерванного потока. Некоторые процессоры делают все эти действия аппаратно, тогда как в других происходит автоматическое переключение на стек прерываний (ARM) а выяснение стека ядра потока и сохранение контекста возлагается на HAL.

4.6.2 Системные вызовы

Реализация системного вызова несколько нетипична, по сравнению с обсуждавшимися ранее компонентами. Поскольку этот компонент инкапсулирует в себе раздел между двумя режимами, он состоит из двух интерфейсов: один из них предоставляется непривилегированным потокам для того чтобы они могли инициировать системный вызов, а вторая часть находится в привилегированном режиме и вызывается в ответ на системные вызовы из непривилегированных потоков. Оба этих компонента логически составляют одно целое и должны изменяться согласованно (Рис. 39).

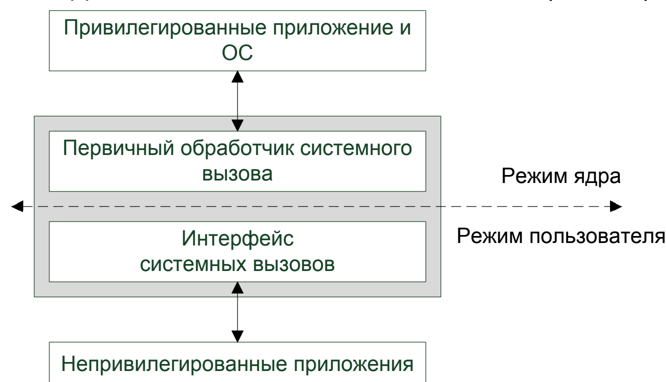

Рис. 39 Абстракция границы между режимами

Помимо переключения режима, в задачу этих модулей входит также абстракция способа передачи параметров между режимами, который также может зависеть от платформы. Обычно используется подход с передачей параметров через регистры, поэтому, поскольку количество регистров отличается у разных процессоров, нужно определить такое количество аргументов, которое, с одной стороны, было бы достаточно для большинства функций ОС, а с другой стороны, было бы реализуемо на большинстве процессоров по возможности без дополнительных накладных расходов. Хорошим приближением является 6 параметров. Большинство функций ОС имеют меньшее количество аргументов, а с другой стороны, процессоры имеющие менее 8 регистров общего назначения не очень распространены.

Таким образом, со стороны непривилегированного кода определяется функция с 6-ю аргументами, которая инициирует системный вызов:

```
register_t syscall(register_t,register_t,register_t,register_t,register_t,register_t);
```

Аргументы должны быть определены таким образом, чтобы была возможность использовать их и как целые значения, передаваемые по значению, и указатели, что позволит передавать дополнительные аргументы по ссылке, если 6-и основных окажется недостаточно.

Со стороны привилегированного кода, по аналогии с прерываниями, существует функция, вызываемая в ответ на запрос уже в привилегированном режиме и получающая параметры, указанные пользователем в виде аргументов. По правилам C допускается возврат только одного значения или указателя, поэтому при необходимости вернуть больше, используются остальные аргументы. Вызов функции-обработчика происходит на стеке, который был указан как привилегированный при инициализации непривилегированного контекста.

Один из аргументов обычно используется в качестве номера функции, которая называется системным вызовом.

```
register_t
syscall_handler(
       register_t, register_t, register_t, register_t, register_t, register_t
);
```

Реализация вызова обычно осуществляется с использованием ассемблера, например вариант для x86 помещает параметры в регистры, читая их со стека и вызывает синхронное программное прерывание с вектором, который должен быть зарезервирован для системных вызовов:

```
syscall:
  movl 0x4(%esp), %eax
  movl 0x8(%esp), %ebx
  movl 0xC(%esp), %ecx
  movl 0x10(%esp), %edx
  movl 0x14(%esp), %esi
  movl 0x18(%esp), %edi
```

```
  int $(SYSCALL_INTERRUPT_ID)
  ret
```

После переключения режима процессор вызывает обработчик, который сохраняет регистры в стек ядра текущего потока и вызывает высокоуровневый обработчик:

```
lowlevel_syscall_entry:
  ...        ; Сохранение регистров и формирование фрейма прерывания на стеке ядра
  push %esp
  call hal_syscall_handler
  addl $4, %esp
  ...        ; Восстановление регистров из фрейма прерывания
  iret

void hal_syscall_handler (hal_intr_frame_t* frame)
{
  frame->eax = hi_level_syscall_handler(
       frame->eax, frame->ebx, frame->ecx, frame->edx, frame->esi, frame->edi);
}
```

Функция-обработчик системных вызовов, так же как и обработчик прерываний, должна быть предоставлена извне.

Существует ряд системных вызовов, единственная функция которых - вернуть пользователю некоторое значение, которое хранится в переменных ядра. Доступ пользователя к этим переменным должен быть закрыт. В качестве примера можно привести идентификаторы потока/процесса и подобные вещи. Для оптимизации подобных вызовов может быть выделен регион памяти который настроен таким образом, что пользователь может его читать, но не может обновлять. В то время как эти переменные обновляются, они обновляются и в этой области. Далее библиотека, работающая в пользовательском режиме может узнать (например с помощью системного вызова выполняемого однократно при старте системы) адрес этой области и впоследствии запросы пользователя на чтение этих переменных могут быть обслужены библиотекой полностью в пользовательском режиме без накладных расходов и без необходимости преключения режимов.

4.6.3 Реализация переключения контекста

В том случае, если фрейм прерывания содержит все необходимые поля включая оба стека, необходимых для непривилегированных потоков, для переключения контекста может использоваться тот же компонент, реализующий работу с фреймами прерываний что и для систем не поддерживающих разделение привилегий. К сожалению, в большинстве процессоров фрейм прерывания не содержит всей нужной информации, поэтому механизм переключения становится специфическим для каждого процессора. По аналогии с фреймом, для контекста вводится дополнительная функция, которая позволяет создать непривилегированный контекст на базе привилегированного.

```
void cpu_context_create_unprivileged(
      cpu_context_t* context, void* usr_stk, void (*entry)(void), void* arg);
```

Эта функция должна вызываться после функции создания привилегированного контекста, поскольку последняя устанавливает стек привилегированного режима и создает в нем фрейм, для которого уже возможны модификации режима. Например, в процессорах x86, в том случае когда контекст соответствует непривилегированному режиму (это определяется по младшим битам регистра CS, являющегося частью конеткста), процессор автоматически извлекает из стека не только адрес возврата, но также и пользовательский стек. Поэтому функция cpu_context_create создает в переданном ей стеке фрейм, а функция cpu_context_create_unprivileged модифицирует регистр CS в этом фрейме чтобы он соответствовал непривилегированному режиму, а затем дописывает во фрейм информацию о пользовательском стеке , после чего точка входа перезаписывается на точку входа в непривилегированный режим.

Некоторые модификации требуются также в функции переключения контекста. В процессорах x86, после восстановления контекста из фрейма соответствующего непривилегированному потоку, регистр стека указывает на пользовательский стек, поэтому возникает вопрос, откуда взять указатель на стек ядра для текущего потока (процессор, как мы помним, должен переключиться на него аппаратно!) если возникнет аппаратное прерывание? Для этого используется специальное поле в расположенной в памяти таблице, называемой сегментом состояния задачи (task state segment, TSS), этот довольно неочевидный механизм сложился исторически. У процессора есть дополнительный регистр, который указывает на TSS, называемый TR. Так что процедура переключения контекста, помимо переключения текущего фрейма, хранимого в глобальной переменной, требует также модификации значения в TSS.

Процессоры ARM Cortex-M используют еще более странный подход: при возникновении аппаратных прерываний, автоматическое сохранение контекста происходит на тот стек который был в момент прерывания, то есть в случае выполнения пользовательского потока - на его пользовательский стек. От возможности его модификации и несанкционированного повышения привилегий спасает лишь то, что, в отличие от x86, состояние процессора, хранящее привилегии в стек не сохраняется, что позволяет разместить его в структуре контекста, которая недоступна пользователю. В этом случае программно реализовывать приходится в том числе и стек ядра: контекст должен содержать оба стека, а функции входа-выхода из прерываний должны использовать тот или иной стек в зависимости от текущего уровня привилегий.

4.7 Защита памяти

4.7.1 MPU

Защита памяти с использованием MPU и MMU существенно различается, особенно в том случае, когда для MMU используется подход с аппартной реализацией прохода по страницам, в этом случае необходимо рамещение в памяти структур таблиц. Вообще говоря, если требуется обеспечить только возможности защиты памяти (установки прав доступа к регионам) то как MPU так и MMU могут быть унифицированы в пределах одного интерфейса, который HAL отображал бы на то или иное устройство. С другой стороны, MMU обладает гораздо более продвинутыми возможностями, и не всегда разумно от

этих возможностей отказываться только ради того, чтобы интерфейс MMU стал похож на интерфейс MPU. Эти соображения говорят в пользу того, что HAL не абстрагирует защиту памяти полностью, а предоставляет отдельные интерфейсы как для MPU, так и для MMU, дальнейшая их унификация осуществляется уже на более высоком уровне в соответствии с его потребностями, в задачу HAL входит лишь унификация различных типов MMU и MPU.

Так как MPU это внешнее по отношению к процессору устройство, которое может быть опциональным, он представлен таким же образом как все другие устройства - набором синхронных функций, главной из которых является функция установки прав доступа к региону.
Поскольку через MPU проходит любое обращение в память, регионы должны быть описаны в том числе и для привилегированного кода.
В случае если ядро не поддерживает нескольких процессов, MPU может быть настроен один раз во время инициализации системы, например таким образом как показано ниже (Рис. 40):

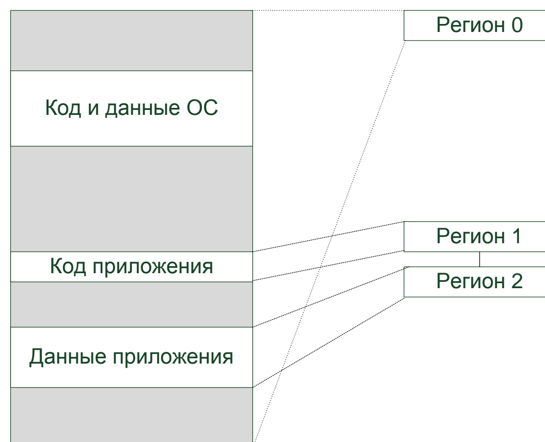

Рис. 40 Пример настройки MPU

Регион 0 определяет права доступа для привилегированных приложений, поэтому он описывает всю дступную память. Два других региона описывают память, которой пользуется непривилегированное приложение. Обычно нужно 2 региона, код имеет права доступа только для чтения, а данные - для чтения и записи.
Если поддерживается несколько процессов которые надо изолировать друг от друга, во время переключения на другой процесс необходимо перепрограммировать MPU, обычно это происходит в обработчике программного прерывания, как часть процедуры переключения контекста.

4.7.2 MMU

MMU развивает идею изоляции далее, предлагая каждому процессу собственное адресное пространство. Хотя в системах жёсткого реального времени MMU используется редко из-за непредсказуемости, связанной с необходимостью кэширования отображения адресов, тем не менее из-за его широкого распространения нельзя совсем обойти его

вниманием.
Как уже говорилось в главе 2, существует два основных типа реализации MMU, с программным и аппаратным проходом по таблицам для трансляции адреса. И в первом и во втором случае интерфейс для работы с MMU довольно простой и написать переносимую реализацию обычно не составляет труда.
Как правило, определяется объект "адресное пространство", который соответствует каталогу страниц, далее для этого объекта определяются операции вроде установки прав доступа, которые модифицируют записи в таблице. Поскольку работа с таблицей отображения происходит только через эти функции, работает ли с ней процессор аппаратно или же TLB управляется HAL в данном случае неважно.
Переключение контекста выполняется более просто, чем в варианте с MPU - требуется лишь загрузить указатель на новую таблицу страниц в соответствующие регистры процессора. На вызывающий код при этом возлагается ответственность, чтобы этот код присутствовал также и в новом отображении.

4.7.3 Распределение адресного пространства

MPU предполагает единственное физическое адресное пространство, для которого задаются права доступа. Тот же подход можно использовать и с MMU. Код ОС и HAL должны присутствовать в адресном пространстве каждого из процессов, для возможности выполнения системных вызовов. Кроме того, ОС сама по себе скомпилирована для работы по определенному адресу и с целью минимизации накладных расходов хотелось бы, чтобы она присутствовала по этому адресу во всех процессах. Эти рассуждения приводят к тому, что ядро должно быть расположено в выделенной области виртуальных адресов, которая не пересекается с виртуальными адресами выделенными пользовательским процессам.
Поскольку, в отличие от MPU, у MMU нет ограничения адресного пространства, в адресном пространстве каждого процесса создается область, специально предназначенная для ОС (обычно в старших адресах), как показано на Рис. 41.

Рис. 41 Распределение адресного пространства

Помимо прочего, такая организация адресного пространства позволяет по самому указателю сказать, какие он имеет права доступа (если он находится в верхней части адресного пространства - он может использоваться только в привилегированном режиме,

если такой указатель получен от непривилегированного потока то он заведомо некорректен).

Отображение также может зависеть от аппаратуры, поэтому логично что структура адресного пространства для данной платформы также определяется HAL. Например, при использовании устройств отображенных на память, доступ к ним со стороны пользовательских приложений должен быть закрыт. Обычно, они располагаются в старших частях физического адресного пространства, и, таким образом, оказываются автоматически доступными для ОС и недоступыми для приложений, в иных случаях возможны модификации адресного пространства таким образом, чтобы исключить доступ пользователя к устройствам.

Из-за размещения ОС в старших адресах, возникает другая проблема: в момент начального старта системы, когда страничное преобразование еще не включено, доступна только физическая память, которой может быть меньше чем виртуального адресного пространства, а поскольку ОС скомпилирована для работы в старших адресах, она оказывается неработоспособной, поскольку любые обращения к старшей памяти будут истолкованы процессором как обращения по такому физическому адресу, что на многих процессорах немедленно вызовет исключение. Решение несколько нетривиально. Для HAL, поддерживающих MMU и трансляцию адресов, используется специфический компонент начальной инициализации. Функция, которая передает управление ОС, пишется на ассемблере таким образом, чтобы не обращаться по старшим адресам явно. Эта функция подготавливает временную таблицу страниц, в которой создается отображение участка памяти ядра как на верхние, так и на нижние части адресного пространства (Рис. 42).

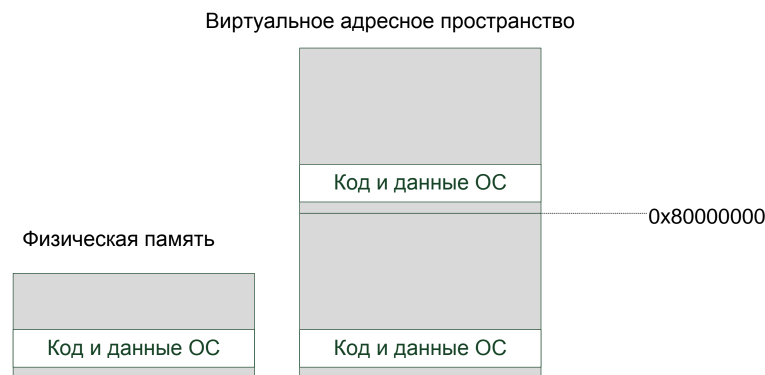

Рис. 42 Адресное пространство во время инициализации

Затем включается страничная адресация, поскольку существует оба отображения, код продолжает выполнение в младших адресах, затем текущий стек и счетчик команд перемещаются в старшие адреса (в код этой же функции, отображенный по другому адресу), после чего отображение младших адресов снимается. После этих действий управление передается в высокоуровневый код инициализации, который работет со старшими адресами.

4.8 Пример реализации приложения на интерфейсах HAL

Рассмотренный набор компонентов HAL достаточен для разработки встроенных приложений без поддержки процессов и изоляции. Хотя HAL является низкоуровневым

интерфейсом и предназначен для реализации поверх него более удобных для использования интерфейсов, таких как потоки и прочее, несложные встроенные приложения могут разрабатываться непосредственно на интерфейсах HAL. Это также может потребоваться при разработке приложений с особыми требованиями к производительности и времени реакции, когда помимо этих требований необходимо обеспечить также и переносимость.

В качестве иллюстрации можно рассмотреть простую реализацию многопоточности на основе рассмотренных интерфейсов. Будет рассмотрен один поток, который будет активироваться обработчиком прерывания. Хотя в рассматриваемом примере поток будет только один с целью минимизации кода примера, его можно легко промасштабировать на большее количество прерываний и потоков. В то время, когда потоки и обработчики прерываний не активны, процессор должен исполнять поток простоя, который переводит процессор в состояние пониженного энергопотребления.

Каждый поток, как поток-обработчик так и поток простоя представлены контекстом. Поскольку в качестве стека потока простоя используется тот стек, на котором происходит инициализация и вызывается функция app_init, явно определять стек нужно только для рабочих потоков. Наконец, используется флаг, указывающий на необходимость переключения на рабочий поток, он устанавливается в активное состояние из обработчика прерывания.

```
static cpu_context_t thread_context;
static cpu_context_t idle_context;
static unsigned int stack[STK_SZ];
static bool thread_ready = false;
```

Обработчик аппаратных прерываний проверяет вектор, и, если он пришел от ожидаемого источника (который указан как SIGNAL_VECTOR), то устанавливается флаг запуска потока и запрашивается программное прерывание для переключения контекста. Обработчик прерывания может обрабатывать множество источников, а также сопоставленных с ними флагов, подход с программным прерыванием позволяет корректно передавать управление планировщику даже в случае вложенных прерываний.

```
void interrupt_handler(void)
{
  if(get_current_vector() == SIGNAL_VECTOR)
  {
    thread_ready = true;
    request_software_interrupt(DISPATCH);
  }
}
```

Обработчик программного прерывания выступает в роли планировщика, он использует глобальную переменную в которой хранит указатель на текущий контекст. Поскольку используется всего один поток, то "планировщик" состоит из одной строки: если рабочий поток активен, то ему и следует отдать управление, в противном случае работает поток простоя. Если после определения потока выясняется, что текущий контекст не

соответствует тому, который должен быть, выполняется операция переключения контекста и обновляется состояние глобальной переменной.

```
static cpu_context_t* current_context;

void software_interrupt_handler(void)
{
  cpu_context_t* const next = thread_ready ? &thread_context : &idle_context;

  if(current_context != next)
  {
    cpu_context_switch(next, current_context);
    current_context = next;
  }
}
```

Сам рабочий поток устроен предельно просто: после действий по обработке прерывания он повышает SPL, для того чтобы исключить одновременный доступ к флагу, сбрасывает собственный флаг готовности и запрашивает программное прерывание. Если между понижением SPL и вызовом планировщика произойдет новое аппаратное прерывание, поток запустится снова, в противное случае планировщик переключит контекст на поток простоя.

```
void thread(void* arg)
{
  while(1)
  {
   ... // Обработка прерывания
   raise_spl(SYNC);
   thread_ready = 0;
   request_software_interrupt(DISPATCH);
   lower_spl(LOW);
  }
}
```

Выполнение начинается с функции инициализации, которая устанавливает глобальную переменную планировщика в начальное состояние (активен поток простоя), а также создает начальный контекст рабочего потока.

```
void app_init(void)
{
  current_context = &idle_context;
  cpu_context_create(&thread_context, &stack[STK_SZ-1], thread, NULL);
  // Разрешение прерываний и размаскирование источника
}
```

В случае наличия нескольких источников прерываний, без использования HAL, нечто подобное пришлось бы так или иначе разрабатывать на стороне приложения. Важно отметить, что получившееся приложение будет одинаково работать на любом процессоре, при этом накладные расходы минимальны и зависят от используемого процессора. На платформах вроде ARM Cortex M практически все функции имеют аппаратный эквивалент и задача HAL сводится лишь к выражению этих возможностей на

языке высокого уровня. Другие платформы, такие как x86, напротив, требуют в большей степени программной реализации этих возможностей.

Пример может быть расширен в том числе на многопроцессорные системы, в этом случае, правда, из средств синхронизации остаются только спинлоки, но для простых и критичных к производительности приложений этого может быть достаточно.

4.9 Резюме

Одним из самых важных критериев качества встроенного ПО вообще и встроенных ОСРВ в частности, является их переносимость. Часто возникает ситуация, когда несколько продуктов используют сходное ПО на разных аппаратных платформах, поэтому возможность писать кроссплатформенное ПО упрощает тестирование и сокращает время выхода продукции на рынок. Из-за того, что портирование необходимо выполнять для каждой платформы, реализация переносимого интерфейса должна быть как можно более простой и исключить дублирование кода, поэтому популярен подход, когда в качестве объекта унификации используется непосредственно аппаратура. При таком подходе переносимый интерфейс является интерфейсом некоторой абстрактной машины, спроектированный таким образом, что он реализуем на всех платформах, которые планируется поддерживать. Набор программных компонентов абстрагирующих те или иные возможности аппаратуры формируют слой абстракции оборудования HAL.

В зависимости от возможностей конкретного процессора, те функции, которые не поддерживаются аппаратно, реализуются программным образом, поэтому накладные расходы вносимые HAL различны для разных процессоров.

[1] Borland Turbo C User's Guide 1987 (p. 224)
[2] Intel 64 and IA-32 architectures software developer's manual volume 2: Instruction set reference
[3] MIPS32 Instruction set quick reference Revision 01.01
[4] ARMv7-M Architecture Reference Manual 2006-2010
[5] ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition 1996-1998, 2000, 2004-2012, 2014, 2018
[6] Faheem Sheikh "Key Findings: booting a symmetric multiprocessing (SM) RTOS from bare metal"
[7] Mellor-Crummey, Scott; Algorithms for scalable synchronization on shared-memory multiprocessors 1991
[8] Atmel AVR32 Architecture document (p. 12)

5 Построение ядра

Темы главы:

- Проектирование интерфейса ядра ОСРВ
- Взаимодействие потоков и обработчиков прерываний
- Архитектура ядра с поддержкой многопроцессорных систем
- Реализация планировщика, потоков и таймеров
- Реализация пассивного ожидания и объектов синхронизации

Интерфейсы HAL, рассмотренные в предыдущей главе, предоставляют переносимый интерфейс на языке высокого уровня и в принципе могут использоваться для разработки несложных встроенных приложений. Хотя на основе этих интерфейсов могут быть реализованы такие объекты как потоки, от пользователя требуется реализация слишком большого количества инфраструктурного кода, такого как механизмы планирования и взаимодействия между потоками. Поскольку в большинстве приложений эта инфраструктура будет содержать много общий частей, логично на основе HAL реализовать еще более высокоуровневый интерфейс ОС, который был бы более удобен для использования.

Несмотря на долгую историю развития операционных систем, устоявшегося и не содержащего оговорок определения ядра ОС на данный момент не существует. В настольных и серверных системах часто под ядром подразумевают часть ОС работающую в привилегированном режиме процессора. Для встроенных систем это определение не всегда применимо, так как зачастую разделение режимов вообще не используется и ОС довольно сложно отделить от приложения, так как они вместе составляют единый бинарный образ.

Согласно некоторым определениям [1], **ядро ОСРВ** реализует потоки и обеспечивает их планирование, а также предоставляет механизмы для синхронизации и коммуникации между потоками, что и будет подразумеваться под ядром в данной книге.

В данной главе речь пойдет о проектировании так называемой libOS. Реализации ядра без разделения режимов, где ОС представляет собой библиотеку, которая компонуется с приложением.

5.1 Политика и механизм

ОС выполняет в системе две основные функции: расширение и адаптация аппаратного обеспечения под нужды прикладного ПО, а также управление ресурсами [2]. Под расширением аппаратуры подразумевается реализация более высокоуровневых и удобных для использования интерфейсов, нежели те, что предоставляются HAL. Реализация управления ресурсами предполагает, что изначально (после загрузки) владельцем всех системных ресурсов, таких как физическая память и устройства, является сама ОС. Прикладные программы, которым нужны ресурсы, обращаются к ОС за их предоставлением, а последняя выполняет функции арбитра. Эти две задачи, абстракция оборудования и управление ресурсами, ортогональны друг другу. Управление ресурсами подразумевает использование некоторой политики, касающейся того, кому и на каких

условиях эти ресурсы могут быть предоставлены. Эти политики, разумеется, могут различаться для разных ОС, однако механизмы, в соответствии с которыми эти политики работают, остаются относительно неизменными, потому что работают над аппаратными абстракциями. Эти соображения постепенно привели разработчиков к принципу отделения политики от механизма таким образом, чтобы на основе одного и того же механизма можно было реализовывать различные политики.

Этот принцип настолько важен, что стоит остановиться на нем подробнее. В качестве иллюстрации можно рассмотреть абстрактную систему управления автомобильным движением. Ее "аппаратной" составляющей являются, например, светофоры. Для централизованного управления ими система может предоставлять высокоуровневые интерфейсы, унифицирующие понятие светофора и скрывающие возможные различия в управлении ламповыми и светодиодными светофорами, однако эта система ничего не знает о правилах дорожного движения, она лишь предоставляет механизм, с помощью которого эти правила могут быть реализованы. Самые различные правила движения, проезд перекрестков, а также прочие связанные правила, формируют политику и могут быть реализованы с использованием единого механизма и без внесения в него изменений.

Такой подход позволяет строить многоуровневые системы, каждый уровень которых предоставляет механизмы, использующиеся верхними слоями для формирования политики. Чем ниже уровень, тем более общими являются реализуемые им механизмы.

5.2 Исполнительная подсистема

Хотя ядро и предоставляет интерфейсы для использования потоков, приложениям часто требуется и многие другие интерфейсы, такие как управление памятью, позволяющее выделять и обобождать блоки памяти, возможности взаимодействия с устройствами и так далее. Большинство из этих компонентов, в свою очередь, пользуются интерфейсами ядра, а потому логически находится выше его уровня абстракции. Расширенное ядро, которое предоставляет дополнительные возможности помимо реализации потоков и средств их взаимодействия называется исполнителем, или **исполнительной системой** (executive) [1] (Рис. 43).

Рис. 43 Ядро и исполнительная подсистема

В функции исполнительной системы может входить также реализация файлов, обеспечение централизованного доступа к устройствам и тому подобное.
Характерным свойством ОС вообще и ее ядра в частности является кроссплатформенность, поэтому общая структура встроенного ПО в терминах слоев выглядит как показано на Рис. 44.

Рис. 44 Уровни абстракции типичной встраиваемой системы

Во многих встраиваемых системах, особенно на основе микроконтроллеров, предполагается существование только одного приложения, которое имеет доступ ко всем имеющимся ресурсам. При этом многие из этих ресурсов могут быть распределены статически во время компиляции и сборки системы, так что необходимость в менеджере ресурсов, который их распределяет во время работы системы отпадает. В таких системах исполнительная подсистема может полностью отсутствовать, а приложение может пользоваться сервисами ядра напрямую.

5.3 Проектирование интерфейса

5.3.1 Выделение ресурсов

Как и многие программы, ядро ОС использует различные данные для своей работы. В частности, объекты, которыми оперирует ОС, такие как потоки, должны быть расположены в памяти, которую кто-то должен выделить. Кроме того, память требуется также для разлчных объектов HAL, таких как, например, контексты. Это выделение может происходить как динамически, во время работы системы, так и статически, во время компиляции и сборки.
При разработке ядра, еще до проработки его архитектуры, необходимо задаться вопросом, какую степень динамики распределения ресурсов это ядро должно обеспечивать. Определение границы, за которой заканчивается статика и начинается динамика, очень важно, потому что это влияет как на производительность, так и на сложность всего устройства. Использование излишней динамики впустую расходует вычислительные ресурсы процессора и снижает срок автономной работы. Кроме того, это дополнительный код, который, помимо того, что нуждается в тестировании, может содержать ошибки и иметь проблемы с безопасностью.

С другой стороны, недостаток динамики приводит к принципиальной невозможности реализовать какие-то функции системы, которые могут стать известны только во время работы. Так что идеальный баланс может быть определен только для конкретного устройства исходя из его функций и решаемых задач.

Как правило, встроенная система решает какую-то одну конкретную задачу (или конечный их набор), все необходимые ресурсы для которой, такие как, например, память, могут быть распределены статически. Тем не менее, ядро не должно явно определять политику распределения памяти, потому что это идет вразрез с описанным ранее принципом разделения политики и механизмов. Также ядро не может заранее знать, какие ресурсы потребуются приложениям, поэтому обычно применяется подход, описанный ниже.

Многие процессоры имеют определенный формат внутренних структур данных, таких как таблицы страниц, дескрипторов, векторов прерываний и прочего. При этом сам процессор не выделяет память под эти ресурсы и не занимается управлением этой памятью, предполагается, что некоторая внешняя система должна создать в памяти эти структуры и вызвать с помощью некоторых инструкций сервисы процессора, которые тем или иным образом будут эти данные интерпретировать. Ядро ОС ведет себя аналогичным образом, с той лишь разницей, что его сервисы реализуются программно. Память под нужные ядру объекты и структуры выделяется какой-то сущностью вне ядра (в некоторых ОС эту функцию выполняет исполнительня подсистема [3]), которая занимается распределением ресурсов, в простейших системах это может происходить во время компиляции, когда все нужные ядру объекты распределяются компоновщиком. При вызове функций ядра все нужные структуры передаются в них в качестве параметров. Такой подход позволяет использовать концепции распределения ресурсов как статические так и динамические.

5.3.2 Потоки

Начать обсуждение реализации ядра следует с потока, как главной сущности, которая ядром реализуется. Идея заключается в том, чтобы предоставить любой независимой части программы свой собственный виртуальный процессор, на котором эта программа будет выполняться [4].

Использование потоков позволяет сделать приложение более модульным, расширяемым и поддерживаемым, так как выполнение потока относительно независимо от других потоков и каждый из них может выполнять определенную задачу. В числе задач ядра - обеспечить параллельное сосуществование многих потоков, которых может быть больше, чем физических процессоров в системе. Очевидно, это предполагает временнОе разделение ресурсов процессора: потоки занимают процессор по очереди, а в задачи операционной системы входит реализация инфраструктуры для такого разделения.

Поскольку текущее состояние процессора может быть сохранено и затем восстановлено, это дает необходимую базу для реализации потоков: в определенных точках кода, называемых точками планирования (scheduling points) выполнение текущего потока может быть приостановлено, состояние процессора в этот момент сохранено, и затем загружено состояние нового потока, который продолжает выполняться.

С точки зрения пользовательского приложения, поток является функцией, которая

выполняется на собственном стеке псевдопараллельно с другими потоками. Из-за того, что большинство встраиваемых систем имеют реактивную природу (обеспечивают реакцию на внешние события) поток, как правило, является функцией содержащей внутри себя бесконечный цикл обрабатывающий входящие события.

```
void thread_function(void* arg) {
  for(;;) {
    wait_for_event(...);
    handle_event(...);
  }
}
```

Способность приостановливать выполнение и затем возобновлять его является одним из важных свойств потока, которое следует из аналогичной функциональности процессоров, многие из которых имеют инструкции, позволяющие приостановить выполнение инструкций до прихода событий извне, таких как прерывания.
По аналогии с состояниями процессора, поток также имеет состояния: готов к работе, выполняется в данный момент, приостановлен, завершен. После создания потока он обычно находится либо в приостановленном, либо в готовом к выполнению состоянии. По мере планирования, поток меняет свое состояние между выполняющимся и готовым к выполнению, в случае вытеснения его другим потоком. Если поток приостанавливает выполнение по собственной инициативе, он переходит в состояние ожидания, выйти из которого он может только с помощью другого потока (так как сам лишается возможности выполняться). Наконец, после того как поток заканчивает работу, он переходит в состояние "завершен". По ряду технических причин, которые будут обсуждаться далее в этой главе, переход в состояние завершения происходит только из состояния выполнения. Хотя многие ОСРВ предоставляют возможности для принудительного завершения потоков без их ведома, в общем случае это является как минимум плохим стилем программирования, а как максимум - ошибкой, поскольку выполнявшийся поток мог работать с данными или устройствами, оставшихся в, возможно, некорректном состоянии. Состояние потока обычно используется только ОС и напрямую недоступно пользовательским приложениям, количество и семантика состояний также сильно зависят от архитектуры ОС. Однако можно определить набор из состояний, а также переходов между ними которые характерны практически для всех современных ОС, включая и FX-RTOS. Диаграмма переходов между описанными состояниями потока показана на Рис. 45.

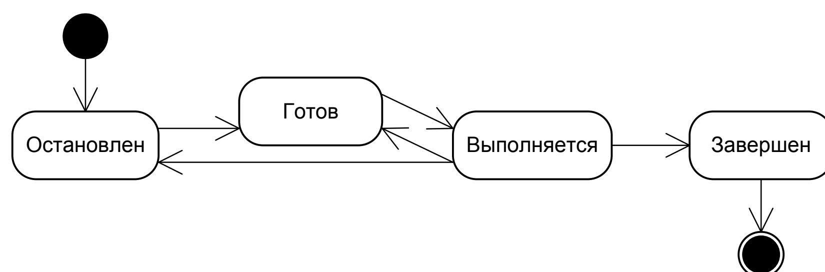

Рис. 45 Состояния потока

5.3.3 Интерфейс потоков

Объект, соответствующий потоку, называется TCB (Thread control block) и содержит аппаратный контекст и прочие атрибуты потока. В данной главе для TCB используется тип fx_thread_t. Этот объект обычно не имеет доступных пользователю полей, он используется только как аргумент для функций ОС.

Во-первых, у потока, как и у любого объекта должен быть конструктор и деструктор. Конструктор принимает в качестве параметров объект, который нужно инициализировать, потоковую функцию, аргумент этой функции, приоритет либо его аналог, позволяющий отличать одни потоки от других, стек потока и различные флаги, например указание, создавать ли поток в активном или приостановленном состоянии.

```
int fx_thread_init(fx_thread_t* thread,
                   void (*func)(void*),
                   void* arg,
                   unsigned int priority,
                   void* stack,
                   size_t stack_sz,
                   bool create_suspended);
```

Деструктор принимает только объект, который нужно удалить. Важно отметить, что деструктор не освобождает память, используемую объектом и должен всегда вызываться для объекта, находящегося в корректном состоянии, то есть инициализированного ранее. На эту функцию распространяется множество ограничений. Например, вызывать деструктор можно только для потоков, выполнение которых полностью завершилось. Ответственность за правильный контекст вызова деструктора возлагается на вызывающий код. Каким образом ядро ОС может в принципе гарантировать выполнение этих условий будет обсуждаться далее.

```
int fx_thread_deinit(fx_thread_t* thread);
```

Поток может быть завершен, либо в результате собственных действий, либо принудительно по инициативе другого потока.

```
int fx_thread_terminate(fx_thread_t* thread);
void fx_thread_exit(void);
```

В качестве аргумента функции fx_thread_terminate можно указывать и текущий поток, функция fx_thread_exit предоставляется для удобства, чтобы избавить вызывающий код от необходимости выяснять указатель на объект текущего потока, кроме того, далее будет показано, что у этих двух функций немного отличается логика работы из-за того, что первая является асинхронной, а вторая синхронной.

Получение результата выполнения потока, а также ожидание его завершения выполняется с помощью функции fx_thread_join.

```
int fx_thread_join(fx_thread_t* thread);
```

После того, как эта функция успешно завершилась, ядро ОС должно гарантировать, что выполнение потока полностью завершено и для него можно вызывать деструктор. Пользователь, со своей стороны, должен гарантировать, что после того, как завершилась функция fx_thread_join никакие другие функции кроме deinit над данным объектом потока не будут вызваны, а деструктор может быть вызван только единожды.
Поток может приостанавливать свое выполнение, а также может быть созданным в приостановленном состоянии. Для переключения между этими состояниями предусмотрены две функции:

```
int fx_thread_suspend(void);
int fx_thread_resume(fx_thread_t* thread);
```

Хотя некоторые ОС предоставляют также асинхронный вариант приостановки потока, когда он может быть приостановлен из другого потока. Использование такого подхода в прикладных программах недопустимо из-за того, что точка, в которой произойдет остановка потока неизвестна, следовательно, использовать эту функцию для синхронизации можно только тогда, когда ее вызывает тот поток, который приостанавливает самого себя, а следовательно, становится известно точное место, где эта приостановка произошла.
В целях предотвращения неправильного использования этой функции, а также чтобы упростить ядро, в текущей реализации рассматривается вариант fx_thread_suspend без аргументов, всегда происходит приостановка вызвавшего потока.
Напротив, функция возобновления выполнения может использоваться только асинхронно, потому что приостановленный поток лишен каких бы то ни было возможностей ее вызвать. С помощью этой функции может быть продолжено выполнение потока, который явно приостановил сам себя, а также запуск выполнения потоков, которые были созданы в приостановленном состоянии.
Естественно, поток должен иметь возможность себя идентифицировать, эту функцию выполняет соответствующий метод fx_thread_self, который должен вернуть указатель на соответствующий текущему потоку TCB.

```
fx_thread_t* fx_thread_self(void);
```

С потоком могут быть ассоциированы различные параметры, такие как приоритет по отношению к другим потокам, привязка потока к процессору в многопроцессорной системе и т.д. Поскольку количество таких параметров может зависеть от конкретной конфигурции, целесообразно сделать изменение параметров через пару методов, которые принимают идентификатор параметра и принимают/возвращают сам параметр который в данном варианте интерфейса может являться (и обычно всегда является) целым числом передаваемым по значению.

```
int fx_thread_get_params(fx_thread_t* thread, unsigned int type, unsigned int* value);
int fx_thread_set_params(fx_thread_t* thread, unsigned int type, unsigned int value);
```

Наконец, широко используемая в прикладном ПО функция приостановки потока на

заданное количество условных временных единиц (в зависимости от реализации могут использоваться как реальные единицы измерения времени, такие как, например, миллисекунды, так и интервал между прерываниями системного таймера, называемый тиком, или кратные им величины).

```
int fx_thread_sleep(uint32_t ticks);
```

Приведенный набор функций является базовым практически для любой ОС. На их основе можно создавать многопоточные приложения в том числе и для многопроцессорных систем.

Реализация приложений с использованием функций ядра похоже на рассмотренную ранее реализацию с использованием интерфейсов HAL. В процессе запуска системы вызывается пользовательская функция, которая должна создать потоки и прочие объекты, далее управление передается этим потокам, которые также могут порождать другие потоки и так далее.

Создание потока выглядит следующим образом:

```
/* Статическое выделение памяти для потока и его стека. */
static fx_thread_t my_thread;
static unsigned int stack[STACK_SZ];

/* Потоковая функция. */
static my_thread_func(void* arg) {
  /* Код потока. */
}

/* Функция инициализации системы. */
void app_init(void) {

  /* Инициализация потока. */
  int error = fx_thread_init(
       &my_thread, my_thread_func, NULL, 10, stack, sizeof(stack), false);

  if(!error) {
    /* Обработка ошибок. */
  }
}
```

Некоторые языки, такие как Erlang или Go определяют также модель многозадачности на уровне синтаксиса самого языка, то есть библиотеки поддержки (runtime libraries) выполняют также часть функций, свойственных ОС. В широко используемых для разработки встроенного ПО языках C и C++ поддержка потоков и стандартных интерфейсов работы с ними появилась только с принятием стандарта 2011 года [5]. До принятия этого стандарта для разработки переносимых приложений использовалось несколько независимых стандартов, определявших интерфейс для реализации многопоточных программ. Среди этих интерфейсов POSIX, OSEK, uITRON и некоторые другие. Поскольку ни один из них не считался общепринятым стандартом, а также из-за того, что каждый из приведенных интерфейсов обладает определенными достоинствами и недостатками, производители встроенных ОС в основном шли по пути создания

собственного интерфейса для потоков и примитивов синхронизации, а затем написания программных оберток для реализации поверх этого интерфейса одного или нескольких стандартных. С принятием стандарта C11 интерфейс для потоков появился и в C, но он по-прежнему уступает по своим возможностям перечисленным интерфейсам.
Поскольку история FX-RTOS началась до принятия стандарта C11, интерфейс потоков был спроектирован проприетарным, как и у большинства подобных ОСРВ.

5.3.4 Контекст функций ядра

Взаимодействие ядра и приложения может быть организовано различными способами. С точки зрения приложения и для удобства написания последнего, вызовы ядра должны выглядеть как вызовы обычных функций, реализованных в библиотеке ядра, дальнейшее же может меняться от одной реализации другой. По соображениям производительности и снижения накладных расходов, наиболее распространен вариант, когда код ядра выполняется тем же потоком, который выполнял и пользовательский код. Данный подход является довольно универсальным и применяется как в небольших ОС для встраиваемых систем так и в "больших" ОС, применяемых в настольных компьютерах и серверах. Выполнение кода ядра внутри пользовательского потока обеспечивает наилучшую производительность, но вместе с тем требует предельно внимательного отношения к синхронизации внутри ядра, поскольку его код интенсивно используется множеством пользовательских потоков и одна и та же функция может находиться в разных стадиях исполнения одновременно даже в однопроцессорной системе. Например, один из потоков может попытаться создать еще один поток, но в момент его создания может произойти вытеснение и тот поток, который начал выполняться следующим, также вызвал функцию создания потока. Таким образом, функция создания потока выполняется одновременно двумя независимыми потоками. В этой связи возникают сложности с обеспечением атомарности: ядро должно иметь возможность предотвращать вытеснение потока в те моменты когда он выполняет код ядра, производящий манипуляции с ядерными структурами данных, которые могут быть разделяемыми и доступ к которым должен быть атомарным. Переключение потоков в такой системе происходят внутри функций ядра, которые вызываются пользовательскими потоками, как показано на Рис. 46.

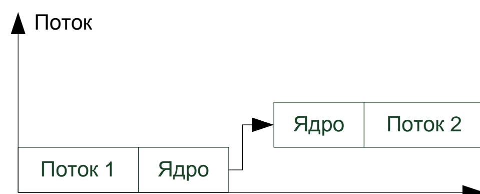

Рис. 46 Выполнение кода ядра в пользовательских потоках

Один из возможных альтернативных подходов заключается в том, что ядро имеет свой выделенный поток, в котором выполняются системные функции, а пользовательские потоки обращаются к нему, передавая параметры через общую память. Переключение контекста в этом случае возникает при возврате из функций ядра, как показано на Рис. 47.

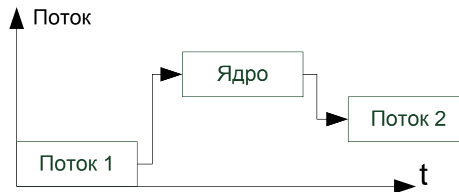

Рис. 47 Выполнение кода ядра в выделенном потоке

Этот метод упрощает ядро, которое становится невытесняемым и код которого может выполняться только одним потоком в каждый момент времени, следовательно, практически не нуждается в синхронизации, синхронизировать нужно только взаимодействие с прерываниями. Эта простота дается не бесплатно, и платить за нее приходится снижением производительности. Кроме того, такой подход оказывается довольно бессмысленным в случае многопроцессорных систем: использование только одного потока ядра приводит к тому что пользовательские потоки выстраиваются в очередь к единственному процессору, тому, который в данный момент исполняет поток ядра. Если же запустить по одному потоку на процессор, то опять возникает необходимость синхронизации доступа к общим структурам данных, то есть в том, чего пытались избежать, выбирая такую архитектуру. Ввиду большей универсальности и производительности целесообразно выбрать первый вариант: то есть позволить пользовательским потокам выполнять код ядра непосредственно.

Способ вызова функций ядра влияет также и на время реакции системы. В случае выполнения кода ядра в контексте пользовательного потока, зачастую синхронизация нужна только на небольших участках кода, манипулирующих глобальными данными системы. Кроме того, код ядра может содержать циклы, запрещая прерывания не более чем на одну итерацию, позволяя коду ядра быть вытесненным. Это позволяет добиться гарантий, что реакция на входящие прерывания будет находиться в ограниченном временном интервале. Противоположный подход с выделенным потоком ядра лишен этих достоинств, поскольку ядро невытесняемо и должно отработать полностью до завершения вызова, циклы в его коде неизбежно приводят к непредсказуемым заранее задержкам реакции, что, конечно, является недопустимым для ОСРВ.

Некоторые реализации предусматривают запрет вытеснения кода ядра, то есть упрощение синхронизации достигается за счет того, что хотя код ядра и может выполняться разным потоками, фактически, в каждый момент времени (в однопроцессорной системе) такой поток может быть только один. Данный подход помимо трудностей с многопроцессорными системами имеет также ряд других негативных побочных эффектов. Длина критических секций, таких как время работы с запрещенными прерываниями, должны быть ограничены. Запрет вытеснения ядра налагает еще более жесткое требование: все функции ядра должны работать за фиксированное время, что на практике не всегда достижимо, поэтому ОСРВ должна обеспечивать возможность вытеснения любого кода в том числе и самого ядра.

5.3.5 Планирование

Если в состоянии готовности к выполнению находится более одного потока, а физический процессор только один, нужно каким-то образом выбрать поток, который следует выполнять следующим, это решение принимается специальным компонентом ОС, называемым **планировщиком**.

Для планирования в реальном времени было разработано множество алгоритмов. Данный раздел носит обзорный характер, так как подробное изложение теоретического обоснования каждого из них выходит за рамки данной книги. Подробную информацию о различных алгоритмах можно найти в [6] и [7].

Понятие планировщика в ядре операционной системы включает в себя два аспекта: во-первых планировщик понимается как некоторый алгоритм, который используется для выбора из нескольких готовых к выполнению потоков того, который будет исполняться следующим; во-вторых, планировщик также должен решать задачу синхронизации доступа к хранилищу готовых к выполнению потоков и предоставлять интерфейсы для изменения различных параметров потоков, таких как приоритеты. Потоки первоначально обращаются к слою синхронизации, который управляет контейнером потоков, как показано на Рис. 48.

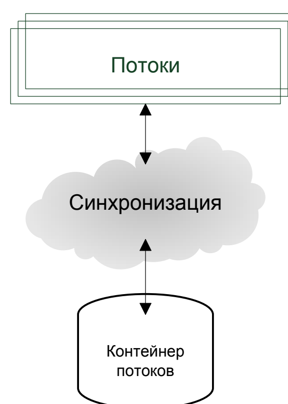

Рис. 48 Механизм обращения к планировщику.

Эти две задачи ортогональны и слабо связаны друг с другом, поэтому целесообразно разделить их на два независимых компонента которые будут рассматриваться отдельно.

Алгоритмы планирования разделяются на несколько классов:

- Использующие или не использующие расписание
- Статические и динамические
- С вытеснением и без вытеснения

Может также использоваться дополнительная классификация, например по признаку использования или не использования эвристик, хотя эвристические алгоритмы не получили широкого распространения во встроенных системах реального времени из-за вносимой в процесс планирования неопределенности.

Алгоритмы использующие расписание предполагают существование описания распределения времени между потоками, которое создается на этапе сборки системы и в дальнейшем не изменяется. То есть потоки всегда выполняются в одной и той же последовательности, что роднит этот подход с различными вариациями циклического

планировщика. Подобный подход часто используется как часть двухуровневой схемы: существует несколько групп потоков, группы получают время по расписанию, тогда как внутри группы используются другие алгоритмы без расписания. Расписание используется как способ совместить в рамках одной системы несколько независимых задач, с гарантированным распределением времени между ними. Если на первый план выходит скорость реакции, то использование расписания страдает от тех же недостатков что и различные циклические алгоритмы.

Для эффективного планирования у потоков должны быть некоторые признаки, которые позволяют различать их между собой. Статические и динамические алгоритмы различаются по типу этого параметра, задается ли он один раз при создании потока (статическое планирование), либо может меняться в процессе работы системы (динамическое планирование). Для иллюстрации динамического подхода можно привети такие алгоритмы как EDF (Earliest deadline first) и SPN (Shortest period next) [8]. Первый использует в качестве параметра предельное время, до наступления которого поток должен получить управление, второй - период, число, указывающее на ожидаемое время работы потока. В качестве примера статического параметра потока можно привести приоритет.

Хотя подход, применяющийся в алгоритмах типа EDF является оптимальным с теоретической точки зрения, на практике довольно сложно определить как ожидаемый период, так и предельное время, которое может варьироваться в широких предалах в зависимости от того, чем поток занят в данный момент. Предельный срок выполнения потока является динамической величиной требующей постоянного обновления по мере работы системы, кроме того, довольно трудно реализовать детерминированные алгоритмы поиска подходящего потока. Несмотря на то, что ОСРВ, использующие динамические алгоритмы планирования существуют, в целом этот подход нельзя назвать популярным. Статические алгоритмы основанные на приоритетности потоков, позволяет добиться большей предсказуемости работы системы в целом, а также содержат меньше накладных расходов и используют более простые структуры данных и алгоритмы.

В принципе возможно использование динамического подхода и для приоритетного планирования. Планировщик с динамическими приоритетами использует повышение либо понижение приоритетов потоков в зависимоти от того, как долго поток находился в состоянии ожидания, а также от нагрузки системы. Такие алгоритмы хорошо себя показали в интерактивных системах, когда динамически повышается приоритет тех потоков, которые в данный момент взаимодействуют с пользователем. Но из-за вносимой неопределенности, динамическое изменение приоритета в системах жесткого реального времени не используется. Планировщик на основе статических приоритетов построен проще: приоритет задается потоку один раз при его создании и больше не меняется. За счет простоты, эффективности и отсутствия накладных расходов именно этот класс алгоритмов планирования получил наибольшее распространение во встроенных ОСРВ.

Алгоритмы с вытеснением предполагают что поток может быть прерван в любое время, как в результате его собственных действий, так и, например, в том случае, если обработчик прерывания переводит в состояние готовности поток с более высоким приоритетом чем у текущего потока. Алгоритмы без вытеснения (называемые также

кооперативными) предполагают согласованное выполнение потоков, когда планирование выполняется не в любой момент, а только в определенных точках в коде, например при завершении потока или в том случае когда он сам решает отдать время другим потокам. Основной теоретической работой, обосновывающей использование статического приоритетного планирования для периодических задач является [9], которая утверждает, что если нагрузка на процессор (сумма отношения времени работы каждого потока к его периоду) не превышает N(2^(1/N) - 1), где N - количество потоков, то существует такое распределение приоритетов и процессорного времени между ними, то все потоки отработают в пределах своего периода.

Вытесняющие алгоритмы обеспечивают наилучшее время реакции, так как наиболее приоритетный поток, готовый к выполнению, всегда получает процессорное время после некоторой фиксированной задержки, однако при использовании большого числа потоков с разными приоритетами возрастают накладные расходы на частые переключения контекста. На Рис. 49 показана ситуация, когда низкоприоритетный поток, обозначенный как 0 должен перевести в активное состояние три других потока, у каждого из которых приоритет выше, чем у активирующего (обозначены как H, M и L, от high, medium и low). Поток 0 активирует поток H, который выполняет некоторую работу и блокируется, далее управление переходит обратно в поток 0, который активирует следующий поток и т.д. При этом происходит 6 переключений контекста. Гораздо более эффективно было бы каким-то образом предотвратить вытеснение на каждом этапе, чтобы активация потоков была выполнена сразу для всех трех потоков, а затем отработал бы каждый из них, как показано на Рис. 50, в этом случае происходит только 4 переключения контекста.

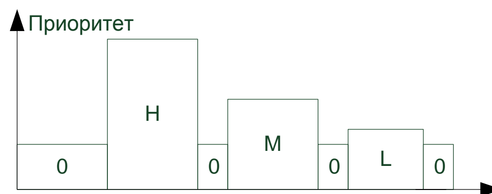

Рис. 49 Вытесняющая активация высокоприоритетных потоков низкоприоритетным

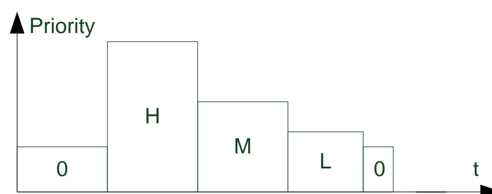

Рис. 50 Невытесняющая активация высокоприоритетных потоков низкоприоритетным

Кроме того, такой подход упрощает синхронизацию, потому что потоки выполняющиеся кооперативно, никогда не вытесняют друг друга, а значит могут не беспокоиться о синхронизации доступа к разделяемым данным. Это перекликается с идеями программирования в стиле "конечного автомата" с известными переходами и на первый взгляд сводит на нет все преимущества вытесняющей многозадачности. Наиболее эффективное решение заключалось бы в том, чтобы потоки, выполняющие какие-то

задачи "в фоне", которым не важно время реакции на события, выполнялись бы кооперативным образом, а другие, которые должны обрабатывать внешние события - вытесняющим. Тогда низкоприоритетные потоки не тратили бы процессорное время на слишком частые переключения, и в то же время не страдала бы скорость реакции системы.

Наиболее широко применяющийся подход предполагает сосуществование вытесняющего и кооперативного планирования в таком виде, что допускается существование нескольких потоков с одинаковым приоритетом и внутри этой группы используется кооперативное планирование. Начав выполняться, поток может быть вытеснен только более приоритетными потоками либо будет продолжаться вплоть до своего завершения, несмотря на наличие других ожидающих потоков с тем же приоритетом.

Минус такого подхода в том, что процессорное время между потоками с одним приоритетом может распределяться непропорционально. Кроме того, порядок, в котором потоки получают управление довольно плохо предсказуем, так как он может зависеть от того, в каком порядке потоки были остановлены.

В некоторых случаях, нужно обеспечить равномерное распределение процессорного времени между группой потоков. Ситуация, когда один поток блокирует все остальные потоки того же приоритета на неопределенное время (пока он сам не решит отдать процессор) не всегда приемлема, поэтому вводится ограничение и верхний предел времени, в течении которого поток может занимать процессор. Это время называется **квантом**. В течении кванта поток может выполняться непрерывно. Если в течении этого времени поток сам уступает процессор другим потокам, то все работает так же, как и в случае простого кооперативного планирования. Если же этого не происходит и есть другие потоки с таким же приоритетом, то происходит принудительное переключение. Потоки с более высоким приоритетом вытесняют менее приоритетные безусловно, как и ранее, однако теперь поток не может захватив процессор выполняться неограниченно долго, если есть другие потоки с таким же приоритетом, после того как поток исрасходовал свой квант, выполнение продолжится в другом потоке с таким же приоритетом. Такой тип планирования называется round-robin.

Описанная модель вытесняющего планирования с возможностью кооперативного планирования внутри группы потоков и с настраиваемыми квантами является де-факто индустриальным стандартом и используется в абсолютном большинстве ОСРВ и ее достаточно для большинства возникающих прикладных задач.

Подводя итог, можно сказать, что в дальнейших главах будет рассматриваться ОСРВ потоки которой имеют фиксированный приоритет задаваемый при создании потока. При переходе более приоритетного потока в состояние готовности, выполняемый поток немедленно вытесняется. Возможно существование группы потоков в рамках одного приоритета, в этой группе осуществляется кооперативное планирование: следующий поток начнет выполняться только после того, как текущий заблокируется по своей инициативе.

5.3.6 Блокирование вытеснения

Несмотря на то, что процессор предоставляет воможности запрета прерываний для

реализации синхронизационных механизмов внутри ядра, необходимость в этом возникает довольно редко. В большинстве случаев, для получения необходимых гарантий требуется только предотвратить вытеснение текущего потока до того момента. пока он не закончит манипуляции с некоторыми глобальными данными. Прерывания при этом остаются разрешенными и снижается вероятность того, что какие-то из них будут утеряны. Для этой цели, большинством операционных систем, предоставляется специальный механизм, позволяющий временно заблокировать выполнение планировщика и предотвратить вытеснение. Кроме того, такой механизм необходим и для реализации самой ОС. По аналогии с запретом прерываний, этот механизм выглядит как пара функций вида запретить/разрешить. В зависимости от уровня доверия к пользовательскому коду использование этого механзма может быть допустимо и для приложений, в частности, в небольших ОСРВ предназначенных для использования в микрокнтроллерах, где отсутствует явное разделение на ОС и приложение, блокирование вытеснения может использоваться для реализации небольших критических секций и синхронизации доступа к разделяемым данным. Хотя использование такого подхода выглядит просто и не требует никаких дополнительных объектов, использовать его нужно с большой осторожностью, потому что на то время пока планировщик заблокирован, вся "многозадачность" ядра фактически отключается. Если период времени, на который блокируется вытеснение не является фиксированным зависит от каких-то внешних параметров и состояния системы, фактически ОС перестает удовлетворять критериям ОСРВ, так как время ее реакции становится непредсказуемым.

5.3.7 Ожидание

После того, как поток решает приостановить собственное выполнение, например, он может ожидать готовности устройства или выполнения некоторого условия, возобновить его выполнение можно с помощью вызова fx_thread_resume. Однако использование этой функции предполагает, что вызывающему коду известен объект потока, выполнение которого следует возобновить, также нужно быть уверенным, что ожидающий поток существует и находится в корректном состоянии (не завершен). Кроме того, события могут ожидать несколько потоков, и необходимо как-то учитывать их все. Для решения этих проблем была разработана концепция примитивов синхронизации.

Примитив синхронизации - это объект (и связанные с ним методы), который служит посредником между ожидающими потоками и потоками которые должны возобновить выполнение ожидающих, как показано на Рис. 51.

Рис. 51 Примитив синхронизации.

Задача в таком случае упрощается, потому что теперь потокам нет нужды знать друг о друге: если потоку нужно дождаться чего-либо, он должен использовать примитив синхронизации, ассоциированный с событием либо условием. Сигнализирующие о

наступлении события потоки также должны работать с примитивом, который берет на себя работу по учету ожидающих потоков и их пробуждению.

Примитив может также иметь внутреннее состояние, указывающее на то, стоит ли блокироваться потокам, которые пытаются ждать данного примитива. **Активацией примитива** называется вызов его метода, приводящий к возобновлению выполнения какого-либо из ожидающих примитив потоков.

Логика работы примитивов может быть самой различной, некоторые из них нотифицируют только одного ожидателя на каждую активацию, некоторые нотифицируют всех. Подробно примитивы и сценарии их использования будут рассматриваться в разделе посвященном межпотоковой синхронизации.

Как и в случае с планировщиком, если ожидающих потоков более одного, а логика примитива допускает пробуждение только одного потока в результате активации, возникает та же проблема: выполнение какого потока следует возобновить. Реализация этого механизма различается в разных ОС, но наибольшей популярностью пользуются два метода: FIFO и метод, основанный на приоритетах. В первом случае, поток который раньше начал ожидание, будет возобновлен первым. Это наиболее простой и эффективый метод, хотя в некоторых случаях, если ожидают несколько разноприоритетных потоков, было бы более предпочтительно выбрать обладающий наибольшим приоритетом. С другой стороны, это может привести к проблеме известной как **голодание**: если примитив ожидают два разноприоритетных потока, и активация примитива происходит с такой частотой, что активированный поток успевает закончить работу и опять встать в очередь ожидания до следующей активации, то при таких условиях, второй поток вообще никогда не получит процессорное время. Более простой алгоритм FIFO лишен такого недостатка, поэтому зачастую сложно сделать однозначный выбор в пользу того или иного алгоритма, некоторые ОСРВ предоставляют возможности для задания политики пробуждения потоков.

Для возможности реализации различных примитивов удобно не определять набор функций ожидания каждого из них, а снабдить интерфейс потоков некоторой универсальной функцией ожидания, на базе которой могут быть построены различные примитивы, она имеет следующий прототип:

```
int fx_thread_object_timedwait(
      fx_sync_waitable_t* object, void* attribute, uint32_t timeout);
```

Примитив синхронизации представлен абстрактным "waitable"-объектом. Атрибут ожидания это некоторая дополнительная информация, которая ассоциирована с операцией ожидания, например, ожидание прихода сообщения в очередь может предполагать существование буфера, в который эта информация должна быть помещена. В большинстве случаев этот параметр не используется, но может пригодиться для реализации синхронизации с передачей данных, подробно эта тема будет обсуждаться в главе посвященной коммуникации между потоками.

Для предотвращения бесконечного ожидания и контроля за максимальным временем, которое поток может находиться в заблокированном состоянии, обычно функции

ожидания имеют в качестве параметра также таймаут. Если он указан, и до его истечения примитив синхронизации не переходит в активное состояние, выполнение потока возобновляется и из функции ожидания возвращется соответствующий код ошибки.
В качестве простейшего примитива синхронизации можно привести объект "событие". Этот объект имеет булево значение в качестве внутреннего состояния и (помимо очевидных конструктора и деструктора) два метода: set и reset. Состояние объекта указывает произошло событие или нет, методы используются для изменения состояния. Когда поток вызывает функцию ожидания с объектом "событие", происходит анализ состояния объекта, если объект активен (имеет значение true), поток продолжает выполнение без ожидания, если неактивен (значение false) - поток блокируется. Установка события с помощью функции set каким-либо другим потоком либо обработчиком прерывания, приводит к тому что все ожидающие потоки (если таковые были) разблокируются и продолжают выполнение. Состояние объекта может изменяться только в результате вызовов двух указанных методов, поэтому если нужно ждать следующего события, объект необходимо предварительно сбросить в неактивное состояние.

5.3.8 Программные таймеры

Многие встраиваемые системы используют разного рода периодичность. В простейших случаях она может быть достигнута с использованием потоков, которые могут блокироваться на заданное время и затем вызывать определенную функцию. Если же требуется отслеживать множество таких функций, каждая из которых может иметь свой период, то реализация такого функционала в прикладном коде слишком усложняет последний. Ко всему прочему, сама реализация потоков зачастую требует периодических вызовов функций для реализации fx_thread_sleep. Для этой цели используются объекты называемые **программными таймерами**.
Программный таймер это специальный системный объект, обеспечивающий однократный либо периодический вызов указанной функции с заданными временными интервалами. Интерфейс таймеров довольно прост и включает в себя лишь несколько фунций:

```
uint32_t fx_timer_get_tick_count(void);

int fx_timer_init(fx_timer_t* timer, int (*func)(void*), void* arg);
int fx_timer_deinit(fx_timer_t* timer);
int fx_timer_set_abs(fx_timer_t* timer, uint32_t delay, uint32_t period);
int fx_timer_cancel(fx_timer_t* timer);
```

Помимо очевидных функций конструктора/деструктора объекта и общесистемной функции запроса текущего значения счетчика системных тиков он содержит лишь две основные функции: установка таймера на заданное время и отмена запущенного таймера. В качестве времени установки используется абсолютное значение счетчика тиков, при котором должна произойти активация. Установка таймера на время относительно времени вызова функции легко реализуется с помощью считывания счетчика тиков и прибавления к нему смещения с последующей установкой таймера на

полученное абсолютное время.
Несмотря на простой интерфейс, таймер является одним из наиболее сложно реализуемых системных объектов. Дело в том, что в отличие от таких объектов как потоки и примитивы синхронизации, состояние внутренних структур данных содержащих активные таймеры зависит от времени, отсчет которого идет независимо от действий прикладных программ и поэтому последовательный запуск одних и тех же программ может приводить к различным результатам из-за естественных сдвигов времени в ту или иную сторону обусловленных аппаратурой. Кроме того, по мере обновления счетчика системных тиков необходимо осуществлять поиск таймеров, которые должны сработать. Использование любой классической структуры данных, такой как дерево, не сможет обеспечить поиск за фиксированное время, что, вместе с непредсказуемостью состояния этих структур данных в каждый момент времени, приводит к потере предсказуемости поведения системы. В ОСРВ реализация таймеров обычно является самой сложной частью системы с алгоритмической точки зрения.

5.4 Структура системы

После обсуждения всех необходимых интерфейсов можно приступить к проектированию структуры ядра. В первом приближении структура системы будет выглядеть следующим образом: поверх HAL находятся два модуля, обрабатывающие соответствующие прерывания, которые асинхронно вызываются HAL - программных и аппаратных прерываний соответственно. Обработчик программных прерываний реализован внутри модуля реализующего потоки и выступает планировщиком. Активация потоками других потоков приводит к установке запроса на программное прерывание, в обработчике которого выполняется планировщик. Если обработчик прерывания активирует примитив синхронизации, то после выхода из всех вложенных потоков также произойдет вызов планировщика с возможным последующим переключением контекста.
Опционально возможно присутствие модуля реализующего таймеры, который также является обработчиком специального прерывания. Описанная структура системы изображена на Рис. 52.

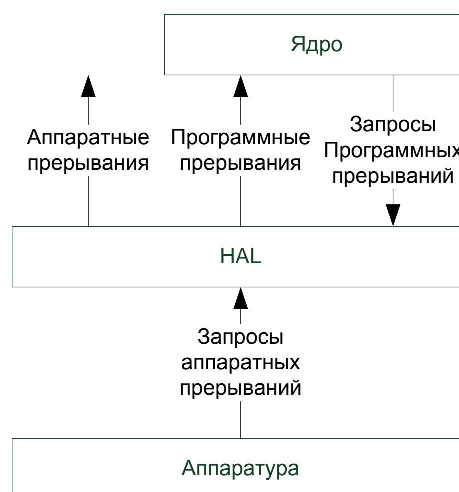

Рис. 52 Структура ядра ОС

Все перечисленные модули являются вполне независимыми: возможно как разработка системы в которой будут использоваться только аппаратные прерывания, так и системы, в которой такие прерывания будут полностью отсутстововать. Таймеры, хотя и необходимы для реализации модуля потоков, могут быть реализованы в виде модуля-заглушки, в случае, если связанная со временем функциональность от потоков не требуется (не используются функции fx_thread_sleep и таймауты в функции ожидания).

В общем же случае встраиваемые приложения, как правило, пользуются всеми перечисленными модулями, поэтому далее будет рассматриваться система содержащая все эти компоненты.

Если таймеры и прерывания выступают только пассивными обработчиками событий, источником которых является HAL, модуль, реализующий потоки, является активной сущностью, которая, помимо обработки, может также инициировать программные прерывания внутри которых работает планировщик. Хотя последний кажется неотъемлемой частью реализации потоков, задача планировщика имеет довольно отдаленное отношение к интерфейсу потоков. Можно себе представить систему, в которой в качестве активных выполняемых сущностей будут использоваться не потоки, а, например, акторы, которые также могут иметь приоритеты и использовать те же правила, что и потоки. Это приводит к идее о том, что планировщик является отдельным модулем, который хотелось бы использовать повторно в различных конфигурациях системы.

Для отделения реализации планировщика от реализации потоков можно выделить внутри объекта потока поля, используемые планировщиком в отдельный объект, представляющий абстрактный "элемент планировщика". Все функции планировщика при этом работают не с объектом потока, а с упомянутым абстрактным объектом.

Абстрактный объект планировщика включается в качестве поля в объект потока, а функции планировщика, вызываемые внутри модуля потоков используют в качестве аргумента не весь объект потока, а только относящуюся к планировщику часть.

Структура модуля потоков при этом приобретает вид, показанный на Рис. 53.

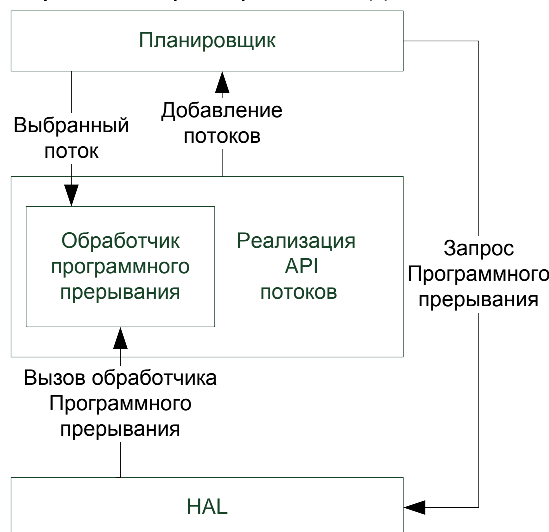

Рис. 53 Структура модуля потоков

В случае изменения состояния потоков, которые происходит как вызовы функций планировщика внутри функций интерфейса потоков, происходят запросы программных

прерываний, которые обслуживаются обработчиком, входящим в состав модуля потоков.
Работу системы можно представить циклически: потоки, с помощью использования примитивов синхронизации или прямыми вызовами (fx_thread_suspend / fx_thread_resume), переводят друг друга в состояние готовности к выполнению, планировщик переключает контекст на другие потоки и так далее. Без использования аппаратных прерываний все выглядит похоже на циклический планировщик, рассматривавшийся ранее.

5.5 Взаимодействие с прерываниями

5.5.1 Латентность

Предсказуемая реакция на внешние события является основной задачей ОСРВ, поэтому не менее важный вопрос, чем планирование - организация взаимодействия между потоками и обработчиками прерываний. Обработка прерывания включает в себя несколько этапов, показанных на Рис. 54.

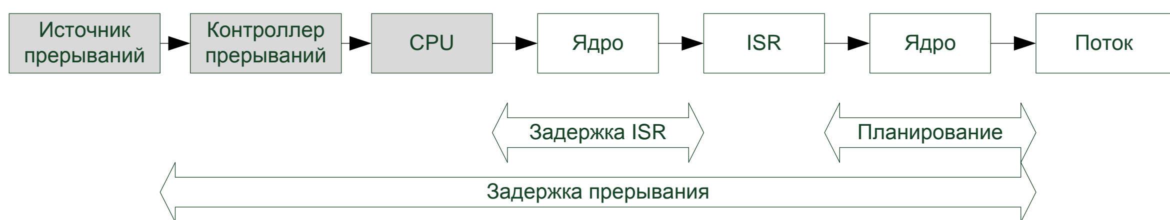

Рис. 54 Этапы обработки прерывания

Вначале источник прерывания устанавливает запрос с помощью контроллера, затем контроллер передает этот запрос процессору. Задержка, вносимая этими компонентами имеет чисто аппаратную природу, но, тем не менее должна учитываться при проектировании. После выполнения процессором действий по первичной обработке прерываний, поиска обработчика, управление получает ОС, а точнее HAL, который после выполнения необходимых действий передает управление ОС. ОС должна найти подходящий внешний (установленный приложением) обработчик (называемый также ISR, interrupt service routine) и передать управление ему, эта задержка, между вызовом первичного обработчика прерывания на самом нижнем программном уровне и вызовом обработчика, установленного пользователем на верхнем уровне API также обычно является константой и называется задержкой вызова обработчика (ISR latency).
Получив управление, обработчик обычно производит необходимые действия с устройством и активирует поток, обрабатывающий соответствующие прерывания, так как возможности обработки прямо в обработчике существенно ограничены.
В описании интерфейса многих ОС упоминается тот факт, что в контексте обработчика прерывания нельзя использовать блокирующие примитивы синхронизации (то есть нельзя использовать ожидание) но редко объясняется,почему это так. С одной стороны, есть объяснение, что обработчики могут быть вложенными, поэтому если заблокировать один, то блокируется вся цепочка. Это одна из причин. Теоретически, можно представить обработчики прерываний, в которых можно использовать ожидание, что для этого может потребоваться? Во-первых отдельный стек у каждого обработчика, который позволял бы

сохранить там необходимый контекст. Во-вторых, обработчик прерывания должен быть некоторой сущностью, которая может находиться в очереди ожидания примитивов синхронизации. Далее, обработчики прерывания должны планироваться, если несколько обработчиков ждут примитива, то когда он будет освобожден, они должны быть возобновлены. Рассуждая в таком ключе дальше, можно прийти к выводу, что обработчикив прерывания станут фактически потоками и возникновение прерывания будет активировать такой поток, который может ждать примитивов и делать все то, что может делать и обычный поток. С таким подходом связано довольно много накладных расходов что входит в противоречие с требованием, чтобы обработчик были максимально быстрым. Поэтому применяется оптимизация: обработчики выполняются на едином выделенном стеке и находятся вне контекста которым управляет планировщик.
После выхода из обработчика управление снова получает ОС, которая, в случае, если из обработчика прерываний был активирован высокоприоритетный поток, должна запустить планировщик и переключить контекст на новый поток. Эта задержка называется задержкой планирования (scheduling latency). Таким образом, суммарное время реакции системы является суммой задержек, часть из которых вносит аппаратура, HAL и ядро ОСРВ.

5.5.2 Отложенные процедуры

Фактическое время реакции зависит не только от времени исполнения обработчика и планировщика, но также и от времени выполнения критических участков кода ядра, которые выполняются с запрещенными прерываниями или заблокированным планировщиком. Если, к примеру, какой-либо поток запрещает прерывания на 1 мс, то, очевидно, даже в случае, если вся цепочка описанная выше, способна отработать за 1 мкс, худшее время реакции составит не менее 1 мс. Поскольку запрет планирования довольно часто применяется в ядре ОС для синхронизации доступа к хранилищу потоков: всякая приостановка либо возобновление выполнения потока может потребовать такого запрета. С другой стороны, потоки могут ожидать событий, связанных с аппаратными прерываниями, то есть внутри обработчиков может изменяться состояние примитивов синхронизации, что автоматически влечет за собой и возможность изменения состояния потоков, что, в свою очередь, означает изменение внутренних структур данных планировщика. Иными словами, во время работы планировщика со своими структурами данных, возможно асинхронное изменение этих данных. Естественно, использование общих данных без синхронизации недопустимо и может привести к их разрушению и краху системы. Поэтому можно поставить следующий вопрос: является ли планировщик прерываемым аппаратными прерываниями?
Самый простой подход, который одновременно является наиболее распространенным во встраиваемых ОСРВ - непрерываемый планировщик. Во время работы планировщика все прерывания запрещаются, следовательно, обработчики не могут выполняться в тот момент, когда планировщик еще не закончил свою работу, поэтому последний может полагаться на консистентность собственных структур данных. То же самое касается и прерываний: с точки зрения обработчиков, состояние планировщика изменяется атомарно, поэтому они могут изменять состояние потоков без риска столкнуться с

промежуточным или неконсистентным состоянием хранилища. Естественно, на время работы с планировщиком обработчик также должен запрещать прерывания, чтобы не столкнуться с проблемой вложенных прерываний. Код операционной системы, выполняемый в контексте пользовательского потока, периодически запрещает прерывания для защиты глобальных структур данных, как показано на Рис. 55.

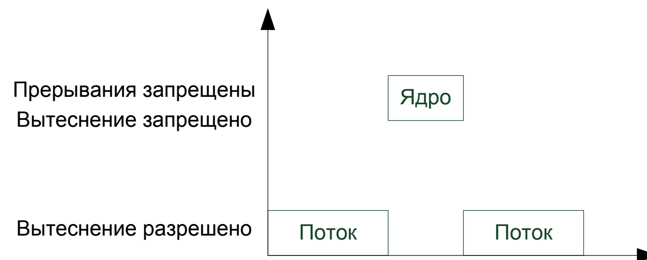

Рис. 55 Критический участок в коде ядра ОС

В большинстве случаев, особенно для ОС предназначенных для систем с ограниченными ресурсами, такой подход вполне приемлем: он обеспечивает наилучшую производительность, а также способствует написанию простого и поддерживаемого кода, как самой операционной системы так и сторонних драйверов и приложений.

На первый взгляд, создается впечатление, что альтернативного варианта не существует, ведь доступ к данным планировщика должен быть атомарным в том числе и для обработчиков прерываний, которые могут происходить в любой момент, если они разрешены. Однако, по мере усложнения планировщика, и, как следствие этого, удлинения блоков кода в ядре, которые должны выполняться с запрещенными прерываниями, увеличивается время реакции системы. Для решения этой проблемы исследователями была разработана концепция **отложенных прерываний**.

Основная идея заключается в том, чтобы разделить обработчик на две части (две независимые функции). Одна из них выполняется в обработчике прерывания как и прежде, эта часть обработчика не имеет доступа ни к каким сервисам ОС, в частности, не может активировать примитивы синхронизации. Всё, что она может сделать - проанализировать регистры устройства, выполнить действия для снятия устройством запроса прерывания и запросить вызов так называемой **отложенной процедуры**[1] (DPC, deferred procedure call), в которой находится вторая часть обработчика (Рис. 56). Отложенная процедура выполняется с разрешенными прерываниями, но с заблокированным планировщиком. В целом это напоминает подход с асинхронными программными прерываниями, которые обсуждались ранее.

[1] В разных ОС вызовы отложенных процедур называются по-разному. Встречаются такие названия как DRQ (deferred request), DSR (deferred service routine), bottom-half, HISR (High-level ISR), DISR (deferred ISR) и прочие. Далее будет использоваться название DPC.

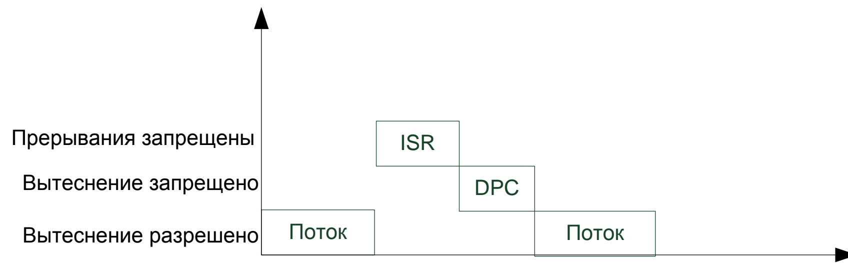

Рис. 56 Обработка прерывания с использованием отложенной процедуры

Это выглядит неоправданным усложнением, ведь обе части все равно должны выполниться, а этот процесс вдобавок дополнительно усложняется очередями и прочими функциями. На самом деле, ключевой момент заключается в том, что очередь отложенных процедур обрабатывается в таком контексте, когда вызовы планировщика запрещены, это позволяет достичь взаимоисключения планировщика и кода отложенных процедур что и обеспечивает синхронизацию между ними. Код же самого планировщика, как и критических секций, требущих его блокировки, отныне может быть прерываемым аппаратными прерываниями (Рис. 57).

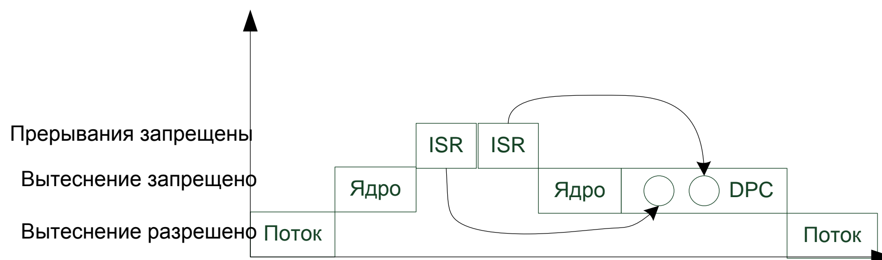

Рис. 57 Активация потока с помощью отложенной процедуры

Аппаратное прерывание пришло в тот момент, когда выполнялся критический участок кода ОС. В очередь была поставлена отложенная процедура. После завершения критического участка, перед разблокированием планировщика анализируется очередь отложенных процедур. Поскольку она непуста, выполняется процедура, которая активирует поток, затем отрабатывает планировщик (возможно, переключащий контекст) и, наконец, управение возвращается в поток.

В случае, если очередь организована с помощью атомарных операций, операционная система может быть построена таким образом, что вообще никогда не будет явно запрещать прерывания. Критические участки кода в такой ОС требуют только запрета вытеснения.

На практике такая возможность не приводит к сколько-нибудь значительным преимуществам, дело в том, что необходимость разделения данных между обработчиками прерываний и планировщиком не исчезает, просто программно реализуется некоторый аналог флага прерываний, функцию которого теперь выполняет флаг запрета планирования и выполнения отложенных процедур. Пользовательские потоки все равно не получат управление раньше, чем отработает вся цепочка из

прерываний и планировщика, плюс дополнительные накладные расходы.
Устоявшегося мнения о том, какой подход лучше, не существует. Каждый разработчик ОС решает этот вопрос по-своему и общее правило такое, что все определяет длина критической секции. Если ОС простая и предназначена для устройств с ограниченными ресурсами, а критические секции внутри ядра относительно невелики, то вполне целесообразно использовать запрет прерываний на время работы планировщика. Если же желательно обеспечить возможность сверхнизкой латентности хотя бы для первичного обработчика прерываний, то лучше использовать подход с очередью отложенных процедур. Это может пригодиться например в том случае, когда обработчики прерываний не пользуются сервисами ОС и потоками, а вся ОС выполняет функции некоторой фоновой задачи.
Другой важный вопрос заключается в том, в какой момент и как именно произойдет обработка очереди. Поскольку функции из этой очереди могут изменять состояние планировщика, сам планировщик следует вызывать после обработки всех таких процедур. Второе требование заключается в том, что все обработчики должны работать в контексте, в котором выполнение планировщика заблокировано (чтобы избежать его вызовов после каждой подобной процедуры). Хорошим кандидатом на эту работу является сам обработчик программного прерывания в котором происходит работа планировщика. Структура модуля потоков при этом приобретает вид, изображенный на Рис. 58.

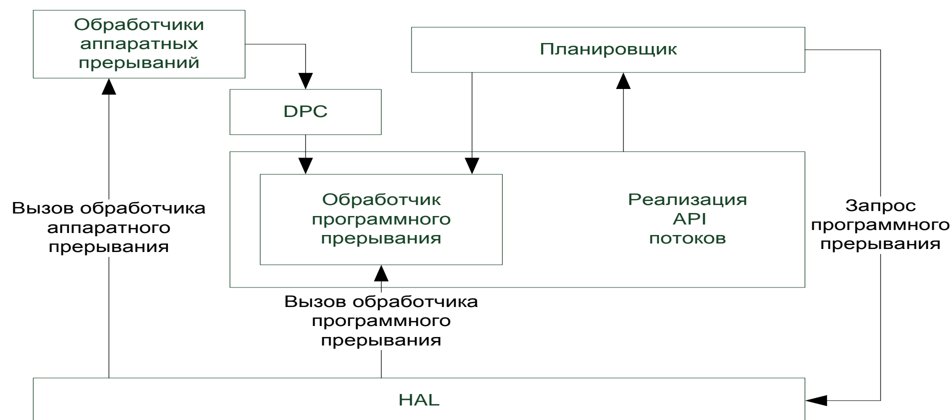

Рис. 58 Структура модуля потоков использующего отложенные процедуры

Всякий раз, когда изменяется состояние планировщика, будь то из обработчиков прерываний либо из критических участков кода внутри ядра, происходит запрос программного прерывания, в котором, перед вызовом планировщика, происходит обработка очереди отложенных процедур.
В отдельных случаях возможна реакция на прерывания полностью реализованная внутри отложенной процедуры. Возможность выстраивать из последних очереди позволяет работать с ними более гибко, чем с прерываниями. В частности, обработка очереди отложенных процедур происходит до вызова планировщика, поэтому время такой реакции будет очень мало. Хотя по смыслу этот подход мало чем отличается от реализации той же логики внутри обработка прерывания, в некоторых случаях такой подход может быть полезным. Например, если устройство время от времени может

генерировать прерывания с очень высокой частотой, драйвер может использовать одновременно несколько отложенных процедур, вплоть до распараллеливания обработки на несколько физических процессоров. Это может существенно увеличить пропускную способность системы, сгладить пики активности и в целом позволить обработать больше прерываний, чем это было бы возможным при реализации обработки внутри, собственно, обработчика прерывания, на время выполнения которого прерывания от данного устройства запрещаются. Некоторую трудность в таком случае представляет реализация очереди отложенных процедур именно как очереди, без введения приоритетов, что вносит непредсказуемость и зависимость времени реакции от других, возможно низкоприоритетных прерываний и отложенных процедур.

В завершение данной темы можно сказать, что некоторые ОСРВ не содержат явного интерфейса отложенных процедур, обработчики прерываний могут использовать примитивы синхронизации обычным образом, а ОС, внутри функций API определяет, если вызов был совершен из обработчика прерывания, то неявным образом этот объект используется для вставки в очередь отложенной процедуры. При этом теряется возможность поставить в очередь несколько отложенных процедур, а также возможность использовать один примитив из нескольких обработчиков, поэтому обычно ядро ОС предоставляет механизм работы с отложенными процедурами предоставляя верхнему уровню принимать решения о том, как их использовать.

5.5.3 Схема синхронизации

Способ, с помощью которого осуществляется взаимодействие между обработчиками прерываний и потоками, называется **архитектурой прерываний**. Это название сложилось исторически, поскольку именно решение вопроса эффективного взаимодействия между обработчиками прерываний и потоками выносилось во главу угла. Однако, как было показано выше, реализация конкретного механизма определяет не только способ взаимодействия прерываний и потоков, но и всю синхронизацию внутри ядра, поэтому, чтобы не акцентировать внимание именно на прерываниях, в данной книге будет использоваться термин "схема синхронизации ядра". Существует два основных подхода, которые обсуждались в предыдущей главе.

Одной из главных задач при разработке ОС, является выбор правильной схемы синхронизации, так как каждая из них имеет свои преимущества и недостатки. После того, как эта схема определена, становится определен протокол взаимодействия между различными сущностями ОС, такими как обработчики аппаратных прерываний, отложенные процедуры, планировщик, пользовательские потоки и так далее. При замене ОС (или при необходимости изменения этой схемы) возникают трудности портирования драйверов устройств с одной схемы на другую.

Ситуация, когда необходимо выбирать между двумя вещами, каждая из которых имеет свою плюсы и минусы довольно часто встречается в разработке встроенного ПО. Выбирая один из методов, разработчик автоматически лишается преимуществ альтернативного подхода. Схема синхронизации определяет не только внутренние механизмы, но также и интерфейсы, которыми пользуются драйверы устройств, поскольку они, как правило, занимаются обработкой прерываний и им, так или иначе, нужно пользоваться

примитивами синхронизации внутри обработчиков. Поэтому, по возможности, хотелось бы сделать настолько универсальное решение, насколько это возможно. К счастью, два описанных метода хотя и кажутся совершенно различными и несовместимыми друг с другом, тем не менее, могут быть скрыты за одним интерфейсом таким образом, что драйвер устройства может быть написан универсальным, то есть, будет способен работать с любой схемой синхронизации. Для этого, правда, может потребоваться перекомпиляция, но в мире встраиваемого ПО это не является проблемой.

Рассмотрим использование двух упомянутых схем с точки зрения изменения SPL. Возможны всего два случая: использование примитивов синхронизации из обработчиков прерываний, когда необходимо из кода с высоким SPL вызвать код с низким SPL, и обратный случай - использование примитивов и планировщика из потоков, в этом случае код с низким SPL должен вызвать код с высоким SPL.

Поскольку код с высоким SPL может прерывать код с низким, из этого следует невозможность синхронного выполнения кода, предназначенного для выполнения с SPL, ниже, чем у текущего кода. Иными словами, при необходимости выполнить код на более низком SPL, это может быть сделано только асинхронно.

Напротив, при необходимости выполнить код на более высоком SPL, он может быть выполнен синхронно, для этого необходимо повысить текущий SPL до того уровня, которым должен обладать вызываемый код.

Отсюда следует такая схема: пользовательские потоки всегда могут использовать одинаковый код для синхронизации как в одной так и в другой схеме, поскольку SPL всегда повышается, это происходит синхронно с выполнением потока. В случае, когда используются отложенные процедуры, повышение должно производиться до уровня запрета программных прерываний (и как следствие этого - вытеснения и планировщика). Если планировщик использует запрет прерываний - то повышение SPL до максимального, который гарантирует запрет как программных так и аппаратных прерываний.

В случае с прерываниями все несколько сложнее. Если используется вариант в котором планировщик является непрерываемым, прерывания также повышают SPL и все происходит по аналогии с потоками. В противном случае, очередь отложенных процедур используется для откладывания вызова до тех пор, пока SPL не будет понижен до соответствующего уровня, где их выполнение разрешено.

Схема с непрерываемым планировщиком называется **унифицированной**, такое название указывает на унификацию использования примитивов синхронизации из потоков и из обработчиков прерываний. Альтернативный вариант называется **сегментированной** схемой, это название следует из принципа разделения обработчиков прерываний на две части или два сегмента, выполняемых асинхронно на разных уровнях SPL.

В случае синхронных вызовов все выглядит просто, все вызовы приводящие к повышению SPL повышают его синхронно и затем выполняют все остальные действия. Асинхронный вызов может быть представлен следующим образом: код пишется так, как если бы всегда использовалась сегментированная схема, то есть использует очередь и две функции. Ключевой момент заключается в передаче адреса отложенной функции в процедуру вставки элемента в очередь, это позволяет, в зависимости от реализации такой функции, либо реально поместить аргументы в объект и вставить его в очередь, либо вызвать

функцию в данном месте непосредственно, что и позволяет реализовать в рамках одного интерфейса обе схемы.
Для управления очередью отложенных процедур используются следующие функции:

```
void dpc_init(dpc_t* dpc_object);
bool dpc_insert(dpc_t* dpc_object, void (*func)(void*), void* arg);
bool dpc_cancel(dpc_t* dpc_object);
void dpc_handle_queue(void);
```

После инициализации объекта "элемент очереди" с помощью dpc_init, он может быть помещен в очередь с помощью функции dpc_insert. Отложенная процедура может быть извлечена из очереди с помощью dpc_cancel. Функции вставки и извлечения объекта в/из очереди возвращают статус указывающий на успех или неуспех запрошенной операции. Обработка очереди осуществляется с помощью функции dpc_handle_queue вызываемой внутри ядра ОС перед вызовом планировщика.
Захваты спинлока в любом случае приводят к повышению SPL до максимального, поэтому реализация этих функций идентична для обеих схем синхронизации. Эти функции могут вызываться как из обработчиков прерываний так и из пользовательских потоков. поскольку в обоих случаях SPL повышается, эти функции работают синхронно. Запрет планировщика может вызываться только из пользовательских потоков, по отношению к которым SPL тоже всегда повышается. В случае унифицированной схемы до уровня запрета всех прерываний (SYNC), либо до запрета программных прерываний (DISPATCH) в случае сегментированной схемы.
В сегментированной схеме синхронизации реализация должна работать именно как очередь, поэтому функции dpc_insert/dpc_remove производят запрошенное действие с очередью, предварительно перемещая переданные параметры в объект DPC. Функция же dpc_handle_queue вызывается как часть обработки программного прерывания, поэтому все функции, поставленные в очередь из обработчиков аппаратных прерываний, реально вызовутся только тогда, когда все эти обработчики закончатся и управление перейдет к наименее приоритетному программному прерыванию, то есть вызов отложенной процедуры произойдет асинхронно относительно запросившего этот вызов обработчика аппаратных прерываний.
В случае унифицированной схемы реализация DPC тривиальна: функция dpc_insert реализуется в виде макроса, вызывающего переданную функцию прямо в месте вызова dpc_insert.
То есть, произойдет вызов отложенной процедуры внутри обработчика прерывания. Обращения к примитивам синхронизации и к планировщику, которые, возможно, содержатся в теле отложенной процедуры будут повышать SPL до уровня синхронизации с планировщиком, то есть до максимального, и, таким образом, очередь отложенных процедур исчезает. Все описанные функции вызываются синхронно, так как, в отличие от сегментированной схемы, не возникает ситуации, когда нужно выполнить код с низким SPL из контекста с высоким SPL.
Возможны также различные оптимизации этой схемы и набор функций может быть расширен. Можно ввести дополнительные функции связанные с запретом прерываний, которые будут разрешать/запрещать прерывания в случае сегмантированной схемы, и

отображаться на макросы-заглушки в случае унифицированной схемы, так как в последней блокировка планировщика автоматически означает и блокировку прерываний. Такие функции могут потребоваться потому, что явное использование функций HAL для управления SPL может сделать невозможным использование альтернативной схемы синхронизации, поэтому следует использовать более высокоуровневый интерфейс, вроде описанного выше.

Подводя итог, можно привести пример кода, обрабатывающего прерывания.

```
static dpc_t dpc;
void DPC(void* arg);

void ISR(void* arg)
{
  /* Работа с регистрами устройства */
  dpc_insert(&dpc, DPC, NULL);
}

void DPC(void* arg)
{
  /* Перевод потока в состояние готовности. */
  ...
}

void Thread(void* arg)
{
  dpc_init(&dpc);
  /* Инициализация примитива синхронизации. */

  while(1)
  {
    /* Ожидание активации примитива синхронизации. */
    /* Обработка прерывания. */
    ...
  }
}
```

В случае использования унифицированной схемы синхронизации, функция DPC вызывается непосредственно внутри ISR, а поток использует повышение SPL до максимального уровня на время работы с данными планировщика. Функция DPC вызывается на том же SPL, что и ISR. В том случае, когда используется сегментированная схема синхронизации, функция DPC вызывается асинхронно по отношению к ISR, а поток может не запрещать прерывания во время использования функций планировщика, поскольку при таком подходе функция DPC работает на уровне SPL равном DISPATCH.

5.6 Многопроцессорные системы

5.6.1 Симметричный мультипроцессор

Наличие нескольких процессоров сразу привносит в операционную систему качественно новые проблемы. Особенно остро стоит вопрос синхронизации: запрет прерываний больше особо ничего не гарантирует, поскольку влияет только на текущий процессор. Со своей стороны, все, что может предложить для синхронизации аппаратура - атомарные

операции и спинлоки.

Один из подходов к построению ОСРВ, поддерживающих многопроцессорность заключается в том, что можно на каждом ядре запустить "однопроцессорные" ядра и организовать взаимодействие между ними через общую память и спинлоки в ней же. ОС, построенная таким образом, дает разработку прикладного ПО намного меньше того, что могла бы. В частности, сильно возрастают накладные расходы на коммуникацию и трудности в использовании общих примитивов синхронизации. Фактически, вся логика связанная с межпроцессорной синхронизацией перемещается в пользовательское приложение. Система при этом выглядит как показано на Рис. 59.

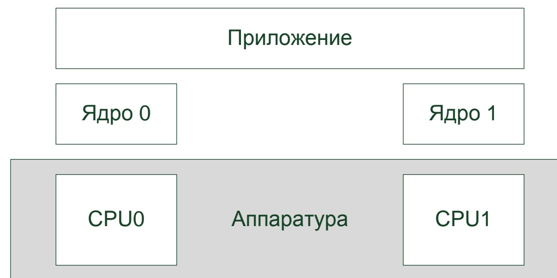

Рис. 59 Использование отдельных экземпляров ОС в многопроцессорной системе

Можно защитить ядро целиком от возможности параллельного выполнения его кода на разных процессорах (Рис. 60).

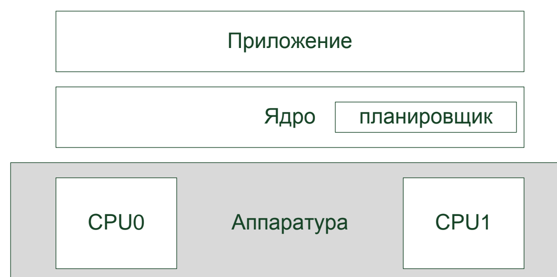

Рис. 60 Использование единого экземпляра ОС в многопроцессорной системе

Такой подход называется BKL (big kernel lock) и немногим лучше первого, так как приемлем только для небольшого числа процессоров и малого кода самого ядра. В противном случае, особенно при росте числа процессоров наступает момент, когда добавление новых процессоров в систему приводит к деградации ее производительности [12], что, разумеется, не порадует ее потенциальных пользователей. Существование такого "стеклянного потолка" у простого решения заставляет разработчиков искать решения более сложные и защищать спинлоками только те места, в которых реально требуется защита разделяемых данных от совместного доступа, чтобы процессоры как можно меньше влияли друг на друга. Это требует тщательного проектирования и не менее тщательного тестирования, но усилия вознаграждаются, и чем более ювелирно построена работа с блокировками, тем более линейно будет расти производительность и тем меньше времени процессоры будут проводить в активных циклах ожидания.

Другой подход предполагается выделение процессоров для выполнения некоторой

работы. При этом может быть ведущий процессор и некоторый набор ведомых, которые получают указания от ведущего, что им делать. Для систем общего назначения это, очевидно, приведет к проблеме бутылочного горлышка на ведущем процессоре, как только мы дойдем до предела его вычислительных возможностей по раздаче работы другим дальнейшее добавление процессоров в систему уже не будет приводить к росту проиизводительности, а добавленные процессоры будут бездействовать. Однако во встроенных системах реального времени в таком дизайе может быть смысл. Использование выделенных процессоров позволяет избежать накладных расходов на балансировку нагрузки, если заранее известно, какого типа задачи будет решать система и известно что возможностей ведущего хватит, то такой подход может дать хорошие результаты. Система с выделенным процессором является частным случаем симметричной системы, поскольку распределить роли между процессорами можно динамически и после этого их не менять, поэтому симметричная система это более общий подход.

Наконец, можно рассмотреть наиболее сложную задачу: каким образом можно построить ядро так, чтобы одни процессоры вообще не влияли на другие до тех пор, пока они не обращаются к разделяемым данным и работали максимально независимо. Такая архитектура также может применяться, обоснованность ее применения зависит от задач стоящей перед встроенной системой.

Поскольку аппаратное обеспечение накладывает ограничения на то, каким образом происходят взаимодействия процессоров, операционная система может лишь дополнить имеющиеся аппаратные механизмы. Так как единственный способ взаимодействия процессоров в классических симметричных многопроцессорных системах (а также в HAL) это обмен прерываниями, логично использовать их для того, чтобы общаться с другими процессорами. Вариант с единым планировщиком не является досаточно хорошим решением, как из-за проблем с масштабируемостью так и производительностью, поэтому обычно каждый процессор имеет свой контейнер и привязанные к нему потоки, такой подход позволяет минимизировать межпроцессорный трафик по шине и, в идеале, добиться нулевых накладных расходов на многопроцессорность (Рис. 61).

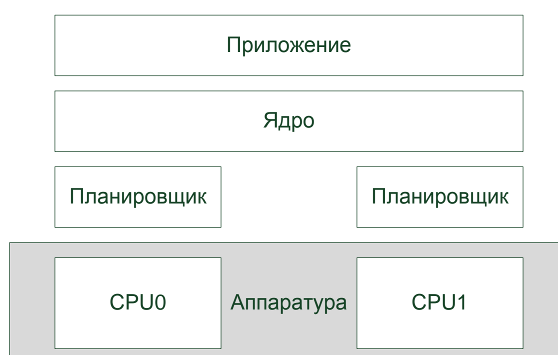

Рис. 61 Гибридный подход к построению многопроцессорной системы

Трудность заключается в том, что, с точки зрения программиста, система должна выглядеть единой, потоки должны уметь возобновлять выполнение и изменять параметры в том числе и удаленных потоков (находящихся в планировщике других

процессоров), что может повлечь за собой изменение структур данных планировщика. Синхронизация такого взаимодействия является довольно сложной задачей даже в случае глобальной блокировки планировщика.

Например, захватив спинлок, можно произвольным образом изменить параметры удаленных потоков, однако сами эти потоки ничего об этом не знают до тех пор, пока не произошел вызов планировщика, поэтому на короткое время возможны даже ситуации когда выполняемый поток не является самым приоритетным на том процессоре, что расходится с поведением однопроцессорной системы.

Таким образом, блокировать весь планировщик и удаленно манипулировать его структурами данных не очень эффективное решение, поэтому можно попробовать перенести логику в сам обработчик межпроцессорного прерывания: в таком случае можно подготовить какой-то контекст, указывающий на то, что нужно сделать, и послать прерывание, обработчик которого выполнится уже на "правильном" процессоре и после изменения состояния сможет сразу же вызвать планировщик. У прерываний есть только такой недостаток, что если приходит еще одно в тот момент когда первое еще не начало обрабатываться, оно будет утеряно. Поэтому, для получения нужных гарантий, можно использовать следующий шаблон асинхронного взаимодействия: каждый процессор содержит очередь некоторых действий о которых его может "попросить" другой процессор. Сама "просьба" выглядит так: поставить дейтвие в очередь, послать процессору прерывание. Если какой-то еще процессор в этот момент обращается в ту же самую очередь, целевой процессор все равно увидит и обработает все элементы очереди.

5.6.2 Асимметричный мультипроцессор

Асимметричные процессоры получают всё большее распространение в области встраиваемых систем благодаря универсальности, которую они могут обеспечить, когда каждый вид задач работает на специально для него выделенном физическом процессоре.

Расплатой за эту универсальность является дальнейшнее усложнение как системного так и прикладного ПО.

В отличие от симметричных мультипроцессоров, которые всегда видятся системным ПО более или менее единообразно - как несколько одинаковых процессоров, имеющих доступ к общему пространству физической памяти, асимметричные мультипроцессоры допускают гораздо большее разнообразие. Это разнообразие может заключаться как, собственно, в том что процессоры разные, имеющие разный набор команд, и, следовательно, вытекающую из этого невозможность использовать общий код как ОС так и приложений, так и в отсутствии глобального физического адресного пространства памяти.

Классическим случаем является общий регион физической памяти, через который может осуществляться взаимодействие, остальная память с точки зрения каждого из процессоров является приватной и недоступной другим процессорам. Некоторые следующие из этого сложности программирования компенсируются тем, что система получается довольно простой в аппаратном смысле: поддерживать в согласованном состоянии небольшой регион общей памяти это совсем не одно и то же что поддерживать всю физическую память согласованной между всеми процессорами и всеми уровнями

кэшей этих процессоров.

Если общая память отсутствует, система лишается большинства возможностей коммуникации и для такой системы наиболее универсальным решением является разработка ПО для каждого процессора отдельно, как если бы они были коммуницирующими отдельными устройствами (при отсутствии общей памяти они фактически и являются таковыми).

Дополнительные проблемы создает также разнообразные способы коммуникации между процессорами. В простейшем случае какие-либо способы коммуникации кроме общей памяти отсутствуют вовсе, а синхронизация доступа к этой общей памяти может осуществляться только за счет активного ожидания на какой-либо переменной также размещенной в общей памяти. В некоторых случаях поддерживается обмен прерываниями, по аналогии с симметричным случаем, это наиболее удобный способ взаимодействия с точки зрения ПО. Используются также экзотические случаи, например оба процессора могут иметь доступ к устройству, которое реализует некоторое подобие аппаратной очереди. Тема программирования ассиметричных мультипроцессоров настолько глубока и многогранна, что для ее более-менее полного описания потребовалась бы отдельная книга, поэтому в данном разделе будет рассмотрен только наиболее удобный для ПО случай:

- у процессоров есть (возможно небольшой, 1-10 Кб) регион общей памяти
- у процессоров совпадает разрядность и порядок байт в слове
- процессоры могут каким-либо образом обмениваться прерываниями.

У такой системы много общего с симметричными мультипроцессорами, поэтому ОС поддерживающая симметричный мультипроцессор может быть относительно легко адаптирована для работы с асимметричными системами подобного рода.

Ключевая идея заключается в том, что если у процессоров входящих в асимметричную систему одинаковый порядок байт в слове, а также одинаковая разрядность, то оба процессора будут видеть одинаковые структуры данных в памяти и трактовать их тоже одинаково, поэтому можно установить на каждый процессор свою собственную ОС, по аналогии с однопроцессорным случаем, но структуры данных, не требующие выполнения кода другого процессора, такие как примитивы синхронизации, могут быть общими и потоки могут ждать друг друга и сигнализировать точно так же, как это происходит в симметричных многопроцессорных системах. Спинлоки также могут использоваться совместно.

Поскольку использование общих примитивов синхронизации предполагает использование некоторой структуры данных, которая хранит потоки, связанные с примитивом, каждый процессор должен иметь доступ к этой структуре данных. При таком подходе в разделяемой области памяти должны быть размещены используемые совместно примитивы синхронизации, а также объекты потоков, которые могут взаимодействовать с разделяемыми примитивами. Все остальные объекты, в том числе и стеки всех потоков, могут быть размещены в приватной области памяти каждого процессора.

Очевидно, из-за возможных различий в системе команд процессоров, код как ядра так и

приложений не может перемещаться между процессорами. ОС, таким образом, используется в роли слоя, абстрагирующего систему коммуникации.

5.6.3 Балансировка нагрузки

Симметричные многопроцессорные системы имеют такое полезное свойство, что нагрузка может распределяться по процессорам динамически во время работы системы.

Во встроенных системах применяется несколько основных подходов, посвященных использованию многопроцесорных платформ для систем жёсткого реального времени. Самый простой из них заключается в том, что потоки жёстко привязываются к одному из процессоров и в процессе работы эта привязка не изменяется. То есть динамическое распределение ресурсов отсутствует. Несмотря на кажущуюся неэффективность этого подхода, он заслуживает внимания хотя бы потому, что зачастую во встраиваемых системах используются выделенные процессоры, то есть физический процессор используется только для выполнения какой-либо конкретной работы. Такой дизайн предполагает распределение потоков по процессорам на этапе инициализации системы. Подобный подход исключает накладные расходы на планирование и балансировку нагрузки, а также улучшает предсказуемость за счет того, что время реакции перестает зависеть от возможных миграций потоков с процессора на процессор. Также использование выделенных процессоров улучшает производительность системы за счет того, что кэш и прочие аппаратные ресурсы процессора не испытывают негативного влияния посторонних потоков. В целом, такой подход похож на асимметричную систему, поскольку преимущества того, что любой поток может исполняться на любой процессоре никак не используется. Альтернативные подходы предполагают тем или иным образом распределять потоки между процессорами.

Из-за наличия в системе разделяемых данных, выполнение кода текущим процессором может задерживаться на спинлоках из-за действий других процессоров, поэтому такая система принципиально обладает худшей предсказуемостью, по сравнению с однопроцессорными системами, где вход в критические секции (с помощью запрета прерываний) всегда происходит за фиксированное время. Из-за сложности реализации и малого распространения соотвествующего оборудования, ОСРВ, поддерживающие симметричные мультипроцессоры, существуют на рынке в меньшинстве.

Одним из способов балансировки нагрузки (за исключением случая с отсутствием такой балансировки) является принцип "N самых приоритетных потоков в системе работают на N процессорах". Очевидно, что жёсткое соблюдение этого условия, когда его нарушение невозможно ни в какой момент времени работы системы, требует глобальных блокировок ядра или планировщика, поэтому сталкивается с сопутствующими проблемами с производительностью.

Альтернативные реализации исходят из того факта, что симметричная многопроцессорная система является принципиально асинхронной на уровне аппаратуры, а значит, можно ее рассматривать как некое подобие распределенной системы, где вышеуказанный принцип может нарушаться, то есть система старается его выполнить, но не гарантирует его жёсткого выполнения в любой момент времени.

Реализация такой политики обычно включает выбор нового процессора для потока в

моемнты перепланирования или перевода его из состояния ожидания в состояние готовности. Каждый процессор поддерживает переменную, указывающую на его состояние, занят ли он или простаивает. При выборе нового процессора для потока можно проанализировать все такие переменные, и, определить новый процессор для потока, если текущий перегружен работой. Правда, состояние целевого процессора может измениться в то время, когда поток на него мигрирует, но вероятность этого события невелика.

Наконец, особняком стоит подход, провозглашающий неэффективность любого автоматического распределения, из-за непредсказуемости, которую он вносит в работу потенциально ответственной системы. Поэтому распределение нагрузки переносится в приложение. ОС предоставляет механизмы для того, чтобы приложение могло управлять привязкой потоков к процессорам, а по умолчанию балансировка нагрузки ядром не выполняется, потоки всегда работают на тех процессорах, на которых они были созданы.

5.7 Завершение работы потока

Завершение работы потока является наиболее сложной из всех операций, которые над ним можно производить. В отличие от изменения приоритета и прочих параметров потока, где, в принципе, допускаются связанные с асинхронностью "аберрации", например, когда в результате одновременно происходящих синхронного и асинхронного изменения приоритета поток может оказаться не в том состоянии, которое фактически было запрошено последним. С завершением работы такого происходить не должно. В дополнение к этому, операция завершения потоком самого себя является синхронной и всегда должна быть успешной так как не возвращает и не может возвращать результат.

Если поток, начав синхронное завершение своей работы, вдруг видит, что его собственное удаление уже производится каким-то внешним потоком, он оказывается в затруднительной ситуации: продолжать работу нельзя, так как эту же работу выполняет и какой-то удаленный поток, однако и вернуть ошибку из функции тоже нельзя, потому что функция не возвращает результата и анализировать его некому. Асинхронное удаление потока также должно учитывать ситуацию, когда поток находился в состоянии ожидания, то есть должно разрывать связь потока с примитивами синхронизации, однако это опять же сопряжено с трудностями, поскольку никогда нельзя быть уверенным в том, что операция завершена. Удалив объект из всех очередей, поток может вновь начать какое-то ожидание, так как выполняется независимо от кода, который пытается его удалить. Если запрещать ему это делать, выставляя некий признак "текущий поток удаляется", увидев его, поток опять оказывается в затруднительном положении, выходом из которого является только активное ожидание.

Варианты с использованием привязанных к процессору очередей, через которые его можно "просить" выполнить некоторые действия, не работают потому, что поток может перемещаться между процессорами (мигрировать), в том числе и в тот момент, когда кто-то начал его удалять. Поэтому его можно долго и безуспешно "ловить", пересылая информацию о нем от процессора к процессору, что не добавляет системе ни предсказуемости, ни производительности.

Все эти рассуждения подводят к идее, что идеальным вариантом была бы некоторая

процедура, привязанная к потоку, которая всегда выполнялась бы в контексте этого потока, и, следовательно, была бы избавлена от всех вышеперечисленных проблем: во-первых миграция потоков больше не является проблемой, так как эта функция всегда мигрирует вместе с потоком, во-вторых она выполняется синхронно с кодом потока, поэтому может делать любые действия от имени самого потока и быть уверенной в том, что сам поток в это время ничего не делает. Такие функции называются асинхронными процедурами[2] (asynchronous procedure call, APC), из-за того, что вызываются асинхронно в контексте потока.

В целом, асинхронные процедуры являются в некотором смысле аналогами отложенных процедур, разница только в том, что последние всегда привязаны к физическому процессору, а не к потоку (Рис. 62).

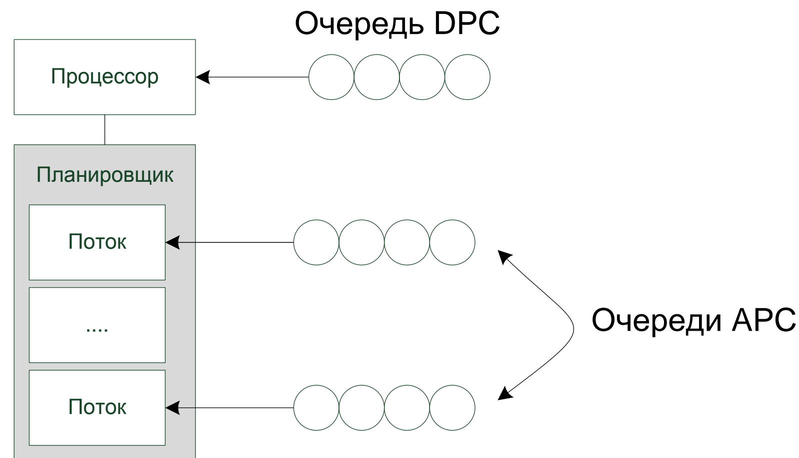

Рис. 62 Асинхронные и отложенные процедуры

Асинхронные процедуры позволяют также улучшить временные характеристики ОСРВ. Поскольку процесс удаления потоков должен быть синхронизирован с самим потоком, единственной альтернативой асинхронным процедурам является блокирование планировщика на время выполнения удаления. Это не обязательно быстрая операция, поэтому может оказать негативное влияние на время реакции всей системы, так как даже высокоприоритетные потоки оказываются лишены возможности выполняться в то время, когда происходит процесс завершения одного из потоков. Использование асинхронных процедур позволяет перенести все действия по удалению потока в сам поток, выполняемый со своим приоритетом, поэтому другие потоки не затрагиваются, а критические секции могут быть намного короче.

Необходимость в существовании таких функций появилась практически сразу после начала систематических исследований в области операционных систем. В UNIX они наываются сигналами и являются частью стандарта POSIX. Возможность выполнить произвольный код в контексте указанного потока широко используется и в других ОС, в том числе там, где это скрыто от пользователя.

---

[2] Как и в случае с отложенными процедурами, в разных ОС асинхронные вызовы процедур называются по-разному (например, ASR или сигналы), хотя и выполяют одну и ту же функцию. Здесь и далее будет использоваться название APC.

Что касается реализации, то в простейшем случае используется битовая маска, хранящая асинхронные запросы. В определенные моменты времени, обычно в обработчике программных и аппаратных прерываний, состояние этой маски анализируется, и, если есть ожидающие запросы, то происходит операция **доставки**. Доставка сигнала или асинхронной процедуры это модификация состояния потока таким образом, чтобы в его контексте начал выполняться обработчик. Интерфейс обычно включает функцию отправки, управления масками и внутреннюю функцию анализа состояния, которая используется внутри ядра ОС и прозводит доставку.

Доставка может быть произведена с помощью модификации фрейма из обработчика программного прерывания:

```
void signal_deliver(fx_thread_t* current_thread)
{
  ... /* Получение функции-обработчика из внутренних структур данных потока.*/

  intr_frame_t* old_frame = intr_frame_get();
  intr_frame_t* new_frame = intr_frame_alloc(old_frame);

  intr_frame_modify(...);
  intr_frame_set(new_frame);
}
```

Со временем, правда, обнаружился такой недостаток, что в случае использования битовой маски повторные сигналы теряются, поэтому в последующих редакциях стандарта POSIX были введены очереди сигналов. Сигналы в этом случае представлены не битами, а объектами, которые помещаются в очередь, следовательно, множественные уведомления потоков с помощью сигналов не теряются.

Наличие асинхронных процедур определяет принципиальное различие между обсуждавшимися функциями синхронного и асинхронного завершения потока. Как и все другие асинхронные взаимодействия, такие как прерывания, вытеснение и выполнение отложенных процедур, выполнение асинхронных процедур может быть временно запрещено потоком. В этом заключается разница между вызовами fx_thread_exit и fx_thread_terminate. Последняя реализуется через асинхронные процедуры, поскольку завершение потока с ее помощью может быть инициировано другим потоком. Поэтому, если поток запретил выполнение асинхронных процедур, он в принципе может использовать fx_thread_terminate для завершения самого себя, фактическое завершение при этом будет отложено до тех пор, пока поток не покинет критическую область в которой выполнение асинхронных процедур запрещено. Вызов же fx_thread_exit в таком контексте приводит к немедленному синхронному завершению работы.

В отличие от множества других концепций, таких как прерывания или отложенные процедуры, необходимость в использовании асинхронных процедур возникает только в достаточно сложных системах, в частности, поддерживающих многопроцессорность. В однопроцессорных системах необходимая синхронизация достигается более просто - запрет прерывания или планирования уже обеспечивает необходимые гарантии, так как переключение контекста не может произойти, текущий поток может делать что угодно и при этом быть в согласованном состоянии относительно всех других потоков, поскольку

он выполняется на единственном процессоре. Это одна из причин того, что концепция асинхронных процедур является не настолько широко известной. Другая причина заключается в том, что использование асинхронных процедур и сигналов в прикладном программировании приводит к более сложной синхронизации и трудностям в поддержке: в любой момент в любой точке кода нужно предполагать выполнение не только прерываний, но и асинхронных процедур.

Асинхронные процедуры в некотором смысле являются виртуальными прерываниями для виртуального процессора которым является поток. Следовательно, как и всякие прерывания, они могут иметь приоритет и должны иметь возможность максировки для того, чтобы код потока мог использовать разделяемые данные с кодом асинхронных процедур. Количество приоритетов зависит от архитектуры ядра и выполняемых внутри асинхронных процедур работ. Поскольку потоки выполняются всегда на уровне SPL = LOW, можно считать уровни приоритетов APC в качестве продолжения основных уровней SPL. В некоторых ОС, как, например Windows NT уровни IRQL (эквивалентные по смыслу уровням SPL) распространяются в том числе и на асинхронные процедуры, поэтому, начиная с уровня DISPATCH и ниже, состояние уже специфично для конкретного потока. Несмотря на некоторую унификацию, такой подход делает систему менее гибкой из-за смешения концепций определяемых аппаратурой и HAL и определяемых конкретной реализацией потоков, хотя он вполне оправдан, если большие изменения архитектуры и конфигурирование не предполагаются. В FX-RTOS эти понятия разделены, уровни маскировки прерывания определяются внутри модуля потоков и реализуются внутри него же, поэтому для управления ими используется отдельное API.

Когда происходит доставка APC внутри функции планирования, все APC данного приоритета маскируются, по аналогии с тем как это происходит для аппаратных прерываний. После обработки APC уровень возвращается на прежний уровень, что позволяет избежать вложения APC.

5.8 Реализация

5.8.1 Реализация планировщика

Планировщик состоит из двух основных компонентов: хранилища планируемых элементов, из которого выбирается наиболее приоритетный элемент и объемлющего слоя, в котором скрыта синхронизация доступа к этому хранилищу. Хотя синхронизация может различаться от системы к системе, устройство хранилища и связанный с ним алгоритм планирования остается практически неизменным для широкого круга систем, поэтому имеет смысл рассмотреть его в первую очередь.

Все функции планировщика выполняются на уровне SPL, определяемом текущей схемой синхронизации. В любом случае, планировщик может предполагать, что его код не вытесняется и асинхронное изменение структур данных невозможно. Если планировщик представляет собой контейнер готовых к выполнению потоков, из которых следует выбрать один, его интерфейс имеет смысл сделать похожим на интерфейс коллекций.

Имеется два вида объектов, сами контейнеры, и элементы, которые в них могут находиться.

```
/*Контейнер, содержащий готовые к выполнению потоки.*/
typedef struct {...} fx_sched_container_t;

/* Тип элемента контейнера. */
typedef struct {...} fx_sched_params_t;
```

Контейнер должен иметь методы для добавления в него элементов, удаления, а также выбора текущего "самого подходящего". В первом приближении этот интерфейс должен выглядеть следующим образом:

```
void fx_sched_container_init(fx_sched_container_t* containter);
void fx_sched_container_add(fx_sched_container_t* container, fx_sched_params_t* item);
void fx_sched_container_remove(fx_sched_container_t* cntner, fx_sched_params_t* item);
fx_sched_params_t* fx_sched_container_get(fx_sched_container_t* containter);
```

Функции, работающие с элементом контейнера ничем не примечательны, поскольку никак не взаимодействуют с контейнером. Желательно, чтобы все функции контейнера работали за фиксированное время, поэтому можно использовать подход, называемый "многоуровневая приоритетная очередь" (multi-level priority queue, MLQ), который является широко распространенным для реализации FIFO-планировщиков. Данная структура данных содержит массив из очередей, количество которых равно количеству приоритетов, используемых в системе. Использование MLQ приводит к тому, что все функции планировщика не зависят как от количества потоков вообще так и от количества потоков, готовых к выполнению.
Контейнер потоков представляет собой структуру изображенную на Рис. 63.

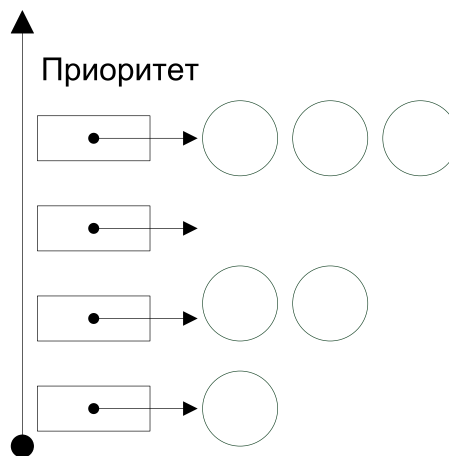

Рис. 63 Структура данных планировщика с фиксированными приоритетами

С каждым приоритетом связан набор потоков, имеющих данный приоритет, поскольку количество приоритетов фиксировано, время поиска наиболее приоритетного потока также ограничено.
Добавление и удаление элементов довольно тривиально: используется вставка либо удаление элемента из соответствующего списка, где в качестве индекса в массиве списков используется приоритет, получаемый из элемента. Главной же функцией планировщика, является, конечно, получение текущего наиболее приоритетного элемента: fx_sched_container_get. Ее можно реализовать в виде цикла ищущего непустой список

начиная с 0-го индекса (или с максимального, в зависимости от того, в каком направлении растет приоритет). Хотя поиск непустого списка в массиве и ограничен сверху количеством приоритетов, то есть соответствует критериям реального времени, на практике стремятся сделать это время константой для всех возможных случаев, поэтому часто можно встретить оптимизации, основанные на поддерживаемой большинством процессоров команде поиска нулевого или единичного бита. Структура данных, используемая таким алгоритмом показана на Рис. 64. В контейнер добавляется дополнительное поле, имеющее тип данных, соответствующий размеру регистра процессора, например uint32_t, для 32-разрядных процессоров. Каждый бит в этом поле определяет, есть ли элементы в списке, индекс которого соответствует индексу бита. Например, если бит 5 равен 1, то 5-ый список в массиве непустой.

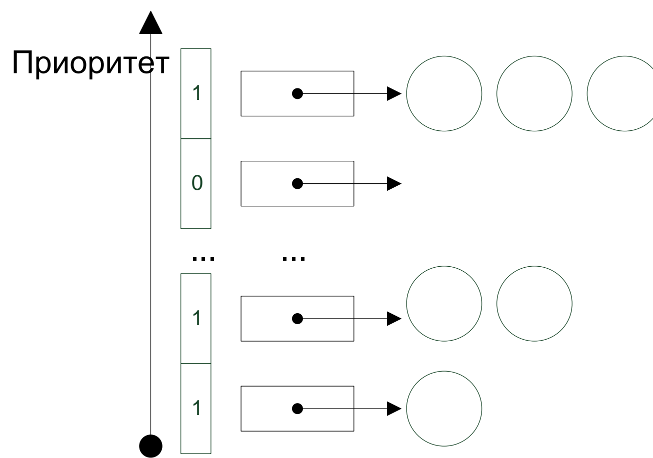

Рис. 64 Оптимизация поиска с помощью битовых операций

Операции добавления и удаления элементов могут пользоваться соответствующими операторами языка С для изменения битовой маски. Например, добавление всегда делает список непустым, так что функция fx_sched_container_add будет содержать строку:

```
mask |= 1 << priority;
```

Удаление же должно быть условным, и в случае, если удаление элемента привело к опустошению соответствующего списка, должен выполниться оператор:

```
mask &= ~(1 << priority);
```

Тогда функция получения наиболее приоритетного элемента будет выглядеть как определение индекса первого непустого списка с дальнейшим обращением по найденному индексу без поиска по массиву.

Различные процессоры предоставляют различные интерфейсы для подсчета битов. Это могут быть как инструкции типа bit_scan, возвращающие непосредственно индекс и какое-либо специально значение, в случае если единичных битов в переданном слове нет, так и инструкции вида CLZ (count leading zeros), возвращающие количество старших нулевых бит. Все эти функции сводимы одна к другой, поэтому реализовав в платформенно-зависимой части систему интерфейс для битовых операций, планировщик

может быть написан кроссплатформенным.
Алгоритм имеет только такой существенный недостаток, что количество приоритетов не может превышать количество битов в слове процессора. Для совсем простых систем 16 или 32 приоритета может быть достаточно, но, разумеется, такого количества приоритетов мало для всех возможных приложений. В случае если количество приоритетов больше, чем количество битов в аппаратном слове, часто применяются комбинированные иерархические схемы. В этом случае битовая маска указывает уже не напрямую на пустые/непустые списки, а на второй уровень битовых масок (Рис. 65).

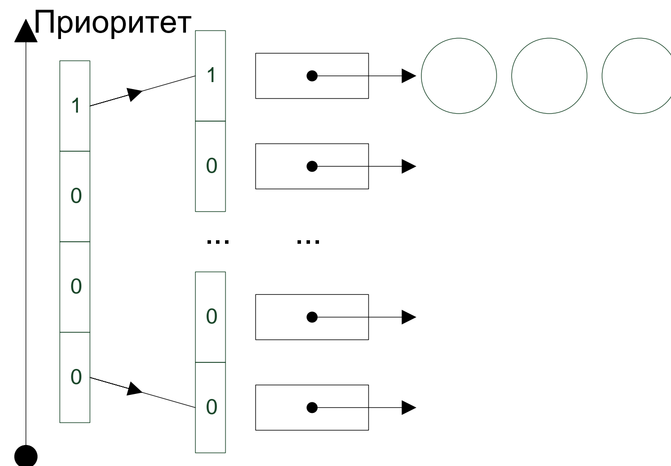

Рис. 65 Иерархическая оптимизация поиска с помощью битовых масок

Биты в первом уровне устаналиваются в 1 в том случае, если соответствующая битовая маска второго уровня имеет хотя бы один ненулевой бит. Поиск происходит в 2 этапа, вначале определяется ненулевой бит в первой маске, затем ненулевой бит в соответствующей маске второго уровня, после чего определяется непустой список. В случае 32-разрядного процессора это дает 32*32 = 1024 уровня приоритета, чего уже вполне достаточно для абсолютного большинства приложений.

Могут применяться также алгоритмы поиска битов основанные на табличных методах.
Функция получения наиболее приоритетного элемента не должна извлекать элемент из контейнера. Поскольку планирование может по разным причинам запрашиваться и в процессе работы одного потока, нецелесообразно каждый раз возвращать текущий элемент обратно в контейнер только для того, чтобы на следующем этапе его извлечь.

5.8.2 Синхронизация планировщика

Контейнер только поддерживает структуру данных, в которой содержатся готовые к выполнению потоки, однако он не отслеживает ни изменение состояния потоков ни текущий контекст. Все эти функции реализует объемлющий слой, в котором заключена синхронизация доступа к хранилищу планировщика. Синхронизационный слой планировщика является наиболее сложной в реализации частью всей подсистемы потоков. Если контейнер планировщика в многопроцессорной системе существует в виде нескольких копий, привязанных к процессору, то слой синхронизации является той

частью, которая позволяет всей остальной системе видеть набор контейнеров как некоторую глобальную для системы структуру данных.
Слой синхронизации может иметь собственные данные, привязанные к элементу, поэтому имеет смысл ввести его собственную структуру данных.

```
typedef struct {... fx_sched_params_t params; ... } fx_sched_item_t;
```

Через данный модуль должны также проходить все запросы блокировки планировщика, на случай использования каких-либо глобальных блокировок внутри модуля, которые должны быть заблокированы. Для этого вводится структура данных fx_sched_state_t, хранящая состояние блокировки планировщика. В большинстве случаев этот тип совпадает с типом SPL. Пара функций предназначенных для блокировки планировщика также тривиальны.

```
void fx_sched_lock(fx_sched_state_t* old_state);
void fx_sched_unlock(fx_sched_state_t state_to_be_restored);
```

Как правило, (если не используются глобальные блокировки) эти функции напрямую отображаются на соответствующие функции, управляющие текущим уровнем SPL как в однопроцессорных так и в многопроцессорных системах.
Следующая группа функций примерно совпадает с соответствующими интерфейсами контейнера.

```
void fx_sched_item_init(
  fx_sched_item_t* item, fx_sched_params_init_t t, const fx_sched_params_t* arg);
void fx_sched_item_add(fx_sched_item_t*);
void fx_sched_item_remove(fx_sched_item_t*);
fx_sched_item_t* fx_sched_get_next(void);
```

В качестве контейнера в данном случае выступает вся система. В том случае, когда используется планирование по расписанию, контейнер может переключаться по таймеру, но далее будет рассматриваться упрощенный вариант. Все эти функции могут использоваться только на уровне SPL, соответствующем планировващику, иными словами, в той области, где планировщик заблокирован с помощью вызовов fx_sched_lock/fx_sched_unlock. Потоки всегда добавляются в контейнер того процессора, на котором просходит вызов fx_sched_item_add, соответственно, планирование также возващает поток из контейнера, соответствующего текущему процессору. Эти же ограничения распространяются и на fx_sched_item_remove: вызывающий код должен гарантировать что удаляемый элемент находится в контейнере того же процессора на котором происходит вызов. На первый взгляд это порождает проблему синхронизации, ведь удаление потоков можно инициировать с любого процессора, а не только с того, на котором поток выполняется в данный момент. Проблема решается с помощью асинхронных процедур: фактическое извлечение потока из контейнера осуществляется самим этим потоком из соответствующей асинхронной процедуры, выполнение которой и запрашивается через функции асинхронного удаления. В случае синхронного завершения работы потока вызовом fx_thread_exit такой проблемы нет, так как поток всегда работает на своем процессоре.

Наконец, последняя группа функций предназначенная для изменения различных параметров потоков. Разберемся, что с потоком вообще можно делать удаленно (из потоков на других процессорах): во-первых, можно изменить ему приоритет или другой аналогичный параметр, во-вторых - если поток был в приостановленном состоянии - возобновить его выполнение, если например на текущем процессоре был активирован примитив синхронизации, который ожидался удаленным потоком, нужно иметь возможность сделать так, чтобы тот поток продолжил выполнение.

```
void fx_sched_item_set_params(fx_sched_item_t* item, const fx_sched_params_t* src);
void fx_sched_item_get_params(const fx_sched_item_t* item, fx_sched_params_t* dst);
void fx_sched_item_suspend(fx_sched_item_t* item);
void fx_sched_item_resume(fx_sched_item_t* item);
```

В дополнение к этому, могут быть предусмотрены отдельные функции для изменения привязки потока к процессорам, управления миграцией и тому подобное. Все эти функции также должны выполняться с заблокированным планировщиком на уровне SPL = DISPATCH, однако в отличие от первой группы функций, которые работают всегда с контейнером текущего процессора, перечисленные функции могут асинхронно воздействовать на удаленные потоки, содержащиеся в контейнерах других процессоров.
Функция fx_sched_remove также должна гарантировать отмену всех возможных асинхронных операций планировщика, которые могут находиться в процессе выполнения в момент ее вызова. После вызова данной функции вызывающий код вправе предполагать, что связи данного потока с другими процессорами разорваны.
В однопроцессорной системе реализация планировщика довольно проста: после входа в критическую область (где переключение контекста запрещено) выполняющийся код является единственным работающим потоком в системе, поэтому он может изменить любые параметры у любого потока и после выйти из критической области и продолжить выполнение.
Для отслеживания текущего состояния потока может использоваться флаг или счетчик, указывающий на то, готов ли поток к выполнению, и, следовательно, находится ли он в контейнере готовых потоков. Для этой цели часто применяется счетчик, указывающий количество вызовов функции fx_sched_item_suspend над данным объектом, перевод элемент в активное состояние происходит тогда, когда соответствующее число раз была вызвана функция fx_sched_item_resume. Для иллюстрации того принципа рассмотрим код функции fx_sched_item_resume:

```
void fx_sched_item_resume(fx_sched_item_t* item)
{
  if(item->suspend_count > 0 && --(item->suspend_count) == 0)
  {
    fx_sched_container_add(&global_container, &item->params);
    request_software_interrupt(DISPATCH);
  }
}
```

Поскольку вызывающий код должен обеспечивать блокировку планировщка, в однопроцессорной системе никакой синхронизации не требуется. Анализируется счетчик

приостановок, и, в случае если он больше нуля, то есть ранее была вызвана функция fx_sched_item_suspend, счетчик декрементируется. Если счетчик обнулился в результате вызова функции, происходит добавление элемента в контейнер и запрос программного прерывания. Поскольку текущий контекст предотвращает немедленную активацию прерывания, фактический вызов планировщика откладывается до выхода из области с заблокированным планировщиком.

Все прочие функции, изменяющие параметры элемента, должны учитывать его возможное присутствие или отсутствие в контейнере, поэтому счетчик должен анализироваться всякий раз, если изменение параметров может повлечь изменение состояния контейнера. Например, изменение приоритета потока, очевидно, влечет изменение состояния контейнера так как элемент возможно потребуется переместить между списками в MLQ или как-то еще воздействовать на структуру данных контейнера. Код функции изменения приоритета:

```
void fx_sched_item_set_params(fx_sched_item_t* item, const fx_sched_params_t* src)
{
  if(item->suspend_count == 0)
  {
    fx_sched_container_remove(&global, &item->params);
    fx_sched_params_copy(src, &item->params);
    fx_sched_container_add(&global, &item->params);
    request_software_interrupt(DISPATCH);
  }
  else fx_sched_params_copy(src, &item->params);
}
```

Если элемент не приостановлен, то есть находится в контейнере, он извлекается оттуда, затем новые параметры копируются в извлеченный элемент, после чего он помещается обратно в контейнер с последующим запросом программного прерывания, поскольку приоритет мог быть понижен и могла сложиться ситуация требующая вытеснения текущего потока.

Разумеется, была приведена наиболее тривильная реализация и на практике могут применяться различные оптимизации. Например, повышение приоритета для текущего потока может быть произведено без вызова планировщика, поскольку он и так был наиболее приоритетным. Если повышается приоритет другого потока, то сравнив его новый приоритет с текущим приоритетом работающего потока можно сделать вывод, продолжит ли работу текущий поток, то есть, хотя вытеснение и возникнет, его результат известен заранее и полное перепланрование производить не нужно. По сути, только операции понижения приоритета могут приводить к необходимости поиска непустых очередей в MLQ-контейнере.

В однопроцессорной системе все необходимые гарантии достигаются просто за счет того, что все функции планировщика должны вызываться в контексте, где планировщик заблокирован. В многопроцессорной системе такая блокировка влияет только на текущий процессор, тогда как в качестве аргументов могут быть указаны и удаленные потоки, выполняющиеся на других процессорах. Задача усложняется тем, что все эти действия могут происходить на фоне основной работы потока: он может начинать и завершать ожидание и даже заканчивать собственное выполнение, кроме того, инициировать

асинхронные операции могут одновременно несколько удаленных потоков на разных процессорах.
К этой задаче есть несколько подходов. Один из них заключается в том, чтобы использовать глобальные блокировки, которые распространяются на все процессоры в системе. В таком случае, в момент изменения параметров потока, достигаются необходимые гарантии консистентности как состояния этого потока так и глобального состояния системы. Этот подход, однако, не очень хорош в плане эффективности, любые глобальные блокировки на часто используемые операции (а возобновление работы потока это, безусловно, одна из наиболее часто выполняемых операций) приводят к тому, что с ростом количества процессоров, производительность начинает деградировать, поэтому имеет смысл избавиться от любых таких блокировок. Более интеллектуральные алгоритмы предполагают использование блокировки на процессор. В таком случае, при необходимости синхронизации с потоком, выполнящемся на другому процессоре необходимо захватывать два спинлока. Большинство действий требует захвата только одного спинлока, захват двух необходим только для реализации, например, миграции, когда затрагиваются сразу два хранилища потоков. Существуют разновидности этого алгоритма, одна из которых применяется в многопроцессорном варианте FX-RTOS.
Параметры потока можно изменять только синхронно с самим потоком, потому что поток может и сам изменять собственные параметры, поэтому вводится дополнительная копия состояния. Все параметры потока, которые могут изменяться асинхронно, существуют внутри объекта потока в двух экземплярах. Один из них, называемый актуальным (actual) состоянием используется планировщиком и допускает изменение только на этом процессоре, с помощью синхронизации основанной на SPL, как и для однопроцессорных систем. Вторая копия используется для асинхронных изменений и называется последним состоянием (latest). При необходимости возобновить выполнение потока, поменять ему приоритет, либо привязку к процессору, эти изменения производятся над копией. Далее запускается процедура, называемая процедурой эквализации, которая должна привести актуальное состояние потока в соответствие с последним состоянием. Для выполнения такой процедуры нужен какой-то способ выполнить код на том процессоре, на котором сейчас находится поток. Такой способ уже был рассмотрен ранее: очередь отложенных процедур как раз и дает возможность выполнять произвольный код на указанном процессоре в системе и выполнять его синхронно с планировщиком. Этот факт является одной из причин того, что в системах поддерживающих многопроцессорность, как правило применяется сегментированная схема синхронизации.
Если целевой поток работает на том же процессоре, что и запрашивающий изменения, все сводится к однопроцессорному случаю. В противном случае используется объект DPC, содержащийся внутри объекта fx_sched_item_t. С его помощью на удаленном процессоре ставится в очередь отложенная процедура, которая копирует последнее состояние в актуальное и, при необходимости, вызывает планировщик, который обновляет свое хранилище.
В отличие от описанного случая с использованием спинлока на процессор, здесь под, собственно, спинлоком выполняется только работа с DPC, такая как вставка в список, что происходит довольно быстро по сравнению с временем, требуемым на манипуляции с

данными локальных планировщиков.
Используемый подход обеспечивает практически линейный рост производительности с ростом числа процессоров, до тех пор, пока выполняющиеся на них потоки не взаимодействуют друг с другом, то есть накладные расходы возникают только в том случае, когда они реально необходимы. Некоторую неопределенность вносит только использование спинлоков в реализации DPC очереди, из-за непредсказуемого (хотя и ограниченного) времени возможного ожидания. Для решения этой проблемы может использоваться специальный контейнер в качестве очереди DPC, реализованный без спинлоков и необходимости ожидания, например lock-free циклический буфер или нечто подобное [12]. При использовании такого подхода можно добиться сведения к минимуму накладных расходов на коммуникацию между процессорами.
Асимметричный мультипроцессор работает сходным образом с поправками на невозможность потоков мигрировать, а также на отсутствие глобальной памяти. В качестве очереди DPC используется очередь реализованная внутри синхронизационного слоя планировщика, поскольку обмен "настоящими" DPC между процессорами в асинхронной системе невозможен из-за невозможности вызовов удаленных функций. В качестве функций должны указываться некоторые условные значения, которые каждый процессор интерпретировал бы должным образом. Например, вместо единой функции fx_sched_item_resume выполнение которой можно запросить на любом процессоре, используется целочисленное поле "идентификатор функции", которое хранит условное значение, которое каждый процессор интерпретирует как вызов своего локального экземпляра fx_sched_item_resume. Очередь расположена в памяти, доступ к которой имеют оба процессора, поэтому конкретный адрес указывается во время сборки экземпляров ОС для каждого из процессоров, задействованных в асимметричной системе.
Потоки расположенные в приватной памяти каждого процессора недоступны другому, для удаленного взаимодействия должны использоваться объекты, расположенные в разделяемой памяти, соответственно DPC могут влиять только на эти потоки.

5.8.3 Реализация ожидания

Как и в случае с планировщиком, который может применяться для планирования не только потоков, но и любых других объектов, реализация связи между потоками и примитивами синхронизации также имеет универсальный характер, и может относиться не только к потокам. Поэтому оправдано выделить абстрактный класс "ожидатель" (waiter) в котором содержится необходимая информация для реализации функций ожидания и инкапсулировать этот объект в объект потока (TCB).
Ожидание предполагает существование двух видов сущностей: активных, которые могут быть представлены, например, потоками, и пассивных, которые являются объкетами синхронизации. Между ними существует связь и для каждого участника доступен интерфейс, позволяющий устанавливать и разрывать эту связь. Активные сущности это объекты, которые могут блокироваться и затем продолжать выполнение, примитивы - это объекты, которые выступают в роли ожидаемого объекта. Далее под термином "ожидатель" будет подразумеваться поток, но следует понимать что это частный случай и

это понятие может быть более широким.

Поскольку примитив синхронизации должен иметь информацию обо всех ожидающих его потоках, логично разместить внутри примитива некоторый контейнер, в котором учитывались бы все ожидающие потоки. Эта связь двунаправленная, потому что поток также должен обладать полной информацией о примитивах синхронизации, которых он ожидает. Нужно это в том числе и для того, чтобы контейнеры ожидающих потоков внутри примитивов всегда были в согласованном состоянии: при завершении потока, последний должен себя извлечь из всех таких контейнеров и разорвать собственную связь с примитивами синхронизации. В дополнение к этому, поток может одновременно ожидать нескольких примитивов, например, ожидание с таймаутом логически эквивалентно одновременному ожиданию срабатывания таймера и активации требуемого объекта. В таком случае, если один из объектов переходит в активное состояние и тем самым возобновляет выполнение ожидающего потока, необходимо чтобы поток перестал "ожидать" второй примитив. Эти рассуждения приводят, в конечном итоге, к тому, что, в общем случае, поток также является контейнером, содержащим нечто, что позволяет получить информацию об используемых потоком примитивах синхронизации.

Связь между потоком и примитивом синхронизации удобно представлять в виде специального объекта, называемого "блоком ожидания" (wait block, далее WB). Этот объект содержит ссылки на участвующие в данной конкретной операции ожидания объекты (поток и примитив) и позволяет сторонам узнавать друг о друге. Именно эти объекты и являются главным кандидатом на использование в качестве элементов контейнера как внутри примитива, так и внутри потока.

На Рис. 66 показана ситуация, когда два потока, ждут двух объектов синхронизации. Поток 1 ждет только одного примитива, тогда как поток 2 ожидает обоих. То есть существует связь потока 1 с примитивом 1 и связи потока 2 с примитивами 1 и 2. Каждый поток выделяет необходимое количество блоков ожидания и инициализирует их в соответствии с тем, каких примитивов он собирается ждать. Далее эти блоки помещаются во внутренний контейнер примитива.

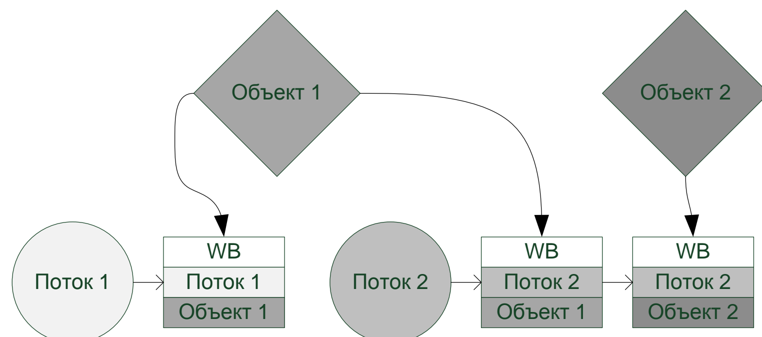

Рис. 66 Использование блоков ожидания

Из рисунка видно, что поле, соответствующее ожидателю, должно всегда быть одинаковым у всех блоков, принадлежащих одному и тому же ожидателю. То же самое справедливо и по отношению к полю объекта: это значение должно совпадать у всех блоков, которые

находятся в контейнере одного и того же объекта синхронизации. Нетрудно увидеть, что теперь можно легко узнать как всех ожидателей данного примитива, так и наоборот, по данному ожидателю можно назвать все объекты, которых он ожидает, что и требовалось.
Если абстрагироваться от потоков, то дополнительно надо пояснить, что будет пониматься в данном случае под "возобновлением". Далее предполагается, что нотификация или возобновление заключается в вызове функции, указатель на которую содержится внутри ожидателя. Ожидатель может указать, сколько таких вызовов он ожидает получить, от 1, в случае если требуется дождаться хотя бы одного из объектов, до N, где N - число используемых блоков ожидания. В последним случае для возобновления выполнения потока необходима активация всех ожидаемых объектов.
Прежде чем приступить к проектированию интерфейса, рассмотрим некоторые свойства данной модели, упрощающие задачу. Хотя, на первый взгляд, кажется, что связи между ожидателем и ожидаемым могут возникать хаотично и в любом порядке на самом деле это не так. Поскольку активной сущностью является поток, либо какой-то его аналог, выполняемый на процессоре, который выполняет инструкции последовательно, можно быть уверенным в том, что операция ожидания будет синхронной для ожидателя. То есть, все вызовы интерфейса ожидателя не могут выполняться параллельно.
Другой важный вопрос заключается в том, кто является владельцем блока ожидания, выделяет под него память и освобождает после того, как ожидание закончилось. На это есть несколько точек зрения, но, так или иначе, выгодно сделать владельцем ожидатель. Только ожидатель знает скольких объектов он собирается ждать, а значит, только он обладает информацией о том, сколько блоков ожидания потребуется.
Наконец, последний штрих: поскольку ожидание может включать в себя перенос данных от ожидателя к ожидателю (например, если примитивом является очередь сообщений) либо от ожидателя к ожидаемому (в случае буферизации данных внутри примитива), может потребоваться дополнительная информация связанная с каждой операцией ожидания. Такая информация называется **атрибутами ожидания** и если она присутствует, она может быть размещена внутри блока ожидания в предусмотренном специально для этого внутреннем поле.
Таким образом, интерфейс должен включать в себя 3 типа объектов: ожидатель, ожидаемый объект, блок ожидания.

```
typedef struct {...} fx_sync_waiter_t;
typedef struct {...} fx_sync_waitable_t;
typedef struct {...} fx_sync_wait_block_t;
```

Так как примитив содержит внутри себя некоторый контейнер, элементы которого могут добавляться и извлекаться асинхронно, а также, возможно, есть состояние самого примитива, которое должно быть синхронизировано с состоянием контейнера, то хорошей идеей будет добавить к интерфейсу функции блокировки и разблокировки примитива. Эти функции должны обеспечивать синхронизацию в том числе и с прерываниями, из которых также могут использоваться примитивы. Обычно, под блокировкой подразумевается захват связанного с объектом спинлока, поэтому блокировка примитива синхронизации подразумевает повышение SPL до максимального.

```
void fx_sync_waitable_lock(fx_sync_waitable_t* waitable);
void fx_sync_waitable_unlock(fx_sync_waitable_t* waitable);
```

Далее, две функции, с помощью которых ожидатель начинает ожидание и завершает его. Операция ожидания начинается с того момента, как блок ожидания помещен в контейнер примитива, поэтому данная функция и создает связь между ожидателем и ожидаемым путем помещения в контейнер примитива предварительно подготовленного и инициализированного блока ожидания. Особо следует отметить, что сразу после вызова этой функции ожидатель, теоретически, может получать нотификации, поэтому он должен быть готов к такому развитию событий. Поскольку данная функция модифицирует контейнер, предполагается, что примитив должен быть заблокирован, поэтому предполагаемый SPL равен максимальному.

```
void fx_sync_wait_start(fx_sync_waitable_t* waitable, fx_sync_wait_block_t* wblock);
```

По завершению ожидания ожидатель может освободить память из-под всех блоков ожидания, поэтому он должен предварительно удостовериться, что все они были извлечены из примитивов, особенно в случае, если использовалось ожидание только одного примитива и выполнение потока возобновилось после нотификации со стороны единственного примитива. Эти гарантии достигаются с помощью следующей функции:

```
unsigned int fx_sync_wait_rollback(fx_sync_waiter_t* waiter);
```

Она должна гарантировать не только тот факт, что объекты извлечены их всех примитивов, но и то, что обращения к их памяти не будет в дальнейшем, так как сразу после вызова данной функции память из-под блоков ожидания может быть освобождена. Поскольку функция может блокировать примитив с помощью спинлоков, предполагается невытесняемый контекст ее вызова.

Теперь обратимся к объекту синхронизации. Конструктор выглядит так:

```
void fx_sync_waitable_init(
      fx_sync_waitable_t* waitable, fx_sync_test_func_t test_and_wait_func);
```

Функция обратного вызова, связанная с примитивом - сердце всего интерфейса. Она используется для того, чтобы проверить состояние примитива, и, при необходимости, начать ожидание с помощью вызова _fx_sync_wait_start. То есть эта функция имеет смысл "атомарно проверить состояние примитива, и если он находится в неактивном состоянии - вставить в очередь переданный блок ожидания". Она имеет следующий прототип:

```
bool test_and_wait(fx_sync_waitable_t* waitable, fx_sync_wait_block_t* wblock ...);
```

Полиморфизм на основе этой функции позволяет релизовать единую функцию ожидания, без размножения этой логики на каждый тип объекта.

Завершает интерфейс функция, предназначенная для уведомлений.

```
void _fx_sync_wait_notify(
  fx_sync_waitable_t* waitable, ...
  fx_sync_wait_block_t* wait_block);
```

Эта функция используется для уведомления потока, соответствующего указаному блоку ожидания. Если в качестве блока ожидания указан NULL, происходит уведомление всех ожидателей.

Интерфейс может показаться сложным, и он таковым и является, но сложность эта проистекает из чрезвычайно сложной задачи, решаемой данным слоем абстракции. С одной стороны, он является сердцем всей операционной системы, потому что практически все функции предоставляемые ядром, так или иначе связаны с синхронизацией, а значит, сводятся, в конечном итоге, к вызовам приведенных методов. С другой стороны, так как сами примитивы синхронизации базируются на данном интерфейсе, его реализация не может использовать никаких средств синхронизации кроме тех, которые предоставляются процессором. Реализация этого интерфейса, особенно в многопроцессорном окружении это довольно нетривиальная задача, несмотря на относительно простую семантику большинства представленных функций.

Представленный интерфейс является каркасом, который должен использоваться как со стороны потоков так и со стороны примитивов.

Функция проверки состояния примитива является довольно сложной вещью, в ней должна быть заключена вся логика работы примитива синхронизации и принятие решения о том, стоит ли данному ожидателю приостанавливать выполненение или нет. Результатом вызова этой функции является значение, указывающее на состояние примитива. Если примитив был в неактивном состоянии (то есть нужно начинать ожидание) функция должна атомарно вставить в очередь примитива блок ожидания.

Типичный примитив синхронизации реализуется следующим образом: главную роль играет функция test_and_wait_func, которая указывается на этапе его инициализации. Если поток выполняет операцию ожидания, это приводит к подготовке нужного количества блоков ожидания и затем вызова такой функции через указатель. После этого, происходит захват блокировки примитива для исключения конкурентного доступа к внутренним структурам данных, которые должны меняться атомарно, далее происходит анализ состояния примитива, и, в случае, если он в активном состоянии (то есть поток может продолжить выполнение), то происходит разблокирование примитива и возврат значения "объект активен". В противном случае используется функция wait_start, предоставляемая каркасом, для вставки переданного в качестве параметра блока ожидания в очередь примитива, с последующим возвратом значения "объект неактивен". По возвращаемому значению вызывающий код должен понять, что блок ожидания был вставлен в очередь и что после выполнения условия ожидания над данным потоком будет вызвана операция возобновления.

Примерная реализация функции примитива приведена ниже:

```
bool test_func(fx_sync_waitable_t* object, fx_sync_wait_block_t* wb, bool wait)
{
  bool wait_satisfied = false;

  /* Блокировка доступа к примитиву. */
  if (...)                        /* Анализ состояния примитива. */
  {
    wait_satisfied = true;        /* Объект активен. Выход без ожидания. */
  }
```

```
  else if(wait)
  {
    /* Объект неактивен. Вставка блока ожидания в очередь. */
    fx_sync_wait_start(object, wb);
  }

  /* Разблокирование доступа к примитиву. */
  return wait_satisfied;
}
```

Другим важным классом функций являются функции активации примитива. В отличие от ожидания, которое выполняется единообразно для всех примитивов, интерфейс функции активации зависит от логики примитива и их может быть и несколько. Пример реализации такой функции приведен ниже:

```
int activate_some_object(fx_sync_primitive_t* object)
{
    fx_sync_waitable_t* const waitable = &(object->waitable);
    ...

    /* Блокировка доступа к примитиву. */

    if(non empty queue)
    {
      fx_sync_wait_block_t* wait_block = /* Получение блока ожидания из очереди. */
      fx_sync_wait_notify(waitable, ..., wait_block);
    }
    else ...

    /* Разблокирование доступа к примитиву. */
    ...
}
```

В отличие от функции test_func, которую должна вызывать на повышенном уровне SPL объемлющая функция, вызовы функций активации примитивов могут происходить в коде пользовательских потоков, поэтому обеспечивать необходимый контекст (блокировка планировщика) они должны сами. Далее происходит блокировка примитива, анализ его состояния, и, если состояние это позволяет (то есть если функция действительно должна активировать объект) и в очереди есть ожидатели, извлекается блок ожидания, соответствующий определенной политике, и вызывается функция нотификации, которая, в конечном итоге, разблокирует ожидателя и он продолжает выполнение. Ожидание завершилось. Схематично описанный процесс показан на Рис. 67.

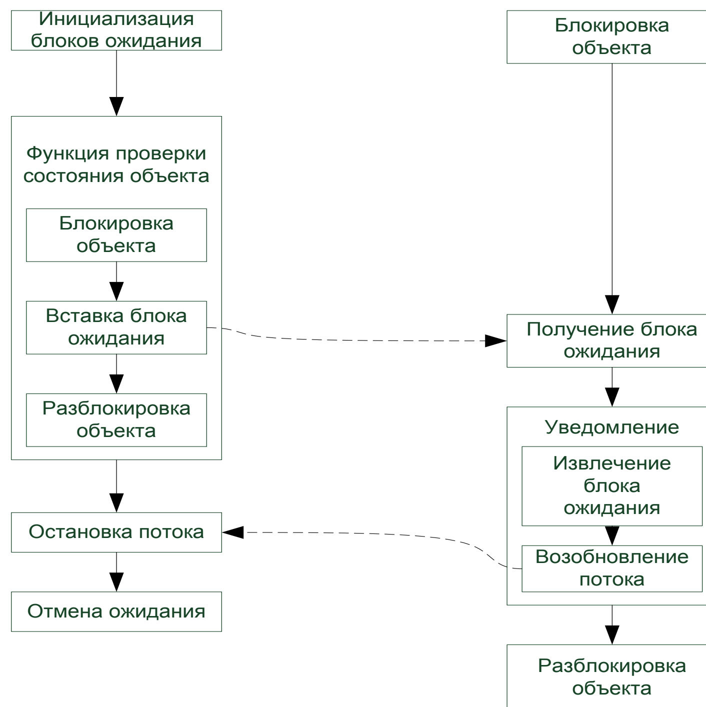

Рис. 67 Алгоритм ожидания

Большинство функций входящих в описанный интерфейс тривиальны. Практически все функции ожидателя, связанные с подготовкой блоков ожидания и самой операции ожидания всегда выполняются в однопоточном, непрерываемом режиме и не требуют никакой синхронизации. Функции, связанные с примитивом, также, в основном, блокируют объект с помощью спинлока, и, таким образом являются синхронизированы между всеми процессорами. Наболее интересны функции активации ожидателя fx_sync_wait_notify и функция завершения ожидания со стороны ожидателя fx_sync_wait_rollback.

Последняя должна извлечь блоки ожидания из очереди объектов. Как блоки ожидания могут оказаться в очереди после того как поток пробуждается? Это может случиться в случае отмены ожидания вследствие иных событий, например таймаута или завершения потока. Для корректного состояния объектов, они, разумеется не должны содержать блоков ожидания из удаленных потоков.

В реализации есть еще одна тонкость. Хотя, по большей части, функции работают в невытесняемом контексте и не требуют явной синхронизации, сама операция ожидания, очевидно, предполагает вытеснение потока, в случае его приостановки. В этот момент возможно также получение потоком асинхронных процедур, которые возобновляют его выполнение и также должны содержать вызов fx_sync_wait_rollback для корректного извлечения блоков ожидания из примитивов, в том случае, если поток находился в состоянии ожидания в момент запроса выполнения асинхронной процедуры. Тогда, после возобновления выполнения потока, он может вызвать операцию fx_sync_wait_rollback еще раз, уже как часть функции ожидания, которая может еще раз

попытаться извлечь блоки ожидания. Для предотвращения такого развития событий в приведенных фрагментах используется обнуление количества активных блоков ожидания внутри функции fx_sync_wait_rollback. После того, как она единожды отработала, повторные вызовы в контексте этого же потока не приведут к фактическим действиям.

Сама нотификация включает в себя извлечение блока ожидания из очереди, установку статуса, и проверку, ожидает ли ожидатель нотификации. Если количество ожидаемых нотификаций меньше, чем было получено ожадетелем, вызывается виртуальная функция уведомления. В случае потоков, это обычно функция, содержащая единственный вызов fx_sched_resume для возобновления выполнения потока.
Разумеется, в "многопроцессорном" варианте все становится на порядок сложнее, потому как сам по себе непрерываемый контекст уже не гарантирует даже существования любого из объектов.
Использование спинлоков, связанных с примитивом, и отказ от глобальных блокировок, создает еще один класс проблем, связанных с существованием самих спинлоков. Если многопроцессорная система использует некоторый набор глобальных блокировок, то, как правило, память под них выделяется на этапе инициализации ядра и далее можно пытаться захватывать эти спинлоки будучи уверенным по крайней мере в том, что обращения в эту память корректны и что по этому адресу действительно находится спинлок. Это роднит их с однопроцессорными системами, которые пользуются запретом прерываний на время выполнения критических секций. Флаг запрета прерываний и способность процессора их запрещать, очевидно, существуют всё время, пока существует сам процессор. Спинлоки же встроенные в удаляемые объекты ядра удаляются вместе с ними, поэтому, прежде чем пытаться их захватывать, нужно быть уверенным в самом существовании объекта. Очевидно, любые попытки добиться гарантии существования спинлока с использованием самого спинлока содержат порочный логический круг и обречены на провал. Для обеспечения таких гарантий нужны некоторые внешние по отношению к спинлоку соглашения.
Отправной точкой в этих изысканиях является тот факт, что в любой ситуации гарантируется существование блока ожидания. Со стороны ожидателя, блок ожидания является локальным, то есть связанным с текущим потоком или его стеком, а потому, пока выполняется код ожидателя, существование его собственной памяти гарантировано. Со стороны примитива все не так очевидно, но, пока примитив заблокирован, функция fx_sync_wait_rollback не может выполниться на стороне ожидателя, а так как выполняется она в контексте с заблокированным планировщиком, поток не может быть завершен до тех пор, пока операция ожидания не завершилась. Если уведомляющий поток под спинлоком примитива увидел блок ожидания от какого-либо объекта, он может быть уверен в существовании этого блока до тех пор, пока он удерживает спинлок. Все это дает возможность использовать память блока ожидания для реализации на ней некоторого lock-free алгоритма, позволяющего корректно реализовать семантику обоих упомянутых функций в многопроцессорной системе.

5.8.4 Реализация потоков

После обслуждения реализации основных составляющих, можно приступить к реализации, собственно, самих потоков.

В структуру потока определяемая HAL и зависящая от аппаратуры структура, представляющая аппаратный контекст процессора. Затем идет объект "элемент планировщика", представляющий данный поток во внутренних структурах данных планировщика, содержащий приоритет и прочие атрибуты потока, а также поле состояния, которое нужно для того, чтобы отличать завершившиеся потоки от работающих и исключения возможности вызова методов над завершенными потоками. Далее в структуру потока следует включить таймер, используемый для реализации функции fx_thread_sleep и контейнер локальных асинхронных процедур для данного потока. Замыкают структуру поля предназначенные для реализации ожидания. Поток сам по себе является сущностью способной ждать, связанные с ожиданием структуры данных инкапсулируются в тип абстрактного ожидателя, от которого наследуется поток. Ожидание завершения потока и его приостановка на заданное время требуют объектов для реализации этого ожидания и в роли таковых выступают два объекта "событие" также встраиваемые в объект потока.

```
typedef struct _fx_thread_t
{
  hal_cpu_context_t hw_context;     /*Аппаратный контекст.*/
  fx_sched_item_t sched_item;       /*Элемент планировщика.*/
  unsigned int state;               /*Состояние потока.*/
  lock_t spinlock;                  /*Спинлок объекта.*/
  fx_timer_t timer;                 /*Таймер для реализации таймаутов.*/
  fx_thread_apc_target_t apcs;      /*Контейнер асинхронных процедур данного потока.*/
  fx_sync_waiter_t waiter;          /*Абстрактный ожидатель.*/
  fx_event_t completion_event;      /*Событие завершения потока.*/
  fx_event_t timer_event;           /*Событие таймаута используемое таймером.*/
  ...
}
fx_thread_t;
```

Каждый процессор в системе должен всегда выполнять какой-то поток, поэтому, на случай, если готовых к выполнению пользовательских потоков нет, на каждом процессоре создается по одному потоку наименьшего приоритета, которые переводят процессор в режим пониженного энергопотребления. Исторически, такие потоки называются Idle-потоками. Принципиально возможно создать систему и без них, так как об отсутствии готовых к выполнению потоков становится известно в обработчике программного прерывания, которое выполняется на стеке прерываний, можно каким-то образом реализовать остановку процессора и перевод его в режим пониженного энергопотребления прямо в самом обработчике. Однако, в зависимости от целей и задач системы может быть полезно использовать idle-потоки для выполнения какой-либо фоновой работы. Таким образом, модуль, реализующий потоки, содержит некоторый контекст, который включает в себя объект idle-потока, а также указатель на текущий выполняемый поток. Все эти данные существуют в виде отдельной копии для каждого процессора.

Обсуждение реализации включает в себя три основные темы: создание/завершение работы потока, приостановка и возобновление выполнения, реализация ожидания.

5.8.4.1 Начало и завершение работы потока

Сердцем всей реализации потоков является, вне всякого сомнения, функция-обработчик программного прерывания, которая реализует планирование и переключение потоков. Ее исходный код, с некоторыми упрощениями, приведен ниже.

```
void software_interrupt_handler(void)
{
  fx_sched_item_t* item = NULL;

  /* Обработка очереди DPC. */
  dpc_handle_queue();

  /* Планирование потоков. */
  item = fx_sched_get_next();

  /* Возврат NULL из функции означает, что состояние планировщика не менялось. */
  if(item)
  {
    fx_thread_t* next = <преобразование sched_item в указатель на поток>;

    /* Если новый поток не равен текущему ...*/
    if(next != context->current_thread)
    {
      /* Обновление указателя на текущий поток и переключение контекста. */
      fx_thread_t* prev = current_thread;
      current_thread = next;
      cpu_context_switch(&next->hw_context, &prev->hw_context);

      /* Установка события завершения потока, если предыдущий поток завершен. */
      ...
    }
  }

  /* Доставка асинхронных процедур для текущего потока. */
  ...
}
```

Упрощения, впрочем, связаны с проверками допущения, отладочными макросами и тому подобным, не имеющим отношения к основной функциональности.

Первым делом, обработчик выясняет текущий контекст, который может меняться в зависимости от текущего процессора. Далее происходит обработка очереди отложенных процедур с последующим вызовом планировщика. Планировщик может быть сделан "ленивым", то есть не производить перепланирование, если не происходило ситуаций, после которых оно могло потребоваться, поэтому ему позволено возвращать NULL в тех случаях, когда текущий поток не изменялся. В противном случае, если было произведено перепланирование, возвращается указатель на элемент планировщика, соответствующий наиболее приоритетному в данный момент потоку. Для получения потока по тому элементу производится приведение типа и проверка действительно ли новый поток не совпадает с предыдущим. Например, в случае понижения приоритета, практически во всех случаях требуется перепланирование и fx_sched_get_next вернет отличное от нуля

значение, однако поток мог остаться самым приоритетным в системе даже и с пониженным приоритетом, поэтому такое действие не должно приводить к переключению потока. Если же все таки переключение требуется, заменяется указатель на текущий поток в контексте модуля, после чего вызывается функция HAL для переключения контекста. После всех этих действий производится анализ наличия у текущего потока асинхронных процедур, если таковые есть, состояние потока модифицируется таким образом, чтобы после возврата в поток начала выполняться асинхронная процедура.

Создание потока довольно тривиально, функция fx_thread_init инициализирует все входящие в структуру объекта пола, и в зависимости от состояния, в котором создается поток (активное или приостановленное) добавляет его в планировщик используя функцию fx_sched_item_add. Гораздо больший интерес представляют функции завершения потока.

Как уже рассказывалось выше, процесс завершения потока должен выполняться синхронно с самим потоком, для чего используются асинхронные вызовы процедур. Для этой цели используется внутренняя функция fx_thread_exit_sync. Эта функция должна использоваться в контексте, в котором удаляемый ею поток не может выполнять никаких инструкций. В многопроцессорной системе это означает, что она должна всегда вызываться в контексте удаляемого потока. В однопроцессорной системе она, в принципе, может быть вызвана и в контексте другого потока, но при этом планировщик должен быть заблокирован. Данная функция удаляет указанный поток из хранилища планировщика, что приводит к тому, что после разблокирования планировщика произойдет переключение контекста.

```
void fx_thread_exit_sync(fx_thread_t* me) {
  ...
  fx_sched_item_remove(fx_thread_as_sched_item(me));
  ...
}
```

Синхронная функция fx_thread_exit при этом содержит только вызов fx_thread_exit_sync и обеспечивает последней необходимый контекст.

В многопроцессорной системе используется асинхронная процедура, выполняемая в контексте целевого потока, которая также блокирует планировщик и выполняет те же самые действия, что и fx_thread_exit.

Таким образом, функция асинхронного удаления потока сводится к проверке статуса потока (на случай, если поток уже был завершен), и, если статус корректный, вставке в очередь асинхронных процедур объекта, который, в конечном итоге, приведет к выполнению функции fx_thread_exit_sync в контексте потока.

Доставка асинхронных процедур потоку происходит во время планирования, после переключения на данный поток. Если поток заблокирован, получается, что асинхронные процедуры не будут ему доставлены до тех пор, пока не завершится ожидание, которое может занимать сколь угодно долгое время. Разумеется, это предотвращает и удаление потока, что нежелательно. Поэтому, после посылки APC, поток принудительно

переводится в состояние готовности. В многопроцессорной системе это включает также рассылку прерываний планировщика процессорам, для того, чтобы перепланирование и доставка асинхронной процедуры гарантированно состоялись в ограниченное время.
После того, как асинхронная процедура доставлена, необходимо модифицировать состояние потока, который мог находиться в состоянии ожидания, в том числе и с таймаутом. Иными словами, может иметь активный таймер и блоки ожидания, находящиеся в контейнерах некоторых примитивов. Для отмены таймера и отмены ожидания используется функция, вызываемая в контексте обраотчика программного прерывания, которая выполняет эти дейтвия с помощью соответствующих функций.

5.8.4.2 Остановка и возобновление выполнения

Остановить выполнение потока можно достаточно просто, для этого нужно только вызвать fx_sched_item_suspend, который в процессе своего выполнения запросит программное прерывание, в результате выполнения которого произойдет переключение контекста. Учитывая тот факт, что в реализуемом API поток может явно приостановить только самого себя, реализация получается предельно простой. Однако, как это часто бывает, подводная часть айсберга оказывается намного большей чем надводная, и приостановка выполнения потока имеет ряд неочевидных последствий с которыми нужно считаться в процессе реализации такой функции. Трудности связаны, главным образом, с асинхронными процедурами. После того, как поток был приостановлен, следующее перепланирование гарантированно приведет к переключению потоков, а значит, для текущего потока никогда не будет произведена доставка асинхронных процедур. То есть, асинхронное удаление потока с помощью функции fx_thread_terminate может закончиться неудачно, что, разумеется, является ошибкой. Одно из решений заключается в том, чтобы использовать спинлок, связанный с объектом потока и изменять состояние потока только удерживая этот спинлок. Тогда, после отправки APC и перехода потока в статус удаляемого, функция приостановки не должна блокировать его выполнение. Если поток находится в состоянии завершения, очевидно, уже не играет никакой роли, будет ли выполнена приостановка или нет, поэтому можно внутри функции fx_thread_suspend можно также вызвать fx_thread_exit и выполнить работу по удалению потока "досрочно".
В том случае, если с состоянием потока все в порядке, вызывается функция fx_sched_item_suspend, которая извлекает его из структур данных планировщика и запрашивает программное прерывание.

```
int fx_thread_suspend(void)
{
  fx_thread_t* me;

  /*Блокировка планировщика...*/
  me = fx_thread_self();

  /* Проверка текущего состояния потока. */
  ...

  if (...) {
    fx_sched_item_suspend(&me->sched_item);
  }
```

```
  ...
  /*Разблокирование планировщика. */
}
```

Возобновление выполнения происходит аналогичным образом. Функция должна только проверить состояние потока на случай, если он уже был завершен (либо находится в процессе завершения).

```
int fx_thread_resume(fx_thread_t* thread)
{
  bool do_resume = false;

  /* Блокировка планировщика...*/
  /* Проверка текущего состояния потока.*/
  ...

  if(...) {
    fx_sched_item_resume(&me->sched_item);
  }

  /*Разблокирование планировщика. */
  ...
}
```

В том случае, если интерфейс предполагает возможность асинхронной приостановки потоков из других потоков, необходимо учитывать также возможность приостановки процесса удаления потока, то есть выполнения асинхронной процедуры, которая работает в контексте потока. Для этого могут использоваться различные подходы, например, выполнение приостановки таким же образом, как и завершения - с помощью асинхронных процедур. В этом случае должна реализовываться некоторая политика приоритезации последних, чтобы в случае наличия одновременных запросов на приостановку и завершение, больший приоритет получало завершение. Для систем реального времени, это, однако, является неоправданным усложнением.

5.8.4.3 *Реализация функций ожидания*

Завершает тему потоков реализация функций ожидания. Хотя изначальный интерфейс предусматривал отмену ожидания только по таймауту, довольно часто возникает ситуация, когда ожидание хотелось бы отменить по произвольному событию. В ОС семейства UNIX для этой цели традиционно используются сигналы, однако их использование в прикладном коде сопряжено с трудностями и не очень удобно. Одним из возможных вариантов, является реализация функции ожидания таким образом, чтобы помимо основного ожидаемого объекта в нее передавался также дополнительный объект "событие", играющий роль триггера, отменяющего ожидание. Таймаут в таком варианте является частным случаем: отменяющее событие устанавливается обработчиком таймера. Примерная реализация функции выглядит следующим образом.

```
int fx_thread_wait_object (fx_sync_waitable_t* object, void* attr, fx_event_t* cancel)
{
  fx_thread_t* me = fx_thread_self();
  fx_sync_wait_block_t wb[2];
```

```
  /*Подготовка блоков ожидания и запрет вытеснения. */
  /*Блокировка планировщика...*/

  do {
    /*Проверка состояния объекта...*/
    object_signaled = object->test_and_wait(object, &(wb[0]), true);

    if(object_signaled) {
      ...
      break;
    }

    /*Провека состояния второго объекта...*/
  }
  while(0);

  if(!wait_skip) {
    fx_sched_item_suspend(&me->sched_item);

    if(waiter satisfied(&me->waiter)) {
      fx_sched_item_resume(&me->sched_item);
    }
    ...

    fx_sched_unlock(...);
    /*Остановка потока происходит в этом месте.*/
    fx_sched_lock(...);
  }

  unsigned rolled_back_wb_num = fx_sync_wait_rollback(&me->waiter);
  /*Получение статуса блоков. */
  return wait_result;
}
```

Функция отводит на стеке место под два блока ожидания, они используются для вставки в очередь основного объекта и отменяющего события. С их расположением связан дискуссионый вопрос. В большинстве случаев расположение блоков ожидания на стеке потоков является оптимальным с той позиции, что используются только тогда, когда реально используется ожидание. Скажем, в простой системе, где используется только синхронизация с помощью fx_thread_suspend/resume без использования примитивов синхронизации, такое их размещение позволит использовать меньше памяти. С другой стороны, в асимметричной многопроцессорной системе лучше расположить в структуре потока, поскольку они могут находиться во внутреннем контейнере примитива синхронизации, разделяемого между процессорами. Очевидно, все находящиеся там блоки должны быть видимы всем процессорам. Этот подход обладает большей универсальностью, но сопряжен с дополнительными расходами памяти.

После инициализации блоков ожидания и подготовки структур данных ожидателя, для каждого из объектов вызывается виртуальная функция, которая анализирует состояние объекта и возможно производит вставку блока ожидания. Объекты проверяются по очереди, если хотя бы один из них находится в активном состоянии, ожидание на этом завершается, в противном случае производится приостановка потока. Здесь также есть один тонкий момент: возобновление выполнения потока в многопроцессорной системе

может произойти сразу после вставки блока ожидания в очередь примитива, то есть еще до того, как поток был приостановлен, соответственно, нотификация с помощью fx_sched_item_resume может быть пропущена. Для того, чтобы избежать такого развития событий, следовало бы производить приостановку потока до анализа состояния примитивов, однако это приводит к проблемам с производительностью, так как даже в том случае, если ждать не требуется, может происходить перепланирование. Решение заключается в том, чтобы проверять состояние ожидателя после приостановки потока, если нотификация объектов все же была произведена до этого момента, можно отменить ожидание. В однопроцессорном случае такое развитие событий невозможно, так как вытеснение заблокировано на время проверки состояния объектов.

Далее происходит разблокирование планировщика, если ожидание не было отменено и ни один из объектов не был активен, в этой точке происходит вытеснение потока и его блокирование. Следующее пробуждение случится по одному из трех событий: активация основного ожидаемого объекта и возобновление выполнения, активация отменяющего события, получение асинхронной процедуры. Возможна так же любая комбинация этих событий. Перед выходом из функции нужно разорвать связь между данным потоком и всеми примитивами, поэтому вызывается функция fx_sync_wait_rollback, которая, помимо прочего, дает точный ответ на вопрос, что именно привело к окончанию ожидания. Если количество отмененных блоков равно нулю, это означает, что они уже были отменены ранее, случиться это могло только в результате доставки асинхронной процедуры. Во всех остальных случаях, функция возвращает 1 или 2, в зависимости от того, сколько объектов были активированы. Анализ поля статуса в отмененных блоках дает ответ на вопрос, который именно из них привел к окончанию ожидания.

В свете всего вышесказанного, реализация функция fx_thread_sleep и fx_thread_join довольно тривиальны: используются события встроенные в структуру потока. Ожидание с таймаутом реализуется также тривиально - событие устанавливаемое таймером передается в функцию ожидания в качестве отменяющего ожидание объекта.

```
int fx_thread_sleep(uint32_t ticks)
{
  int res;
  fx_thread_t* me = fx_thread_self();
  fx_timer_t* timer = &me->timer;
  fx_event_t* timeout_event = &me->timer_event;

  fx_event_reset(timeout_event);
  fx_timer_set_rel(timer, ticks, 0);
  res = fx_thread_wait_object(&timeout_event->waitable, ...);
  fx_timer_cancel(timer);
  return res;
}
```

Важно отметить, что fx_thread_join может вызываться только из одного потока. Поскольку, по соглашению, память из-под объекта потока можно освобождать сразу после того, как эта функция вернула управление, в случае нескольких ожидателей может возникнуть ситуация, когда часть из них уже были нотифицированы, а часть еще нет, поскольку процесс нотификации ожидателей события последовательный. Если один из первых

проснувшихся ожидателей освободит память, используемую под объект потока и его стек, возможно разрушение очереди ожидания с последующим крахом системы.

Существует также важное соглашение, касающееся функций, которые принимают указатель на поток в качестве аргумента и могут быть вызваны из других потоков. Функция fx_thread_join, после своего завершения, гарантирует, что все операции ожидания, в которые мог быть вовлечен поток, завершены и косвенных обращений к памяти потока (через примитивы синхронизации) в дальнейшем не произойдет. Это позволяет корректно завершать выполнение потоков и освобождать занимаемую ими и их стеками память, после того, как функция fx_thread_join вернула управление.

Для упомянутого же класса функций, принимающих аргумент потока непосредственно (это функции fx_thread_terminate, fx_thread_get_params/set_params, fx_thread_resume) разумеется, нельзя этого гарантировать, поскольку ядро ОС не может знать какие еще потоки могут иметь указатель на удаляемый поток. Эта ответственность возлагается на пользователя. Если удаляемый поток может быть аргументом перечисленных функций, пользователь должен гарантировать существование объекта потока все время работы этих функций и только после этого освобождать занимаемую потоком память.

```
int fx_thread_join(fx_thread_t* thread)
{
  int err =  fx_thread_wait_object(&(thread->completion_event.waitable), ...);
  ...
  return err;
}
```

Большинство функций универсальны, их код не зависит от конфигурации, и практически не содержит накладных расходов. Некоторые вопросы вызывает только реализация асинхронных процедур: в однопроцессорных системах они попросту не нужны, но, тем не менее присутствуют в коде.

Здесь можно применить такой же трюк, как и в случае отложенных процедур. Существует две реализации асинхронных процедур, определяющих типы данных и интерфейс, используемые модулем потоков. Одна из них, предназначенная для работы в многопроцессорном окружении, работает так как описано выше: поддерживает очередь асинхронных процедур для каждого потока. Альтернативный вариант, пользуясь тем, что вызовы функций, вставляющих элемент в очередь асинхронных процедур, вызываются в регионах с заблокированным планировщиком, могут вызывать эти процедуры синхронно, поскольку все необходимые гарантии синхронизации с целевым потоком соблюдаются. Получаемая таким образом система становится очень близка к тому, что можно было бы получить если изначально проектировать систему как "однопроцессорную", что и позволяет считать реализацию потоков универсальной.

5.8.5 Реализация таймеров

Таймеры являются одним из наиболее сложных компонентов ядра к реализации которого существует несколько подходов. В простейшем случае применяется единый отсортированный по времени срабатывания список (Рис. 68).

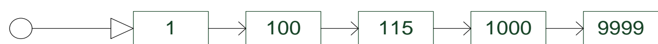

Рис. 68 Реализация таймеров с помощью единого упорядоченного списка

Работает такая реализация довольно тривиально, при установке нового таймера он добавляется в соответствующее место этого списка, а далее обработчик прерывания от аппаратного таймера периодически проверяет время таймера, расположенного в голове списка, и, в случае если оно истекло, вызывает функцию-обработчик таймера. Этот процесс продолжается до тех пор, пока не будут обработаны все истекшие таймеры в списке. Нетрудно увидеть, что, поскольку, обработка происходит в контексте прерывания на повышенном уровне SPL, время реакции системы начинает зависеть от количества активных таймеров в системе и такое ядро теряет право считаться ядром ОСРВ.

Перенос обработки в отложенные процедуры также не особо помогает, потому что, хотя с латентностью обработчиков прерываний все в порядке, потоки и планировщик все равно получают время непредсказуемо, относительно прерываний, которые с ними, возможно, взаимодействуют. Эти соображения приводят к тому, что такие реализации не используются в ОСРВ практически никогда, исключения заслуживает только случай, когда в силу специфики приложения или устройства, таймеров в списке не может быть более, скажем, 5-7. В таком случае, особенно если устройство располагает считанными килобайтами оперативной памяти, использование данного алгоритма может быть оправданным по причине его исключительной простоты и компактности кода.

Следующая группа алгоритмов имеет значительно более широкое распространение и основана на наблюдении, что в каждом тике системного таймера могут сработать далеко не все, а только определенные таймеры, имеющие определенное значение таймаута. Например, если текущий тик кратен 10, то в нем, очевидно, не могут сработать таймеры, у которых значение таймаута кратно 5. Работает это так: создается массив очередей, состоящий из N элементов, в момент добавления нового таймера, вычисляется его позиция в массиве, например как T mod N, где T - его абсолютное время срабатывания, а N - размер массива очередей. Таймаут определяет очередь, в которую будет помещен таймер. N при этом подбирается так, чтобы обеспечить хорошее "перемешивание" таймеров по очередям. Использовать значения вроде 10 или 20 является плохой идеей, так как большинство таймаутов будут кратны этому значению и попадут в нулевую ячейку, поэтому обычно используется небольшое простое число, например 11 или 19. Существует также указатель на текущую позицию в этом массиве, который каждый раз сдвигается по приходу очередного прерывания от таймера.

Поскольку указатель двигается циклически, вся конструкция напоминает циферблат, с которым ассоциированы очереди. Из-за этого сходства семейство таких алгоритмов называется "колесом времени" (timing wheel). Пример такой структуры данных показан на Рис. 69.

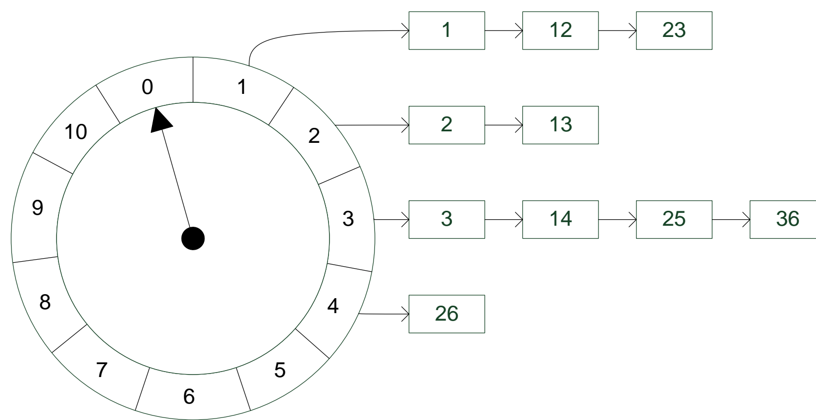

Рис. 69 Реализация таймеров с помощью timing wheel.

Значение абсолютного таймаута однозначно определяет ячейку, в которой может находиться данный таймер: например 25 mod 11 = 3, то есть таймер с таймаутом 25 попадет в ячейку 3. По мере получения тиков системного таймера, указатель циклически обходит "циферблат", проверяя, время срабатывания только у текущей очереди.

Используя такой подход, можно добиться того, что в каждой очереди находятся только таймеры, которые могут сработать в данном тике, поэтому, хотя формально сложность алгоритма остается O(n), на практике, в каждой ячейке будут находиться лишь несколько таймеров. При вставке также происходит анализ только таймеров находящихся в одной из очередей, а не во всех, поэтому алгоритм имеет намного лучшую производительность, чем предыдущий вариант, особенно в случае большого числа таймеров. Однако, несмотря на все плюсы, алгоритм страдает теми же недостатками что и предыдущий: в случае плохого стечения обстоятельств (вероятность этого мала, но не равна нулю) таймеры могут оказаться в одной очереди. Кроме того, в отличие от предыдущего случая, максимальное время реакции которого может быть легко протестировано, в рассматриваемом подходе довольно сложно предсказать заранее, как будут распределяться таймеры по "циферблату" в процессе работы системы, поэтому фактические максимальные задержки могут остаться неизвестными.

Рассмотренные подходы предполагают обработку таймеров непосредственно в обработчиках прерываниях или отложенных процедурах, но эти же алгоритмы могут быть применены и при обработке таймеров в потоках. Возникающий при этом порочный круг (таймеры используют потоки, которые используют таймеры для своей работы) обходится тем, что поток, обрабатыващий таймеры, сам не должен их использовать. В некоторых случаях, таймеры не могут использовать также потоки, обладающие более высоким приоритетом чем тот, который обрабатывает таймеры. В таком случае, обработчик таймерного прерывания в каждом тике пробуждает таймерный поток, который работает со структурами данных, хранящими активные таймеры и вызывает в своем контексте функции-обработчики. Обработка таймеров в обработчике прерывания или в отложенных процедурах страдает теми же недостатками, что и выполнение кода ядра в выделенном потоке, обсуждавшееся ранее: работа неизвестной заранее длительности происходит в невытесняемом контексте, что приводит к неизвестному времени реакции. Перенос

обработки таймеров в потоки позволяет сделать этот процесс вытесняемым и добиться временного детерминизма.
Наиболее простые реализации такого подхода активируют поток обработки таймеров в каждом тике, то есть обработчик прерывания и функция потока выглядят примерно так:

```
void tick_handler(void)
{
  fx_thread_resume(&worker_thread);
}

void timer_worker_thread(void* arg)
{
  while (1)
  {
    fx_thread_suspend();

    while(1)
    {
      /* Извлечение очередного таймера из очереди. */
      ...
      /* Вызов функции-обработчика таймера. */
      ...
    }
  }
}
```

Разумеется, реальные алгоритмы подобного типа более сложны и учитывают варианты, когда обработка очереди может занимать более одного тика, а также исключают возможность потери тиков из-за того, что поток не успел себя приостановить до получения сигнала из прерывания. Способы организации такого взаимодействия между потоками и обработчиками прерываний будет обсуждаться в следующей главе.
Обработка очереди происходит по шагам, а сам поток, обрабатывающий таймеры, остается вытесняемым в процессе обработки очереди. Поскольку на время работы с разделяемыми структурами данных планировщик блокируется, в однопроцессорной системе этого достаточно для синхронизации доступа к очереди.
В отличие от обработки таймеров в непрерываемом контексте, такая реализация требует специальной обработки таймаутов планировщика для реализации round-robin алгоритмов планирования. Таймер, используемый для отсчета кванта потока не может обрабатываться этим потоком, так как вызов функций планировщика в нем будет относиться уже к потоку, обрабатывающему таймеры. Также нельзя не отметить и существенно более слабую производительность: активация потока должна происходить в каждом тике системного таймера, что, в случае высокой частоты последнего, приводит к существенным накладным расходам. Но есть и более существенная проблема: в случае использования упорядоченных очередей, потоки, желающие вставить таймер в очередь, вынуждены блокировать планировщик на время поиска подходящего места в упорядоченной очереди, что вновь приводит к проблемам с непредсказуемыми задержками и временем реакции, так как такой поиск обычно выполняется с заблокированным вытеснением.
Наиболее продвинутые варианты используют реализации, не выполняющие линейный

поиск. Алгоритм похож на колесо времени с той лишь разницей, что списки таймеров не отсортированные. Обработчик прерывания сдвигает указатель, и, в случае, если встретился непустой список, запускает поток, который должен полностью скопировать список в локальную структуру данных, а затем проанализировать каждый таймер который там содержится. Для истекших таймеров вызываются обработчики, а остальные вставляются обратно в очереди. Ценой повышения накладных расходов достигается абсолютный детерминизм таймерного API: все функции вносят фиксированную задержку не зависящую от количества таймеров в системе, а поток-обработчик остается вытесняемым после обработки каждого таймера.

Данный алгоритм является довольно распространенным, но, хотя и решает проблему детерминизма, привносит другие проблемы. Если размер "колеса" слишком велик (порядка сотен и тысяч элементов) это приводит к слишком большим расходам памяти, особенно в системах в ограниченными ресурсами. Если же количество очередей порядка десятка, то при использовании таймеров с большим числом таймаута будет происходить слишком много лишней работы, связанной с отсутствием упорядочения в очередях. Например, если период таймерного тика равен 1 мс (наиболее широко распространенное значение во встроенных ОСРВ), размере очереди порядка 30 элементов, установка таймера на 2-3 секунды (2000-3000 мс) приведет к тому, что такой таймер до своего срабатывания будет около 100 раз извлечен из очереди и вставлен в нее снова (один проход по очереди занимает 30 мс, для истечения таймаута в 3000 мс, требуется 100 проходов по ней). Помимо, собственно, извлечения тамеров, их анализа, и вставки обратно это требует также всех связанных с этим накладных расходов: пробуждение потока, планировщик, сопутствующие переключения контекстов и так далее. В том случае, если таймеров более одного и большинство очередей заняты, поток, обрабатывающий таймеры, должен будет пробуждаться каждую миллисекунду только для того, чтобы сделать по большей части бесполезную работу. Очевидно, при определенном количестве таймеров, система будет 100% времени заниматься пересортировкой очередей и будет лишена возможности выполнить хоть какую-то полезную работу и предоставить время остальным потокам.

Для улучшения производительности была предложена модификация этого алгоритма основанная на иерархии [13]. Далее будет рассматриваться двухуровневая иерархия основанная на минутах и секундах, хотя этот же алгоритм может применяться для произвольного количества уровней. Вместо одного массива очередей, как в вариациях алгоритма колеса времени, используется несколько массивов, каждый из которых представляет определенный уровень иерархии (Рис. 70).

Рис. 70 Структуры данных для иерархического хэширования таймеров

При установке таймера, например, на время 5 минут, 15 секунд, определяется наибольшая степень иерархии, относительно которой используется ненулевое смещение от текущей позиции. В случае использования двухуровневой иерархии это минуты: +5 минут. Таймер помещается в ячейку соответствующего минутам массива, со смещением 5, относительно текущего положения. На каждом тике сдвигается указатель, соответствующий низшему уровню иерархии, в приведенном рисунке, это секунды. После того, как проходит 60 секунд и стрелка, соответствующая секундам, совершает полный оборот, происходит сдвиг "минутной стрелки" за которым следует анализ таймеров, которые расположены в этой ячейке. Поскольку прошла минута, относительно того времени, когда они были вставлены в очередь, все эти таймеры после обработки попадают уже в массив более низкого уровня иерархии. Например, если были вставлены несклько таймеров, со значениями таймаута 5 минут 15 секунд, 5 минут 20 секунд и 5 минут 35 секунд. После того, как прошло 5 минут и "минутная стрелка" перешла на этот список, до срабатывания таймеров остается 5, 20 и 35 секунд соответственно, поэтому они извлекаются из "минутного" массива и размещаются в соотстветствующих ячейках массива нижнего уровня.

Данный алгоритм позволяет практически без накладных расходов использовать таймеры установленные на секунды, минуты и так далее, при этом все операции выполняются за фиксированное время, а обработка таймеров остается вытесняемой на любом этапе своей работы.

Разумеется, используются и другие алгоритмы, в том числе использующие деревья поиска и соотвтстствующую задержку порядка O(log n).

В многопроцессорной системе и без того сложный вопрос становится сложнее еще на порядок. Таймеры могут запускаться и отменяться не только на текущем процессоре и задача синхронизации с обработчиком прерывания теперь не сводится к запрету прерываний. Очевидно, что использование глобальных структур данных таймеров снова приведет к возникновению "бутылочного горлышка", как и в случае с планировщиком, поэтому следует проектировать систему максимально распределенной между процессорами. Как правило, каждый процессор в такой системе имеет свой собственный источник периодических прерываний, из-за возникающей неоднозначности, какому из процессоров направлять прерывания если такой источник только один. В связи с этим, все структуры данных таймеров, а также потоки, если таковые используются, дублируются для каждого процессора, по аналогии с тем, как это делается для планировщика.

Рассмотренные ранее алгоритмы опираются на периодический источник прерываний, с помощью которого отсчитывается время. В рамках борьбы за время автономной работы получают распространение ОСРВ, не использующие периодические прерывания для своей работы (tickless). Основной идеей, стоящей за подобными реализациями, является тот факт, что зачастую системе известно, когда именно необходимо получить очередное прерывание, скажем, если используется структура данных вроде списка или дерева, то известно, когда должно случиться прерывание, которое приведет к срабатыванию таймера. Если же активных таймеров нет, источник прерываний можно просто выключить до того момента, пока они не будут установлены.

Аппаратный таймер программируется каждый раз таким образом, чтобы приход очередного прерывания случился точно в тот момент, когда должен сработать очередной программный таймер. Это усложняет функцию функции fx_timer_get_tick_count, поскольку уже недостаточно просто вернуть значение программного счетчика тиков, необходимо также считать и проанализировать счетчик аппаратного таймера.

В случае использования алгоритмов типа колеса времени задача усложняется, но тоже решаема - известно расстояние в тиках до ближайшего непустого списка. Это не настолько идеальное решение, как в случае, когда можно узнать именно таймаут ближайшего таймера, но тоже неплохое. К тому же, в случае отсутствия таймеров, все реализации могут выключать таймер или программировать его на некоторый максимальный период для того, чтобы иметь возможность реализации функции получения количества тиков.

Несмотря на все эти плюсы, существенно усложняется реализация, хотя бы в таком аспекте, что для каждой платформы требуется наличие слоя абстракции аппаратного таймера, который может иметь как убывающий, так и возрастающий внутренний счетчик. При этом, выигрыш от tickless-режима прямо пропорционален частоте аппаратного таймера. Например, для большинства встроенных систем приемлем период тика порядка 10 мс (100Гц). Обработка прерывания таймера, в течении которой никакие таймеры не срабатывают, происходит примерно за 100-200 инструкций процессора (если какие-то таймеры срабатывают, то, очевидно, прерывание нужно и для tickless-реализаций, разница есть только за счет "лишних" вызовов обработчика). Можно грубо подсчитать, что для типичного микроконтроллера с частотой 100Мгц, способного выполнять одну инструкцию за такт это приводит к накладным расходам в виде "лишних" 20 000 инструкций, которые процессор мог бы не выполнять в tickless-режиме. Учитывая, что в секунду такой процессор выполняет 100 млн инструкций, экономия составляет лишь около 0,02%. То же самое касается и энергосбережения. Ситуация меняется с ростом частоты системного таймера, если, например, приложению необходимы задержки с точностью до десятков микросекунд, это приводит к частоте таймера в 0,1 МГц, что приводит к расходованию на "лишние" таймерные прерывания около 20% всех вычислительных ресурсов процессора, и, соответственно, энергии. С дальнейшим ростом частоты, система может оказаться в принципе не реализуемой с помощью периодических прерываний, так как процессор будет лишен всякой возможности из них выйти и выполнять пользовательские приложения. Таким образом, tickless-режим может помочь добиться большей точности и времени автономной работы устройства, путем существенного усложнения реализации.

Все эти соображения позволяют заключить, что единственно верного решения в данном случае нет, tickless-таймеры могут оказаться хорошим подспорьем в ситуациях, когда время автономной работы стоит на первом месте, либо нужна настолько высокая точность измерения времени, что обработка такого количества периодических прерываний неэффективна. В большинстве же случаев, даже для автономных систем, простота и надежность системы являются главным приоритетом, что позволяет иметь более переносимую и простую реализацию.

5.9 Резюме

Минимальным набором обязатенностей ядра ОС является поддержка потоков, планирования, механизмов синхронизации и программных таймеров. Ядро ОС реализуется на интерфейсах HAL, а потому является переносимым между различными аппаратными архитектурами.

При проектировании ядра ОСРВ нужно принимать во внимание следующие соображения:

- Для минимизации времени реакции оптимально использовать вытесняющее планирование, обеспечивающее фиксированную задержку. В отдельных случаях допустимо использование кооперативного планирования, хотя система построенная с использованием такого планирования не отличается хорошими параметрами времени реакции и напоминает вариации циклического планировщика, ее проще поддерживать и расширять, а кроме того, многие ОС позволяют сосуществование кооперативного и вытесняющего планирования путем настройки приоритетов.
- В случае использования аппаратного обеспечения с ограниченными ресурсами, оптимально использование ядра с унифицированной схемой синхронизации, которая обеспечит наилучшее быстродействие при минимуме накладных расходов. При использовани многопроцессорных систем, а также в случае, когда требуется минимальная задержка при вызове обработчиков прерываний, приоритет следует отдать ОС с сегментированной схемой синхронизации, поскольку она позволяет проще распределять работу между процессорами, путем отправки им DPC, а также в такой системе абсолютное большинство критических секций внутри ядра ОС являются прерываемыми.
- Хотя многопроцессорные системы получают все больше распространение, поведение такой системы не отличается предсказуемостью в том случае, если используется автоматическая балансировка нагрузки между процессорами, поэтому при необходимости использования такой системы предпочтительно использовать подход с выделенными процессорами, когда роль каждого процессора не меняется во время работы и миграций потоков между ними не происходит.
- Помимо потоков и взаимодействий между ними, важным компонентом ОСРВ являются таймеры, поскольку с учетом времени так или иначе связано большинство встроенных приложений. В том случае, когда число таймеров невелико, может использоваться один из простых алгоритмов основанных на сортированных списках. Хотя время их работы зависит от количества таймеров в

системе, и как следствие этого, от количества таймеров зависит также и время реакции, если это число остается в пределах 10-15 для всех режимов работы системы, вносимыми задержками можно пренебречь (при условии быстрой работы самих обработчиков таймеров). В случае же активного использования приложениями таймеров, а также в случае большого количества потоков, активно использующих таймауты, требуется применение одного из алгоритмов обеспечивающих фиксированное время работы любой функции таймерного API.

Система, спроектированная в данном разделе, содержит все необходимые компоненты, свойственные ОСРВ. По сравнению с интерфейсами HAL, предоставляемые интерфейсы гораздо более удобны для использования. Впрочем, возможности синхронизации в такой системе достаточно бедны. В следующем разделе будут подробно рассметрены более сложные примитивы синхронизации, а также проблемы возникающие при их использовании и способы их решения.

[1] Philip Laplante; "Real-time systems design and analysis." Third edition 2004 (p.73)
[2] Э.Таненбаум, А.Вудхалл "Операционные системы. Разработка и реализация" 3-е издание, "Питер" 2007 (с. 22)
[3] Д.Соломон, М.Руссинович, "внутреннее устройство Microsoft Windows" 6-е издание. "Питер 2013 (с.77)
[4] Anderson, Dahlin; "Operating systems. Principles and practice" 2011-2012(p. 138)
[5] ISO/IEC 9899:201x Committee Draft April 12, 2011 N1570; 7.26
[6] Xiaocong Fan, "Real-Time Embedded Systems Design Principles and Engineering Practices" Elsevier 2015
[7] Giorgio Buttazzo, "Hard Real-time computing systems. Predictable scheduling algorithms and applications" 3rd edition. Springer 2011
[8] W. Stallings, "Operating systems. Internals and design principles" 7th edition. Prentice Hall 2012 (Chapter 9)
[9] Liu, Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment" 1979
[10] US Patent №8607249B2: "System and method for efficient concurrent queue implementation" 2013
[11] Varghese, Lauck "Hashed and Hierarchical Timing Wheels: Data structures for the Efficient Implementation of a Timer Facility" 1987
[12] Silas Boyd-Wickizer et al. "An Analysis of Linux Scalability to Many Cores"

6 Взаимодействие потоков

Темы главы:

- Обзор методов низкоуровневой синхронизации
- Универсальные примитивы синхронизации и их реализация
- Проблема инверсии приоритета
- Часто используемые специальные примитивы

6.1 Задача синхронизации

Ядро ОСРВ предоставляет приложениям возможность создания потоков, которые выполняются независимо друг от друга. При этом каждый поток работает со своей собственной непредсказуемой скоростью, обусловленной частотой и длительностью его вытеснения другими потоками, прерываниями и так далее, которые возникают непредсказуемо. Если бы каждый поток использовал только свои локальные ресурсы, такие как стек, то никакой проблемы бы не возникло. В реальных системах многопоточные приложения обычно решают какую-то общую задачу, поэтому, так или иначе, потокам необходимо взаимодействовать друг с другом, так как существует некоторое разделяемое состояние.

Хотя понятие синхронизации обладает более широким смыслом, с точки зрения ядра ОСРВ, под этим подразумевается обычно два основных типа взаимодействия между потоками: **взаимное исключение** (mutual exclusion) доступа к разделяемым данным для обеспечения согласованности этих данных, и **уведомление** другого потока о некоторых событиях (notification). В качестве примера первого можно привести обеспечение монопольного доступа к ресурсу, такому как глобальные структуры данных или устройства, а в качестве примера второго - уведомление потока, обрабатывающего сетевые пакеты о приходе нового пакета. Уведомление также называется **условной синхронизацией**, поскольку часто подразумевает существование некоторого условия, о котором следует уведомлять поток.

Используемый метод синхронизации должен обладать двумя фундаментальными свойствами: **надежностью** (safety) и **жизнеспособностью** (liveness), определение которых можно найти в [1]. Говоря же неформально, первое свойство гарантирует, что, грубо говоря, "при правильном использовании не может произойти ничего катастрофического". Второе свойство, жизнеспособность, должно гарантировать, что система в целом будет совершать прогресс, то есть синхронизация не должна приводить, например, к ситуации, когда никаких ошибок не произошло, но продолжение работы невозможно из-за входа системы в состояние, из которого нет выхода. Для иллюстрации этих свойств можно привести несколько примеров.

После запуска многопроцессорной системы каждый процессор должен определить свой номер и настроить стек. Допустим, существует массив, в котором перечислены указатели на стеки каждого из процессоров. Если не использовать атомарные инструкции, код для каждого из процессоров мог бы выглядеть примерно так:

```
static volatile unsigned cpu_num = 0;
```

```
void cpu_startup(void)
{
  const unsigned my_id = cpu_num++;
  set_interrupt_stack(my_id);
}
```

Для инкремента был бы сгенерирован код вида: "прочитать в регистр", "прибавить 1", "записать". В некоторых случаях, такой код вполне мог бы работать, однако в том случае, если все процессоры будут работать строго одновременно, вначале каждый из них прочитает переменную и получит my_id = 0, далее увеличит ее, получив 1, после чего все процессоры попытались бы записать в переменную 1. При дальнейшей попытке использовать один и тот же стек для прерываний произошла бы фатальная ошибка: разрушение структур данных и переход выполняемого алгоритма в неопределенное состояние.
Под жизнеспособностью подразумевается отсутствие взаимных блокировок и пробуксовки (starvation). Например, если два потока пытаются захватить два спинлока в разном порядке:

```
void thread1(...)
{
  spinlock_acquire(&spinlock1);
  spinlock_acquire(&spinlock2);
}

void thread2(...)
{
  spinlock_acquire(&spinlock2);
  spinlock_acquire(&spinlock1);
}
```

Может возникнуть ситуация, когда оба потока одновременно захватывают первые спинлоки, в таком случае каждый из них ждет другого. Система пришла в состояние взаимоблокировки, из которого нет выхода и дальнейшая работа невозможна. Хотя в синтетических примерах решение очевидно - нужно захывативать спинлоки всегда в одном порядке и проблемы не будет, в реальных задачах необходимо тщательное проектирование для того, чтобы можно было гарантировать отсутствие взаимоблокировок, которые могут возникать неявно. Например, ранее рассматривался каркас для примитивов синхронизации, который подразумевал существование ожидателя и ожидаемого, а также связи между ними, выражаемую блоком ожидания. Если предположить, что каждый из этих объектов имел бы собственный спинлок, то, например, для отмены ожидания нужно было бы захватить как спинлок ожидателя, так и ожидаемого, поскольку это действие должно быть атомарно с точки зрения и того и другого. Учитывая тот факт, что отмена ожидания может быть инициирована как с одной, так и с другой стороны (уход ожидателя по таймауту либо активация объекта) получается, что возможна ситуация, когда отмена будет инициирована одновременно с точки зрения как потока, так и объекта синхронизации. В этом случае, каждый из объектов вначале захватывает свой собственный спинлок, после чего пытается захватить второй, что и

приводит к возможности взаимной блокировки.
Другой пример: спинлоком может быть защищено состояние планировщика на каждом из процессоров (спинлок связан с процессором). При миграции потоков может потребоваться захват обоих спинлоков, для процессора откуда поток извлекается и процессора, на который он мигрирует. При роковом стечени обстоятельств, когда будет инициирована миграция двух потоков в противоположных направлениях, вновь возникает взаимоблокировка. К сожалению, подобные ситуации возникают чаще чем хотелось бы. В том случае, когда, как в случае с процессорами, оба спинлока могут быть известны заранее, применяется подход, когда они захватываются в порядке, определяемым их адресом. В первом же случае, когда второй спинлок неизвестен до захвата первого, такой подход, очевидно, не сработает и требуется более сложное решение для конкретного алгоритма.
Близким к взаимоблокировке является понятие **пробуксовки** (starvation). В этом случае система работает, но прогресса в решении задачи при этом не происходит. Проиллюстрировать это можно предыдущим примером, снабдив спинлоки некоторым таймаутом (например, количеством прокруток в цикле после которого считается, что ожидание завершилось неудачей). Может возникнуть ситуация с циклическими попытками захвата спинлока, при этом оба потока продолжают работать и никаких катастрофических последствий не происходит, тем не менее, полезной работы эти потоки не совершают.
Задачи синхронизации постоянно возникают не только в области программирования, но и в реальной жизни. Например, необходимо обеспечивать синхронизацию доступа к пешеходному переходу со сторону пешеходов и автомобилей, доступ самолетов к авиатрассам, обеспечивать поездам монопольный доступ к участкам железной дороги и так далее и тому подобное. Есть также много бытовых примеров, которые стали каноническими и приводятся во многих книгах, посвященных программированию: "общий сад" (shared yard) [2] и "слишком много молока" (Too much milk) [3]. В первой речь идет о двух соседях, которые имеют большой общий сад и содержащих кота и собаку, которых необходимо время от времени выпускать в сад, но нельзя допустить их одновременного нахождения там. Во второй задаче рассказывается о двух студентах, которые должны поддерживать заполненным холодильник, причем таким образом, чтобы продукты закупались только тогда, когда они отсутствуют. Иными словам, во всех ситуациях, где присутствует некоторый общий для нескольких пользователей ресурс, всегда неизбежно возникает проблема синхронизации доступа к этому ресурсу.
Если свойства надежности или жизнеспособности для механизма синхронизации не выполняются (в зависимости от чередования потоков нарушаюся соглашения, которые определены для задачи), такой механизм называется содержащим **гонки** или **условия гонок** (race condition). То есть успешность выполнения алгоритма зависит от того, успел ли какой-то поток сделать свою работу вовремя. Поскольку у пользователя есть ограниченные возможности влияния на чередование потоков, получается, что всё зависит от воли случая, что, разумеется, недопустимо.
Помимо разделения механизмов синхронизации на два типа: взаимное исключение и уведомление, используется также дополнительная классификация, основанная на

количестве взаимодействующих потоков. Например, "один к одному", когда один поток желает уведомить другой о событии; "один ко многим", когда один поток уведомляет несколько других; "многие к одному" - такие взаимодействия, как обращение многих клиентов к серверу, и, наконец, "многие ко многим", данный тип взаимодействия используется при коллективной обработке данных, например получение задач через сеть с нескольких сетевых адаптеров и распределение этих задач на некоторое количество процессоров.

6.2 Низкоуровневые механизмы синхронизации

Прежде чем приступить к рассмотрению примитивов синхронизации, полезно разобраться с более низкоуровневыми механизмами, которые предоставляются напрямую аппаратурой или HAL. Хотя использование примитивов синхронизации предпочтительно из-за их универсальности, в некоторых случаях (в основном связанных с достижением максимальной производительности) целесообразно пользоваться более быстрыми механизмами.

6.2.1 Запрет прерываний и вытеснения

Самый простой способ реализации взаимного исключения в однопроцссорных системах - запрет прерываний. В терминах HAL это означает повышение SPL до уровня SYNC.

```
spl_t prev_spl = raise_spl(SPL_SYNC);
...
lower_spl(prev_spl);
```

Этот способ обеспечивает взаимное исключение как с потоками, так и с обработчиками прерываний. Из-за того, что в большинстве процессоров запрет прерываний выполняется всего одной инструкцией, этот метод отличается исключительной скоростью и отсутствием накладных расходов. Несмотря на то, что он не подходит для многопроцессорных систем, метод нашел широкое применение в системах однопроцессорных. Многие системы, особенно те, в которых не используется ОСРВ, используют этот метод в качестве основного механизма синхронизации. Другой проблемой данного метода является **гранулярность** синхронизации. Поскольку запрет прерываний влияет на весь процессор, то использование такого подхода влияет не только на объекты, вовлеченные во взаимодействие, но и на всё, что может выполняться на данном процессоре. Это можно сравнить с системой управления светофорами в городе, которая имеет всего два положения "зеленый свет всем, кто едет с севера на юг и обратно, красный - тем кто едет с запада на восток и обратно" и его противоположность. Очевидно, что если в городе всего 5-6 улиц, данное упрощение может быть оправданным, но не более того. Таким образом, под **тонкой гранулярностью** (fine grained) подразумевают такие механизмы, которые влияют только на объекты, вовлеченные во взаимодействие, в противопложность этому ставится **грубая гранулярность** (coarse grained), которая подразумевают влияние в том числе и на те объекты, которые не вовлечены во взаимодействие, вплоть до влияния на всю систему в целом.
В системах реального времени использование SPL для синхронизации сопряжено также с

проблемой влияния на время реакции системы: если время выполнения критической секции нельзя предсказать заранее, то, очевидно, такая система не может реагировать на прерывания с заданной наперед задержкой.

6.2.2 Неблокирующая синхронизация

Для синхронизации как однопроцессорных так и многопроцессорных систем можно использовать атомарные инструкции, предоставляемые процессором. Если их набор достаточно универсален, с их помощью, теоретически, можно решить любую прикладную задачу. Поскольку использование этих инструкций не предполагает блокировки потока и ожидания, то такой тип синхронизации называется неблокирующим.

Среди главных преимуществ этого типа синхронизации можно отметить универсальность и способность работать в любом контексте. Например, может использоваться для синхронизации между обработчиками прерываний (где ожидание и приостановка выполнения запрещены) и потоками. Существует ряд проверенных алгоритмов, таких как, например, циклические буферы или атомарные списки [4], которые могут использоваться в приложениях. Однако за пределами нескольких проверенных шаблонов и алгоритмов, использование неблокирующей синхронизации сопряжено с трудностями. Многие современные процессоры не являются последовательно консистентными, поэтому неблокирующая синхронизация подразуемвает также правильное использование барьеров памяти и прочего. В этом случае разработка неблокирующего алгоритма превращается в настоящую исследовательскую задачу, а внесение в него изменений может потребовать решения задачи практически сначала. Из-за сложностей в разработке и поддержке неблокирующая синхронизация не завоевала широкого распространения. Подробнее о методах разработки и анализа неблокирующих алгоритмов можно прочесть в [2].

6.2.3 Использование спинлоков

Единственным средством взаимного исключения для многопроцессорных систем, предоставляемым HAL являются спинлоки.

```
spinlock_acquire_at_level(&spinlock, SPL_SYNC);
...
spinlock_release(&spinlock);
```

Из-за особенностей реализации, на использование спинлоков накладывается множество ограничений, таких как возможность их захвата только в регионах, где запрещено вытеснение, поэтому на них автоматически распространяются ограничения на длину критических секций, свойственные для запрета прерываний. Во многих реализациях захват спинлока предполагает также и запрет прерываний, для исключения возможности попытки захвата спинлока обработчиком прерывания, после того как он был захвачен потоком, то есть для предотвращения взаимоблокировки.

Поскольку спинлок предполагает активное ожидание в виде цикла, с ними связано также много проблем с энергопотреблением. Тем не менее, в простых системах, а также в том случае, когда длина критической секции ограничена, при соблюдении ограничений, связанных с невытесняемым контекстом, спинлоки являются мощным средством

синхронизации и с оговорками могут использоваться во встроенных приложениях, так как вносимые ими накладные расходы невелики.

6.2.4 Использование приоритетов

В том случае, если используется известная заранее политика планирования потоков, например FIFO, синхронизация между потоками может быть реализована с помощью этой политики. Запрет прерываний и прочие подобные методы, используются для того, чтобы предотвратить вытеснение потока во время выполнения критической секции, но те же самые гарантии могт быть получены путем временного повышения приоритета потока. Например, если система содержит несколько потоков и известно, что приоритет каждого из них не более некоторого N, то, с помощью повышения собственного приоритета до N, можно гарантировать отсутствие вытеснения для FIFO планировщика (Рис. 71).

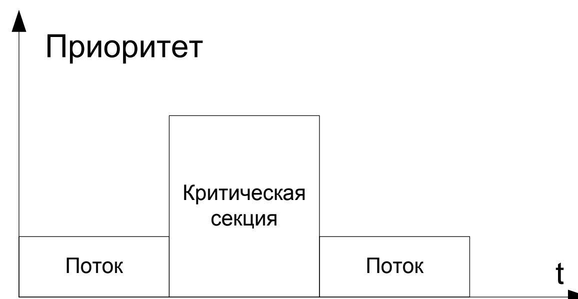

Рис. 71 Критическая секция с повышенным приоритетом потока

При этом, если общие данные существуют только для группы потоков, допустимо существование в системе и более приоритетных потоков. В отличие от запрета прерываний, подобный подход позволяет добиться независимости времени реакции от длины критических секций.

Метод довольно эффективен, если политика планирования известна и не будет меняться, а также в случае небольшого количества потоков. Если планировщик содержит оптимизации, позволяющие избавиться от перепланирования при повышении приоритета, то накладные расходы также невелики.

К сожалению, в многопроцессорной системе этот метод страдает от тех же проблем, что и запрет прерываний. Кроме того, неявные предположения о политике планирования не без оснований считаются плохим стилем программирования и приводят к плохой переносимости кода между различными ОС. При портировании приложения на другую ОС оно может оказаться неработоспособным, причем найти причину ошибки может быть непросто.

Помимо использования планировщика, могут применяться и другие методы, эксплуатирующие определенные гарантии синхронизации, которые предоставляются механизмами ядра ОС. Например, поскольку гарантируется, что отложенные процедуры DPC выполняются на одном процессоре строго последовательно, их также можно использовать в целях достижения взаимного исключения даже в многопроцессорной системе.

6.2.5 Синхронизация с прямым уведомлением

Рассмотренные методы в основном используются для взаимного исключения, вместе с

тем, часто встроенная система представляет собой набор обработчиков внешних событий, для чего требуется обеспечить возможность обработчика прерывания уведомить поток о возникающих событиях. Такой тип взаимодействия также может быть реализован с помощью интерфейса потоков без использования дополнительных объектов и называется **синхронизацией с прямым уведомлением** (direct notification). В этом случае поток использует SPL и функции приостановки для проверки условия и приостановки собственного выполнения если работы нет, а обработчик прерывания использует функцию возобновления выполнения потока, как способ его уведомления о событиях. Многие устройства не предполагают возникновение новых прерываний, до обработки предыдущих, поэтому уведомление обрабатывающего потока из обработчика может выглядеть следующим образом:

```
void ISR(void* arg) {
  /* Отключение прерываний от устройства. */
  ...
  dpc_insert(&dpc, DPC, NULL);
}

void DPC(void*arg) {
  thread_resume(&thread);
}

void Thread(void* arg) {
  do  {
   /* Обработка прерывания. */
   ...

   sched_lock(...);
   /* Включение прерываний от устройства. */
   ...
   thread_suspend();
   sched_unlock(...);
  }
  while(1);
}
```

Обработчик прерывания маскирует прерывания от устройства (некоторые устройства делают это автоматически) и ставит в очередь DPC, которая возобновляет выполнение потока. Поток, в свою очередь, обрабатывает прерывание, после чего входит в критическую область с повышенным SPL, где размаскирует данный вектор и останавливает сам себя. Прерывание может возникать в любой момент после его размаскировки, в том числе и между размаскировкой и остановкой потока, но, поскольку выполнение DPC запрещено на уровне SPL планировщика, фактическое исполнение функции thread_resume произойдет только после выхода из критической секции, то есть после выполнения thread_suspend, и, таким образом сигнал не будет потерян. Эти же рассуждения остаются справедливы и для унифицированной схемы синхронизации, код остается корректен и в этом случае.
В том случае, если устройство допускает возникновение новых прерываний до того как были полностью обработаны предыдущие (так устроены, например, многие сетевые адаптеры), возникает проблема: если прерывание возникло до входа в критическую

секцию, DPC также будет выполнена, следовательно thread_resume будет выполнена для потока, который не находится в состоянии ожидания, а значит сигнал будет потерян. Для решения этой проблемы можно немного усложнить приведенный код.

```
void ISR(void* arg) {
  if(atomic_add(&counter, 1) == 0) {
    dpc_insert(&dpc, DPC, NULL);
  }
}

void DPC(void*arg) {
  thread_resume(&thread);
}

void Thread(void* arg) {
  do  {
    /* Обработка прерывания. */
    ...

    sched_lock(...);
    if(atomic_add(&counter, -1) == 1) {
      thread_suspend();
    }
    sched_unlock(...);
  }
  while(1);
}
```

Используется счетчик прерываний (переменная counter). Если прерывание приходит в момент, когда поток не был в критической секции, этот факт будет учтен в переменной. Когда, после декремента счетчика, поток обнаружит, что счетчик отличен от нуля, остановка потока не выполняется и он совершает еще одну итерацию. Если же прерывание произошло в момент между декрементом счетчика и остановкой потока, применяются те же самые рассуждения, что и в предыдущем случае. С некоторыми оговорками данный метод применим и в многопроцессорных системах, требуется только обеспечить выполнение DPC на том же процессоре, что и обрабатывающий прерывания поток.

Данный подход отличается быстродействем и применяется во многих реальных приложениях. Тем не менее, он обладает и рядом недостатков. Во-первых, он не отличается простотой, использование атомарных операций требует тщательного продумывания возможных сценариев поведения системы, а возможные ошибки довольно трудно локализовать. Во-вторых, невозможно указать таймаут для ожидания, что может быть важно в системах повышенной надежности.

Описанный выше механизм, когда один агент (в данном случае, обработчик прерывания) сигнализирует другому (потоку) о наличии некоторой работы и продолжает выполнение без ожидания ответа называется **односторонним рандеву** (unilateral rendezvous).

6.3 События

Возвращаясь к предыдущему примеру, с уведомлением потока обработчиком прерывания (случай, когда обработчик маскирует дальнейшие прерывания) можно

избавить пользователя от необходимости создавать в коде критические секции с заблокированным планировщиком. Для этого можно применить простейший объект синхронизации - объект "событие".
Событие имеет внутреннее состояние в виде бинарного значения, указывающего на то, находится ли объект в активном состоянии или нет, а также 3 основных метода, не считая конструктора и деструктора:

```
int event_set(event_t* event);
int event_reset(event_t* event);
int event_timedwait(event_t* event, uint32_t timeout);
```

С помощью первых двух можно менять состояние события, третья функция используется для ожидания. Всякий поток, при попытке ожидания события, которое находится в неактивном состоянии, блокируется до тех пор, пока событие не станет активным. Иными словами, до тех пор, пока какой-то другой поток не выполнит операцию event_set. После этого объект остается в активном состоянии до тех пор, пока это состояние не будет сброшено вызовом event_reset. Если событие активно, попытки его подождать завершаются сразу же, то есть блокирования потока не происходит, а состояние объекта не меняется. Хотя в основном этот функционал используется для уведомления о некоторых событиях, откуда и происходит название примитива, объект можно также сравнить с дверью, которую можно открывать и закрывать, а попытка пройти через дверь, приводит к ожиданию до тех пор, пока дверь не будет открыта.
Задачу с уведомлением потока о прерываниях можно реализовать более просто с использованием такого объекта:

```
static event_t event;

void ISR(void* arg) {
  /* Отключение прерываний от устройства. */
  ...
  dpc_insert(&dpc, DPC, NULL);
}

void DPC(void*arg) {
  event_set(&event);
}

void Thread(void* arg) {
  do  {

   int error = event_timedwait(&event, ...);
   ...
   fx_event_reset(&event);

   /* Обработка прерывания. */
   ...

   /* Включение прерываний от устройства. */
   ...
  }
  while(1);
}
```

В отличие от использования прямого уведомления, с использованием события можно ожидать прерывания с таумаутом, что может быть полезно. К сожалению, данный пример не будет работать, в случае, если новые прерывания могут приходить во время обработки текущего, поскольку попытки установить в активное состояние событие, которое уже установлено, теряются. Также события не подходят для реализации взаимного исключения. Тем не менее, несмотря на эти недостатки, объект представлен во многих ОС (например, в Windows он называется Notification event) и оказывается полезным во многих других ситуациях. Главное достоинство события в том, что это широковещательный (то есть переводящий в состояние готовности все ожидающие потоки, синхронизация типа 1 к N) объект, то есть может использоваться в том случае, когда количество ожидающих не известно заранее.
В качестве примера такой ситуации можно рассмотреть следующую задачу: как и ядро ОС, приложение может содержать функционал, похожий на таймеры или DPC, когда объект содержит функции, последовательно обрабатываемые в некотором контексте. При необходимости отменить выполнение такого объекта, недостаточно только извлечь его из очереди, требуется еще и обеспечить гарантию, что функция-обработчик не выполняется в данный момент и ресурсы используемые ей можно освобождать. Поместить отдельный примитив синхронизации в каждый объект может быть слишком накладным, особенно в случае, если отмены происходят нечасто. Поэтому можно применить следующий подход: функция обработки сбрасывает событие на время обработки объектов:

```
event_reset(&event);

while(...) {
  /* извлечение и обработка объектов. */
}

event_set(&event);
```

А другие потоки, после извлечения объекта из очереди, дожидаются активного состояния события. Активное состояние события означает, что обработка очереди в данный момент не происходит. Поскольку заранее неизвестно, сколько именно потоков приходится пробуждать на каждой установке события, такая логика работы примитива это как раз то, что требуется, реализация такого же поведения с использованием других объектов синхронизации была бы более сложна.
Два события могут использоваться для синхронизации между клиентом и сервером. В этом случае каждый из них использует свой объектдля ожидания, и второй - для уведомления другой стороны. Такой тип взаимодействия предусматривает не только уведомление о каком-либо запросе, но и ожидание ответа, поэтому называется **двухсторонним рандеву** (bilateral rendezvous).
Несмотря на то, что в отдельных случаях использование событий оправдано, в целом они не завоевали широкого применения в программировании из-за перечисленных ограничений.

6.4 Семафоры

Подлинно универсальным объектом синхронизации, в том смысле, что он может использоваться как для реализации взаимного исключения, так и для уведомлений, является предложенный в 1965 году **семафор**. Название происходит от железнодорожного семафора, который обеспечивает нахождение только одного поезда на данном участке пути. Абстрактный семафор, используемый в программировании, работает примерно таким же образом.

Семафор представляет собой счетчик и имеет два основных метода: post и wait. В разных реализациях эти методы называются по-разному (в оригинале P и V). Один из методов (post) используется для того, чтобы увеличить счетчик, а другой для уменьшения и ожидания. Если поток пытается уменьшить счетчик семфора, а тот равен нулю, поток блокируется до тех пор, пока какой-то другой поток или обработчик прерывания не выполнит увеличение счетчика, после чего заблокированный поток продолжает выполнение, предварительно уменьшив счетчик.

Интерфейс семфора, не считая таких функций как конструктор и деструктор, выглядит так:

```
int sem_post(sem_t* sem);
int sem_timedwait(sem_t* sem, uint32_t timeout);
```

Подобным образом он выглядит и в POSIX и в большинстве других ОС.

Тот факт, что счетчик семафора атомарно уменьшается в результате его ожидания, позволяет реализовать взаимное исключение в том случае, если начальное состояние счетчика было 1. Критическая секция при этом выглядит так:

```
/* Если поток пробуждается без ошибок, значит значение счетчика семафора стало равно
нулю и любой другой поток попытавшийся войти в критическую секцию, защищаемую
семафором, будет заблокирован. */
int error = sem_timedwait(&sem, <таймаут>);
if(!error)
{
  /* Код критической секции. */
  ...

  /* Увеличение счетчика семафора и пробуждение одного из ожидающих потоков, если
таковые есть. */
  sem_post(&sem);
}
```

Нотификация с использованием семафора выполняется сходным образом. Из-за наличия счетчика, а не бинарного значения, как у событий, возможны нотификации в то время, когда обрабатываются предыдущие, и они не будут потеряны.

```
static sem_t sem;

void ISR(void* arg) {
  ...
  dpc_insert(&dpc, DPC, NULL);
}
```

```
void DPC(void*arg) {
  /* Включение прерываний от устройства. */
  sem_post(&sem);
}

void Thread(void* arg) {
  do  {
   int error = sem_timedwait(&sem, TIMEOUT);

   /* Обработка прерывания. */
   ...
  }
  while(1);
}
```

В приведенном примере, обработка прерываний может быть довольно легко распределена между процессорами, в том случае, если на каждом из них запущено по потоку, ожидающему одного и того же семафора. Тогда последовательные увеличения счетчика семафора будут пробуждать потоки на других процессорах.
Семафоры являются одним из старейших методов синхронизации и поддерживаются практически во всех ОС, как встраиваемых так и общего назначения. Если задачу можно решить с помощью семафоров, следует отдать им предпочтение, поскольку они поддерживаются во всех открытых стандартах и исключают возникновение каких-либо проблем с портированием. Из-за того, что в результате одной нотификации пробуждается всегда только один поток, семафоры также завоевало широкое распространение в области разработки ПО реального времени, так как все операции с семафором выполняются за фиксированное время, в отличие от операций с событием, время выполнения которого зависит от числа ожидающих потоков.
Помимо рассмотренных выше типичных задач взаимного исключения и уведомления, с семафорами связан также ряд задач, считающихся классическими, таких как, например, "проблема обедающих философов". Исчерпывающий список таких задач можно найти в [5].
Помимо широко известных алгоритмов работы основного API, в реализации семафоров имеются особенности, которые могут различаться в разных ОС, но тем не менее важны для прикладного программирования и должны быть учтены. Одной из таких особенностей является механизм обработки очереди: в каком порядке происходит пробуждение ожидающих потоков и по какому принципу выбирается тот единственный, который следует разбудить даным вызовом увеличения счетчика семафора. На первый взгляд кажется, что, поскольку речь идет об ОСРВ, то логично всегда выбирать поток, имеющий наибольший приоритет среди ожидающих, такое поведение и в самом деле предпочтительно для некоторых задач. Тем не менее, при этом может следующая проблема: если высокоприоритетный поток после его нотификации успевает вновь начать ожидание семафора, до того, как произойдет следующая нотификация, в следующий раз будет возобновлен опять этот же самый поток, то есть другие ожидающие потоки могут вообще никогда не получить управление. Кроме того, поиск приоритетного потока в очереди сам по себе может быть затратен, он требует либо просмотра всей очереди, либо специальной структуры данных, позволяющей найти высокоприоритетный поток без

анализа всех ожидающих. Эти соображения приводят к тому, что в некоторых случаях, по соображениям производительности и более равномерного распределения работы по ожидающим потокам, предпочтительно использовать FIFO-политику нотификации потоков: те из них, которые начали ожидание раньше, будут возобновлены первыми. В зависимости от ситуации, бывают нужны семафоры обоих типов, поэтому многие ОС (в том числе и FX-RTOS) позволяют при создании семафора указать, какую политику обработки очереди следует использовать, что позволяет сочетать в рамках одной системы семафоры различных типов.
Что касается реализации семафоров, то она выглядит следующим образом: Объект семафора обычно включает в себя счетчик, некоторое представление очереди ожидающих потоков, а также дополнительные поля, такие как спинлоки для низкоуровневой синхронизации доступа к очереди и счетчику.

```
typedef struct _fx_sem_t {
  fx_sync_waitable_t wait_queue;
  uint32_t semaphore;
  ...
}
fx_sem_t;
```

В отличие от событий, у которого ожидание объекта не отражается на его состоянии, ожидание семафора приводит к изменению его счетчика (примитивы, работающие таким образом называются примитивами с разрушающим ожиданием). Функция проверки состояния семафора, которая используется для ожидания, должна атомарно проверить состояние счетчика, если он более нуля, происходит декремент и выход из функции без ожидания, в противном случае поток должен быть заблокирован и поставлен в очередь.

```
bool sem_test(fx_sync_waitable_t* object, fx_sync_wait_block_t* wb, ...) {
  fx_sem_t* sem_obj = /* Приведение типа от переменной object. */
  bool wait_satisfied = false;

  /* Захват спинлока объекта. */
  ...

  if(sem_obj->semaphore > 0) {
    --sem_obj->semaphore;
    wait_satisfied = true;
  }
  else
    fx_sync_wait_start(object, wb);

  /* Освобождение спинлока объекта. */
  ...
  return wait_satisfied;
}
```

Сердцем функции увеличения счетчика, которая вызывает возобновление ожидающих потоков, является следующий код (выполняется с захваченным спинлоком объекта):

```
  if (/*Очередь блоков ожидания непуста.*/) {
    fx_sync_wait_block_t* wb = ... /* Получить блок ожидания из очереди. */
    fx_sync_wait_notify(&sem->waitable, ..., wb);
```

```
}
else
  ++sem->semaphore;
```

Если у семафора нет ожидающих потоков, его счетчик увеличивается, и на этом выполнение функции завершается. В противном случае, после получения блока ожидания, который зависит от политики очереди (FIFO или в соответствии с приоритетом), возобновляется выполнение одного из потоков, обновления счетчика при этом не происходит, так что, если у семафора всегда в момент нотификации есть ожидающий поток, его счетчик может вообще никогда не стать отличным от нуля.

6.5 Разновидности семафоров

За годы использования семафоров неоднократно предпринимались попытки сделать семафор еще более универсальным и подходящим под еще большее число возможных ситуаций. Для этого часто в интерфейс вводились дополнительные функции, которые вкратце будут рассмотрены в этом разделе.

Классические семафоры предполагают, что не существует способа узнать, будет ли поток заблокирован или нет до начала ожидания. Некоторые реализации (в частности, семафоры, включенные в стандарт POSIX), позволяют считывать значение семафора, хотя использование этой функции подвержено ошибкам, так как после считывания значение семафора может в любой момент измениться.

Широкое распространение имеет также немного урезанный вариант семафора, счетчик которого ограничен только двумя значениями: 0 и 1. Такой семафор называется **бинарным** и является усовершенствованной версией события. В ОС Windows такой объект синхронизации так и называется - синхронизационное событие (synchronization event) или событие с автоматическим сбросом (auto-reset event). Несмотря на то, что повторные попытки увеличения счетчика семафора теряются, в том случае, если он уже равен 1, такое поведение бывает, в некоторых случаях, желательно, поэтому бинарные семафоры часто оказываются даже более полезны, чем классические. Для примера можно привести ситуацию с потоком-сервером и клиентами. Сервер необходимо уведомить о наличии работы, а далее он проверяет свои структуры данных и обрабатывает запросы от всех клиентов. В этом случае, очевидно, что серверу нет нужды просыпаться столько раз, сколько было запросов, а последующие нотификации, в случае, если обработка запросов еще не началась, должны игнорироваться, поскольку будут учтены и так.

Для создания бинарных семафоров может предоставляться отдельное API, как в Windows. Альтернативный подход заключается в том, чтобы при создании семафора указывать максимальное значение, которое может принимать его счетчик, этот подход реализован в FX-RTOS.

Другое усовершенствование заключается в том, чтобы позволить пробуждение потока не только в случае ненулевого счетчика, но счетчика, большего определенного значения. В том случае, если поток ожидает сразу нескольких нотификаций от нескольких потоков, он может ожидать значения счетчика, например, больше 5, и возобновлять выполнение только тогда, когда семафор превысит это значение, после выполнения пяти операций post. Хотя, в некоторых случаях, такой вид семафров был бы полезен, он, очевидно,

требует анализа очереди ожидания всякий раз при выполнении операции post, поскольку каждый ожидатель может ожидать своего собственного значения семафора. По этой причине такие семафоры редко предоставляются как основное API в ОС реального времени.
Интересно, что бинарный семафор так же как и обычный, является универсальным примитивом, это означает, что на основе бинарных семафоров можно реализовать обычные и наоборот. Реализация не очень сложна и ее можно оставить в качестве упражнения.

6.6 Счетчики событий и секвенсоры

В качестве альтернативы семафорам были предложены и другие примитивы синхронизации. К ним можно отнести счетчики событий и секвенсоры, разрабтанные в 1979 году [6]. Счетчик событий, это, как следует из названия, счетчик, с которыми сопоставлены три основных метода:

```
int advance(event_count_t* object);
int read(event_count_t* object, int32_t* value);
int await(int32_t value);
```

В отличие от семафора, счетчик событий никогда не уменьшается. Потоки могут ожидать определенного значения счетчика с помощью функции await, и увеличивать его с помощью advance. Это немного напоминает работу семафора, но страдает обсуждавшейся выше проблемой: поскольку каждый поток может ждать любого значения счетчика и потоки начинают ожидание в произвольном порядке, в общем случае необходимо в результате каждого увеличения счетчика просматривать очередь для определения потоков, которые должны начать выполнение.
Более интересным объектом является секвенсор. Он содержит единственный метод, который, по сути, сводится к атомарному инкременту.

```
int32_t ticket(sequencer_t* object);
```

В основе его использования лежит такой же принцип, как и у электронной очереди. Каждый поток должен получить "талон", в котором содержится номер потока в очереди, а в операции ожидания указывается полученый номер (для ожидания используется операция await). Если текущий номер объекта меньше полученного номера - поток блокируется до того, пока счетчик не достигнет заданного значения с помощью вызовов advance.
По сравнению с семафором, секвенсор обеспечивает последовательность, то есть потоки получат доступ к разделяеому ресурсу именно в той последовательности, в которой они его запрашивали, но это одновременно является минусом, потому что, если по каким-либо причинам, поток, очередь которого подошла, не может выполняться, все остальные потоки будут ждать, даже если они могли бы получить доступ к ресурсу и закончить свою работу.
В некоторых задачах может оказаться полезной возможность оценить, сколько примерно

придется ждать. По аналогии с цифровой очередью, вытянув номер, указывающий, что впереди находятся 50 человек, можно пойти заниматься другими делами, вернувшись к очереди позже. Если потоку есть чем заняться помимо ожидания, он может, определив, с помощью своего "талона" и функции read, примерное количество потоков перед ним, заняться другими делами, вернувшись к ожиданию позже.
Счетчики событий сложнее с точки зрения реализации: счетчик никогда не уменьшается, поэтому, рано или поздно, он переполняется и логика использования примитива должна это учитывать. По причине не очень широкой поддержки в операционных системах, сложностей в реализации, а также в связи с более сложным прикладным интерфейсом, счетчики событий и секвенсоры не завоевали большой популярности в прикладном программировании.
Интересно, что, хотя в области прикладного программирования секвенсоры и счетчики событий не получили распространения, аналогичные идеи используются при реализации некоторых разновидностей спинлоков. В этом случае спинлок представляет собой два значения, одно используется как секвенсор, а второе - как счетчик событий. Всякий раз, при попытке подождать спинлок, ожидающий поток атомарно увеличивает секвенсор и получает свой номер. Ожидание происходит до тех пор, пока счетчик событий меньше полученного значения "талона". Освобождение спинлока происходит путём увеличения счетчика событий. Данный метод позволяет реализовать FIFO-политику захвата спинлоков, но для ожидания всеми потоками используется одна и та же переменная, поэтому рассмотренные выше спинлоки с очередью обладают лучшей производительностью.

6.7 Инверсия приоритета

Если семафоры используются для взаимного исключения, то помимо очевидных проблем вроде взаимоблокировки, которые могут возникать при попытке ожидания двух семафоров в обратном порядке, существует менее очевидная проблема, которая также имеет далеко идущие последствия. Например, рассмотрим систему, в которой имеется 3 потока с различным приоритетом (далее называемые потоками с высоким, средним и низким приоритетом), причем высокоприоритетный и низкоприоритетный поток имеют некоторые общие данные защищаемые бинарным семафором, используемым для взаимного исключения. Такие общие данные могут возникать косвенным образом даже тогда, когда приложение, вроде бы, в явном виде ничего такого не содержит. Например, механизм работы с памятью может быть устроен таким образом, что потоки добавляют освобождаемый блок память в некоторый глобальный список, а далее специальный, выделенный для этого, низкоприоритетный поток может заниматься дефрагментацией и прочими затратными действиями, сопряженными с освобождением памяти. При этом между потоками существуют общие данные (тот самый глобальный список), который нужно защищать. Возвращаясь к примеру, можно предположить, что поток со средним приоритетом может начать выполняться в тот момент, когда низкоприоритетный находится в критической области, защищаемой семафором. Поскольку среднеприоритетный поток имеет более высокий приоритет, то при политике планирования, такой как FIFO, низкоприоритетный поток не получит управления до тех

пор, пока он не закончит работу, что автоматически означает невозможность выйти из критической секции, а значит, в нее не сможет войти высокоприоритетный поток, даже если будет готов к выполнению. То есть высокоприоритетный поток получается, по сути, вытеснен среднеприоритетным потоком, потому что тот вытеснил третий поток, который удерживает нужные высокоприоритетному ресурсы. Такая нежелательная ситуация называется **неограниченной инверсией приоритета** (unbounded priority inversion). Неограниченной она называется потому, что время, на которое среднеприоритетный поток вытесняет низкоприоритетный потенциально не ограничено.
Обсуждавшаяся выше ситуация показана на Рис. 72.

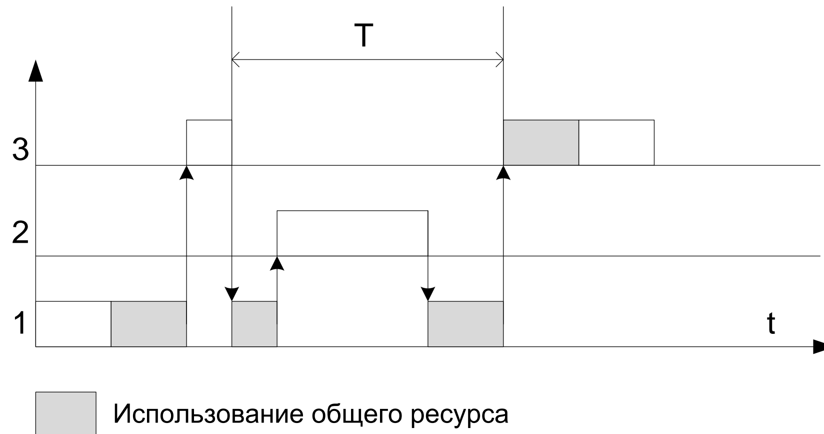

Рис. 72 Инверсия приоритета

Изначально выполняется наименее приоритетный поток 1 и входит в критическую секцию. Затем активируется высокоприоритетный поток 3 и пытается захватить ресурс. Поскольку последний занят, поток 3 блокируется и продолается выполнение потока 1. В какой-то момент активируется поток 2 и вытеняет поток 1. После завершения работы потока 2, поток наконец выходит из критической секции и работа потока 3 может быть продолжена. Очевидно что время ожидания потока 3, показанное как T зависит теперь не только от длины критической секции в потоке 1, но и от времени вытеснения этого потока третьими потоками, что и составляет суть проблемы инверсии приоритета.
Для ожидающего высокоприоритетного потока неизвестно, кто владеет ресурсом, поэтому для простых семафоров задача нерешаема. Хотя может быть запрограммирована явно на уровне приложения, например путем повышения приоритета для потоков, которые собираются войти в критическую область.
Хотя создается впечатление, что инверсия приоритета может возникать только в случае удержания нужных кому-то ресурсов, на практике она может возникать в самых неожиданных местах. Например, в предыдущей главе рассматривалась реализация таймеров, при которой обработка последних происходит в выделенном для этого потоке. Если таймеры используются потоками, приоритет которых выше, чем у потока таймеров, возможна следующая ситуация: высокоприоритетный поток останавливает собственное выполнение с помощью fx_thread_sleep или с помощью ожидания с таймаутом. Другой поток, с меньшим приоритетом, но большим, чем у потока, обрабатывающего таймеры вытесняет последний. Соответственно, таймер, который должен возобновить выполнение высокоприоритетного потока не будет вовремя обработан, то есть вновь возникает

специфический вид инверсии приоритета.

6.8 Мьютекс

Основной проблемой, вызывающей инверсию приоритета, является тот факт, что в системе нет информации о том, какой именно поток должен сделать инкремент семафора. Если бы эта информация была, можно было бы повысить приоритет тому потоку, который должен сделать sem_post, что решило бы проблему. Поэтому со временем был разработан объект называемый **мьютексом** (mutual exclusion). Он работает практически так же, как и бинарный семафор и используется для задач взаимного исключения, но есть и одно важное отличие от семафора - у мьютекса есть "владелец", тот поток, который находится в критической секции. Знание о владельце позволяет реализовать механизмы для обнаружения взаимоблокировок, а также защиту от инверсии приоритетов. Интерфейс типичных мьютексов выглядит следующим образом:

```
int mutex_timedacquire(mutex_t* mutex, uint32_t timeout);
int mutex_release(mutex_t* mutex);
```

Первая функция "захватывает" мьютекс, то есть ожидает, пока он освободится, а затем делает текущий поток его владельцем. Вторая функция - освобождает.
Одна из проблем, которые легко решаются с помощью мьютексов, это попытки ожидания потоком мьютекса, который уже был им предварительно захвачен. В случае с семафором, если он используется для взаимного исключения, поток может, находясь внутри критической секции, вновь попытаться его подождать, что, разумеется, приведет к взаимоблокировке. Такая ситуация может возникать в сложных системах, где функции реализующие критические секции с помощью семафоров вызываются друг из друга. Эта ситуация, очевидно, приведет к зависанию приложения, однако теоретически, приложение может эту ситуацию разрешить, сделав инкремент семафора из какого-то другого потока. В случае с мьютексом такой подход не сработает, если поток уже является его владельцем, то, в том случае, если этот же поток вновь попытается его подождать, никакой другой поток уже не сможет вывести систему из этого состояния, поскольку освободить мьютекс может только его владелец. Существует два взгляда, как следует решать эти проблемы. Первый вариант предполагает анализ текущего владельца мьютекса, и, если он совпадает с текущим потоком, из функции возвращается ошибка, мьютекс при этом остается захваченным, а приложение получает возможность обнаружить и обработать такой тип взаимоблокировки. Второй вариант увеличивает внутренний **счетчик вложения** мьютекса, таким образом, его освобождение произойдет тогда, когда поток-владелец вызовет операцию освобождения столько раз, сколько раз была вызывана операция захвата. Мьютексы поддерживающие такую возможность называются **рекурсивными**.
Еще одним отличием мьютексов от семафоров является возможность использования мьютекса только для задач взаимного исключения, мьютексы не могут использоваться для уведомления о событиях. Происходит это главным образом потому, что, как правило, во встроенных системах уведомления используются из обработчиков прерываний. Обработчик прерывания не может блокироваться и использовать функции ожидания, а

значит, не может быть владельцем мьютекса, которым может быть только поток. Соответственно, не являясь владельцем, обработчик прерывания не может и освободить мьютекс, поэтому все ОС обычно запрещают использование мьютексов в обработчиках прерываний. То же самое касается сигналов и асинхронных процедур. Хотя в этом случае, говоря формально, выполнение происходит в потоке, блокирование разрешено, а потенциальный владелец теоретически известен, использовать мьютексы также запрещается. Запрет связан с тем, что асинхронная процедура может начать выполнение в тот момент, когда поток уже владеет данным мьютексом. Хотя взаимоблокировок при этом не возникнет и рекурсивный мьютекс может быть корректно захвачен и освобожден внутри асинхронной процедуры, состояние данных, которые защищаются критической секцией может быть не определено, поэтому такой сценарий потенциально опасен и может привести к нарушению целостности данных.
Вместе с решением одной проблемы, мьютексы порождают другие. Семантика владения, которая предполагается для мьютекса поднимает проблему, которой не существует для семафоров, а именно - завершение потока во время владения мьютексом. Если поток явно не освободит мьютекс, значит никакой другой поток не сможет его захватить и система окажется в заблокированном состоянии. Существует несколько подходов к решению этой проблемы, но удовлетворительного решения не дает ни один из способов, поэтому самый оптимальный вариант - не завершать потоки во время критической секции. Один из возможных подходов: отслеживать мьютексы, которыми владеет поток, как часть состояния потока (список захваченных мьютексов, находящийся в TCB), что дает возможность их освободить при завершении работы потока. Критическая секция при этом, возможно, остается в неконсистентном состоянии, поскольку поток может быть завершен асинхронно в любой момент. Хотя потоки могут продолжить работу, может возникнуть несогласованность и разрушение структур данных. В этой связи бывает лучше оставлять мьютекс заблокированным, система будет заблокирована, но, хотя бы удастся избежать работы с возможно некорректными данными. Реализация алгоритмов разрешения подобных ситуаций усложняет и утяжеляет реализацию мьютексов, поэтому, если поток может быть завершен, лучше использовать бинарный семафор и реализовывать механизм защиты от инверсии приоритета явно в приложении (путем, например, повышения приоритета потока на время выполнения критической секции).
Из-за дополнительной функциональности (по сравнению с семафорами) мьютексы, в целом, работают медленнее в большинстве реализаций, семафор является более легковесным примитивом, однако для задач взаимного исключения предпочтительно использование именно мьютексов, поскольку они решают некоторые специфические проблемы, которые могут возникнуть.

6.8.1 Наследование приоритета

Знание о владельце позволяет организовать защиту от проблемы инверсии приоритета. При попытке подождать мьютекс, посколььу его владелец известен, можно сравнить приоритет владельца и приоритет текущего потока, захватывающего мьютекс. Если приоритет ожидающего больше, чем приоритет владельца, приоритет последнего асинхронно повышается. То есть повышение приоритета владельца вызывает тот факт, что

кто-то с большим приоритетом попытался захватить мьютекс [7]. При этом возникают некоторые трудности в реализации, в реальных системах потоки могут не только пытаться захватить мьютекс и ожидать, но и, например, не дождавшись в результате таймаута завершать ожидание без захвата мьютекса. Все такие ситуации необходимо отслеживать и корректно повышать и понижать приоритеты в соответствующих случаях. Проблема усугубляется тем, что поток-владелец и сам может менять свой приоритет в процессе работы, поэтому корректная синхронизация этих процессов гарантирующая невозможность некорректной установки приоритета потока, это довольно нетривиальная задача, требующая использования нескольких спинлоков и решения возможных взаимоблокировок.

Большинство ОС реализуют наследование приоритета только на одном уровне, то есть от ожидателя мьютекса к его владельцу, но оказывается, что этого бывает недостаточно. Поток может владеть сразу несколькими мьютексами, поэтому инверсия приоритета может возникнуть опосредованно, через цепочку владельцев: когда кто-то пытается ждать мьютекса, владелец которого сам ждет некоторого другого мьютекса, владельцем которого является третий поток. На Рис. 73 показана ситуация, когда потоки (показанные окружностями с числом приоритета) ожидают мьютексов (показаны прямоугольниками), при этом поток T1 с приоритетом 8 уже владеет одним из мьютексов. В тот момент, когда поток T1 начал ожидание, потоку T2, владеющему нужным мьютексом был повышен приоритет (до 8). После этого, мьютекс, которым владеет поток T1 могли начать ожидать потоки с гораздо боле высоким приоритетом. Это ожидание повысит приоритет T1 до уровня 20, но избавиться от инверсии приоритета не удастся, потому что поток T1 сам заблокирован и следовало бы повышать приоритет потоку T2.

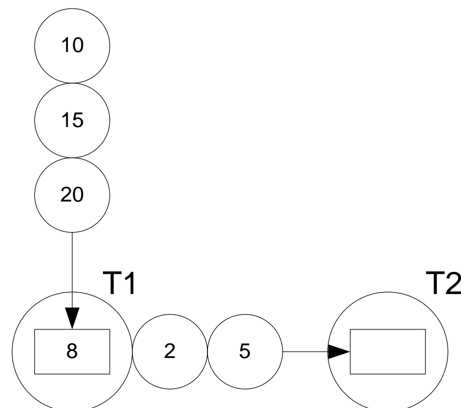

Рис. 73 Цепочка ожидания мьютексов

В этом случае требуется пройти по всей цепочке и соответствующим образом изменить приоритеты. Проблема осложянется еще и тем, что нет некоторого объекта, который можно было бы использовать как общую точку синхронизации, требуется знание глобального состояния системы. Иными словами, гарантированное выполнение наследования приоритетов с учетом цепочек требует глобальной блокировки ядра, которая чревата большими проблемами с производительностью и масштабируемостью. Кроме этого, поскольку проход во всем цепочкам занимает некоторое время, пропорциональное количеству и длинам этих цепочек, поэтому могут также возникнуть

проблемы с предсказуемостью этого процесса. Могут также применяться комбинированные техники, когда система пытается обеспечить распространение приоритета по цепочкам, но не гарантирует этого в любой момент времени.

6.8.2 Потолок приоритета

Реализация наследования детерминированным образом в многопроцессорной системе чрезвычайно сложна и чревата ошибками, даже если предположить, что она корректно реализована, эта реализация может настолько усложнить работу объекта синхронизации, что само использование мьютексов будет обладать существенными накладными расходами по сравнению с семафорами, что в свою очередь будет провоцировать разработчиков прикладного ПО не использовать мьютексы вовсе. Тем более программе, в большинстве случаев, известно, какие потоки могут захватывать мьютекс и может ли в данном конкретном случае произойти инверсия приоритета.
Одним из решений является перенос логики избежания инверсии в пользовательское приложение. С мьютексом сопоставляется приоритет, называемый **потолком**, по аналогии с потоком (указывается при создании). Все потоки, захватывающие этот мьютекс, получают повышение приоритета до потолка. При этом не требуется сложной логики внутри реализации мьютекса, алгоритм работает за фиксированное время и обеспечивает необходимую защиту от инверсии приоритета. Именно такой вариант реализован в FX-RTOS и многих других встроенных ОСРВ. Данный протокол называется протоколом с непосредственным потолком приоритета (immediate priority ceiling protocol, IPCP). Существует еще оригинальный протокол потолка приоритета (Original priority ceiling protocol, OPCP) , но он довольно сложен в реализации и практически не применяется на практике. Мьютексы с потолком приоритета чувствительны к порядку, в котором мьютекс освобождается. Нарушение этого порядка чревато неправильной установкой приоритета потока после завершения критической секции. Если для обычных мьютексов их освобождение в порядке обратном захвату это рекомендация (нарушение приводит к снижению производительности из-за лишних переключений контекста), то для мьютексов со статическим потолком - требование.

6.8.3 Реализация мьютексов

Переходя к реализации мьютексов, стоит рассмотреть вариант со статическим потолком приоритета, как наиболее простой и в то же время эффективный. Объект мьютекса содержит, как и все примитивы синхронизации, абстрактный ожидатель, поле указывающее на текущего владельца, счетчик рекурсивных захватов данного мьютекса, а также поля, связанные с реализацией потолка приоритета: параметры планировщика, ассоциированные с мьютексом, а также оригинальные параметры потока-владельца, сохраняемые в структуру мьютекса на время его захвата (восстанавливаются после выхода из критической секции).

```
typedef struct _fx_mutex_t {
  fx_sync_waitable_t wait_queue;
  ...
  fx_thread_t* owner;
  fx_sched_params_t ceiling_params;
  fx_sched_params_t original_params;
```

```
}
fx_mutex_t;
```

Хотя логика работы мьютекса сходна с бинарным семафором, реализация функции ожидания несколькьо сложнее. В том случае, если используется потолок приоритета, если оказалось, что поток успешно захватил мьютекс, его параметры планирования копируются в специально предназначенное для этого поле внутри объекта мьютекса. Затем к потоку применяются параметры планирования определенные как потолок мьютекса.

```
bool mutex_test(fx_sync_waitable_t* object, fx_sync_wait_block_t* wb, ...) {
  fx_mutex_t* mutex = /*Приведение типа от переменной object. */
  fx_thread_t* const me = fx_thread_self();
  bool acquired = true, ceiling = false;

  /* Захват спинлока мьютекса. */
  ...

  /* Проверка владельца и захват, если мьютекс был свободен. */
  if(!mutex->owner) {
    mutex->owner = me;
  }
  else
  {
    /* Мьютекс занят другим потоком, вставка в очередь ожидания. */
    fx_sync_wait_start(object, wb);
    acquired = false;
  }

  /* Освобождение спинлока мьютекса. */
  ...

  if(acquired == true) {
    /* Сохранение оригинального приоритета в структуре мьютекса и
       установка потоку потолка приоритета мьютекса. */
  }
  return acquired;
}
```

Функция освобождения мьютекса устроена чуть более сложно. Дело в том, что во время ожидания, поток не должен увеличивать свой приоритет, потому что ожидание может быть завершено и без захвата мьютекса, например в результате таймаута или доставки асинхронной процедуры. Нежелательно, чтобы поток даже кратковременно работал на высоком приоритете, если мьютексом он не владеет. В то же время, также нежелательна работа на низком приоритете когда мьютекс уже захвачен, так как это может привести к инверсии приоритета, от которой, по идее, механизм должен защищать. Выход из этой ситуации заключается в том, что приоритет потоку повышает не он сам, а тот, кто освобождает мьютекс. Фрагмент функции освобождения мьютекса представлен ниже, предполагается что весь этот код выполняется в защищенной спинлоком мьютекса критической секции.

```
  /*Сохранить оригинальные параметры планирования владельца в локальной переменной.*/
  fx_sched_params_t sched_params;
```

```
  fx_sched_params_copy(&mutex->original_params, &sched_params);

  /* Найти следующего владельца мьютекса, и повысить ему приоритет. */
  if (/* очередь непуста */) {
    mutex->owner = /* Новый владелец из очереди ожидания. */

    /* Сохранение оригинального приоритета нового владельца в структуре мьютекса.*/
    ...
    /* Установка приоритета новому владельцу.*/
    ...

    /* Возобновление выполнения ожидающего потока. */
    fx_sync_wait_notify(&mutex->wait_queue, ..., wb);
  }
  else
    mutex->owner = NULL;

  /* Понижение собственного приоритета до оригинального. */
}
```

Установка новых параметров планирования происходит под спинлоком, поэтому могут потребоваться дополнительные соглашения или еще один спинлок, в том случае если состояние потока также им защищено.

6.9 Мониторы

Все рассмотренные выше примитивы достаточно низкоуровневые. С их помощью решаются простые задачи, вроде взаимного исключения и уведомлений. Хотя управление доступом к нескольким экземплярам некоторых ресурсов является основной задачей семафора, очевидно, что используя один семафор задачу решить нельзя. Например, классическая задача "читатели-писатели", требуют еще более сложной логики в коде приложения. Эта задача состоит в в том, что есть буфер некоторого фиксированного размера, один или более потоков могут вносить в него данные, другая группа из одного или более потоков - извлекать эти данные. Если буфер заполнен, потоки-писатели должны ждать, пока в нем не освободится место, если буфер пуст - потоки-читатели должны ждать, пока в нем появятся данные. Подобные задачи довольно часто возникают во встроенном ПО и требуют использования нескольких объектов типа семафора. В свою очередь, использование нескольких семафоров автоматически приводит к возможным взаимоблокировкам, в случае их захвата в разном порядке.
Можно считать, что проблема примитивов, хранящих внутреннее состояние, является то, что они требуют, чтобы всякая задача могла быть переформулирована таким образом, чтобы состояние задачи отображалось на состояние примитива. В случае, если состояние алгоритма сложное и включает много параметров, доступ к нему должен синхронизироваться с помощью примитивов, обеспечивающих взаимное исключение и тут может возникнуть проблема блокировки внутри критической секции.
Более универсальным случаем было бы попробовать реализовать примитив таким образом, чтобы пользователь сам определял состояние, соответствующее задаче, а не пытался придумать, как состояние его задачи отображается на предопределенное поведение такого примитива как семафор. **Монитор**, предложенный в 1974 году, является как раз таким типом объекта [8].

Монитором считается не просто объект, наподбие семафора, а совокупность следующих сущностей: набор разделямых данных и функций, при выполнении которых обеспечивается взаимное исключение, а также один или более особых объектов, называемых **условными переменными**. То есть, в терминах ООП, монитор это класс, для методов которого обеспечивается взаимное исключение. Условная переменная играет роль очереди ожидания, имеющейся в других примитивах, но с одним отличием, которое будет рассмотрено далее. Так как одного только взаимного исключения, которое обеспечивается для функций монитора недостаточно, нужна еще возможность условной синхронизации, для реализации которой и используются условные переменные.
Монитор задумывался как механизм, поддержка которого должна быть встроена непосредственно в язык программирования, и следить за взаимным исключением должен компилятор. В языке С поддержка мониторов отсутствует, поэтому реализовывать взаимное исключение необходимо "вручную", обычно для этого используется мьютекс.
Что касается условных переменных, то они имеют два метода: wait, используемый для ожидания и signal для пробуждения ожидателей.

```
int cond_timedwait(cond_t* cond_variable, mutex_t* mutex, uint32_t timeout);
int cond_signal(cond_t* cond_variable);
```

На первый взгляд неясно, зачем потребовалось создавать какие-то специальный объект, если все эти возможности существуют в рассмотренных ранее примитивах: для взаимного исключения можно использовать мьютекс, а для условной сигнализации - семафор, непонятно, зачем все усложнять понятием монитора и условных переменных. Ответ заключен в логике работы условной переменной. Это отличие заключается в том, что она, во-первых, должна использоваться только в критической секции, во-вторых - ожидание условной переменной приводит к тому что поток помещается в очередь, затем разблокируется мьютекс и только после этого поток блокируется. За счет того, что поток начинает ожидание виртуально *ДО* того, как он освободил мьютекс, возможные уведомления со стороны других потоков не будут пропущены, а состояние алгоритма получается защищенным мьютексом и вынесенным в пользовательскую программу. Такой механизм работы, когда после начала ожидания поток не блокируется, а выполняет еще некоторые дополнительные действия довольно сложно реализовать на других примитивах, поэтому условные переменные как правило требуют поддержки со стороны ядра ОС.
В целом, монитор это как бы "вывернутый наизнанку" примитив, состояние которого определяется программой, а не заключено внутри объекта в качестве его состояния.
Разумеется, после того, как поток получил сигнал и его выполнение возобновилось, он должен опять захватить мьютекс. Таким образом, ожидание переменной происходит вне критической области, поэтому другие потоки могут выполняться и сигналить ждущему.
Например, если реализовать семафор с помощью монитора, то для этого потребовалось бы три объекта: переменная, хранящая счетчик семафора, а также мьютекс и условная переменная, используемые для взаимного исключения и ожидания соответственно:

```
cond_t condition_variable;
mutex_t mutex;
uint32_t semaphore;
```

Функция ожидания содержала бы следующий код:

```
mutex_timedacquire(&mutex, ...);
do {
  if(semaphore > 0) {
    --semaphore;
    break;
  } else {
    cond_timedwait(&condition_variable, &mutex, ...);
  }
}
while(1);
mutex_release(&mutex);
```

В отличие от семафора, здесь состояние представлено непосредственно в пользовательском приложении (переменной semaphore), а доступ к переменной синхронизируется с помощью мьютекса. Если выясняется, что необходимо ожидание, поток блокируется с помощью условной переменной. Особое внимание следует обратить на передачу мьютекса в функцию ожидания условной переменной, что позволяет вначале поместить поток в очередь ожидания, а затем освободить мьютекс.
Увеличение счетчика семафора (аналог sem_post) происходит так:

```
mutex_timedacquire(&mutex);
++semaphore;
cond_signal(&cond);
mutex_release(&mutex);
```

Из-за того, что отсутствует возможность узнать, есть ли потоки, заблокированные на условной переменной, функция ожидания содержит цикл. После увеличения семафора, происходит уведомление одного из ожидателей с помощью функции cond_signal. Если ожидающие потоки отсутствовали, выполнение этого метода не приводит ни к каким последствиям. Впоследствии, если другие потоки попытаются уменьшить счетчик семафора, они увидят его ненулевое значение. Если же ожидающие потоки присутствуют, тогда после уведомления одного из них он вновь попытается захватить мьютекс и уменьшить счетчик семафора. В отличие от примитивов, хранящих внутреннее состояние, условная переменная состояния не имеет, а потому вызов функции cond_timedwait всегда приводит к ожиданию и блокировке потока.
В остальном, эти функции примерно соответствуют рассмотренным выше функциям реализации семафора с помощью очередей ожидания, то есть условная переменная позволяет пользователю организовывать собственные очереди ожидания защищаемые мьютексом.
Хотя это и не так очевидно, но возможна и обратная реализация монитора на семафоре. Пример можно найти в [9].
Реализация условных переменных гораздо проще всех рассмотренных примитивов из-за

того, что она практически идеально отображается на внутренние API очередей ожидания, используемые ядром ОС.
Функция проверки состояния примитива, используемая для реализации ожидания выглядит так:

```
bool cond_test(fx_sync_waitable_t* object, fx_sync_wait_block_t* wb, ...)
{
  fx_mutex_t* mutex = ... /*Получение указателя на мьютекс через атрибуты ожидания. */

  /* Захват спинлока условной переменной.*/
  ...

  fx_sync_wait_start(object, wb);

  /* Освобождение спинлока условной переменной */
  ...

  fx_mutex_release(mutex);
  return false;
}
```

Указатель на мьютекс передается через атрибуты ожидания, то есть через блок ожидания. Проверка того, что данный поток является владельцем мьютекса должна выполняться в объемлющей функции, поэтому, если выполнение дошло до ожидания, можно предполагать, что данный поток является владельцем мьютекса и вызов fx_mutex_release всегда будет успешным. Вставка блока ожидания в очередь условной переменной перед освобождением мьютекса и является той важной особенностью, которая требует поддержки со стороны ядра ОС. Такое поведение трудно реализовать на других примитивах, так как в этом случае, вставка в очередь и блокировка потока выполняются атомарно, лишая возможности выполнить какие-то дополнительные действия, такие как освобождение мьютекса.
Уведомление ожидателей условной переменной также выглядит довольно тривиально.

```
int fx_cond_signal(fx_cond_t* cond, ...)
{
    fx_sync_wait_block_t* wb;

    /* Захват спинлока условной переменной.*/
    ...

    wb = /* Получение блока ожидания из очереди */
    fx_sync_wait_notify(&cond->waitable, ..., wb);

    /* Освобождение спинлока условной переменной */
    ...
}
```

Хотя любой монитор должен иметь только один мьютекс для корректной реализации взаимного исключения, условных переменных может быть много. Например описанная выше задача читателей и писателей могла бы использовать две условные переменные, одну для читателей и другую для писателей.
Некоторые реализации условной переменной содержат также функцию broadcast, вместе

с signal, которая вызывает пробуждение всех ожидающих потоков, а не только одного.

```
int cond_broadcast(cond_t* cond);
```

Использование такой функции может оказаться полезным в ситуации, когда не всякий поток, среди заблокированных, может продолжать работу. Например, если монитор защищает аллокатор памяти, то, после того, как кто-то освобождает память (возвращает ее аллокатору), нужно пробудить те потоки, кому эта память может быть нужна. Но планировщик потоков не может знать заранее размер запрашиваемой памяти, поэтому следующий в очереди поток может обнаружить, что, хотя памяти и стало больше, его конкретный запрос не может быть удотворен, потому что ему требуется памяти больше, чем было освобождено. В таких случаях лучше пробуждать все ожидающие потоки с тем чтобы каждый из них вновь попытался получить свой ресурс.
Если реализация монитора с условными переменными без широковещания на семафорах не отличается сложностью, реализация широковещания гораздо более нетривиальна, что дает дополнительные плюсы для ОС, поддерживающие такие примитивы на системном уровне.
Мониторы широко применяются в программировании так как поддерживаются в таких распространенных языках как Java и C#. Для ускорения работы применяется множество оптимизаций, которые не рассматривались в данной главе. Тем не менее, следует помнить, что применение оптимизаций, улучшающих среднюю производительность, как правило сопряжено со снижением предсказуемости, поэтому в ОСРВ эти оптимизации используются редко.

6.10 Специфические примитивы

Механизмы синхронизации, рассмотренные выше, такие как монитор и семафор являются универсальными, то есть с их помощью можно реализовать любой механизм синхронизации и решить любую задачу координации действий нескольких потоков. Тем не менее, многие задачи являются типичными. Хотя их и можно реализовать на базе универсальных примитивов синхронизации, это могло бы привести к дублированию кода и возможным ошибкам, поэтому ряд стандартов, в том числе POSIX, определяют некоторые дополнительные примитивы, которые не столь универсальны, но удобны для решения некоторых часто возникающих задач. В области встроенных систем использование специфических примитивов синхронизации также распространено в связи с особыми требованиями к производительности. Далее вкратце (без подробного рассмотрения реализации) будут рассмотрены некоторые такие примитивы.

6.10.1 Флаги событий

Первый примитив, который имеет широкое распространение в области встроенных систем, но отсутствует в настольных и серверных ОС называется **флаги событий** (event flags). На практике часто встречаются задачи, когда нужно ждать либо выполнения нескольких условий одновременно либо какой-либо комбинации условий. Зачастую это требует использования нескольких объектов и довольно сложно в реализации, поэтому для этих целей может использоваться специальный примитив. Типичной ситуацией, когда

может потребоваться ожидание нескольких объектов это выделение одного потока для взаимодействия с несколькими устройствами. Разумеется, все это можно сделать и на традиционных примитивах, но вся логика по выяснению "что произошло" переносится в приложение.
Флаги событий представляют собой переменную, обычно имеющую тип размерности данных процессора, то есть 16 или 32 бита. С помощью методов объекта можно индивидуально устанавливать и сбрасывать биты, содержащиеся в переменной, а также ждать любой их комбинации. Причем ожидание может как атомарно сбрасывать биты, так и оставлять их установленными. В этом смысле примитив представляет собой набор из событий или бинарных семафоров в зависимости от типа ожидания, для которых можно атомарно ожидать любой комбинации. В отличие от стандартных примитивов, интерфейс для работы с флагами событий различаются в разных ОС.
Указывается состояние флагов, а также опция, следует эти флаги установить, либо сбросить (если set = true то флаги устанавливаются, если false - сбрасываются).
Очевидный минус, отличающий флаги событий от рассмотренных ранее примитивов - необходимость просмотра всей очереди для выяснения, какой поток необходимо разбудить, так как каждый из них может ждать собственной комбинации флагов.
Например, на Рис. 74 показана очередь ожидания флагов событий, текущее значение которых равно 010 (внутри потока-окружности указано ожидаемое значение с флагом AND). Установка младшего бита с переходом значения в 011 должна разбудить ожидателя, ждущего 001, но два других ожидателя должны остаться заблокированными.

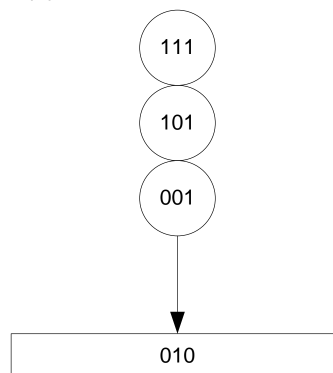

Рис. 74 Очередь ожидания для флагов событий

Поскольку пробуждение может быть связано с изменением состояния примитивов, просмотр выполняется под спинлоком примитива, то есть время просмотра очереди (и время владения спинлоком) пропорционально количеству ожидающих потоков. Это расходится с требованиями жесткого реального времени, поэтому использование данного примитива требует со всей ответственностью подойти к проектированию системы, чтобы было заранее известно и ограничено количество ожидающих потоков.

6.10.2 Барьеры

Многие задачи, в основном связанные с вычислениями, предполагают разбиение задачи на последовательность этапов таким образом, что каждый этап может хорошо распараллеливаться, но следующий этап может начаться лишь после завершения

предыдущего. В качестве примера можно привести вычисление с участием матриц. Операции с матрицами хорошо распараллеливаются, но к следующему этапу вычисления часто можно приступать только когда закончился предыдыдущий.
Разумеется, задача решается как с использованием семафоров, так и условных переменных, но есть примитив, предназначенный специально для решения такого типа задач - **барьер**.
Интересной особенностью барьера является то, что он имеет всего один метод, который является одновременно и методом ожидания и методом нотификации.

```
int barrier_timedwait(barrier_t* br, barrier_key_t* key, uint32_t timeout);
```

При инициализации барьера указывается число потоков, которые участвуют в "забеге". Далее каждый поток, вызывающий функцию ожидания увеличивает внутренний счетчик барьера и блокируется, в том случае, если значение счетчика меньше количества участвующих потоков. После того, как все потоки подходят к барьеру, счетчик сбрасывается, а они разблокируются и продолжают выполнение до следующего ожидания барьера и так далее. Поток, который вызывает разблокирование может узнать об этом с помощью параметра "ключ", который для такого потока устанавливается в специальное значение, отличающиееся от значения у остальных потоков.
Корректное использование барьера предполагает, что потоки не должны завершаться во время работы с ним, в противном случае результат работы не определен. В FX-RTOS, как можно видеть из описания, если поток дошел до функции ожидания, окончание его этапа фиксируется, если потом поток был завершен или решил завершить ожидание по таймауту это не повлияет на барьер.

6.10.3 RW-блокировки

Ранее уже была рассмотрена задача читателей и писателей применительно к некоторому буферу. Существует подвид этой задачи, когда читатели не забирают данные, записанные писателем, а используют совместно. Например, может существовать поток обновляющий данные с датчиков, причем все датчики должны изменяться согласованно. И есть набор всех остальных потоков, которые могут эти данные читать. Самый простой вариант - использование мьютекса и критической секции. Но это требует выстраивания в цепочку и всех читателей, хотя для данного алгоритма это не требуется, так как читать данные могут и несколько потоков. В случае, если писатель обновляет данные довольно редко, это может привести к снижению производительности. По сути, требуется особый вид критической секции, умеющей разделять разные виды запросов: писатель должен быть синхронизирован с другими писателями и с читателями, тогда как читателям синхронизация между собой не требуется. Такой специальный примитив называется RW-блокировкой (RW lock, reader-writer lock) и имеет три метода, не считая стандартных конструктора и деструктора.

```
int rwlock_rd_timedlock(rwlock_t* rw, uint32_t tout);
int rwlock_wr_timedlock(rwlock_t* rw, uint32_t tout);
int rwlock_unlock(rwlock_t* rw);
```

Состоянием примитива являются счетчик читателей и указатель на писателя, причем, если писатель присутствует, то количество читателей должно быть равно нулю для реализации корректного взаимного исключения. Поскольку есть две независимые друг от друга очереди, имеет смысл использовать два waitable-объекта, которые будут использоватся при попытке подождать от имени читателя и писателя соответственно. Эти объекты разделяют общий спинлок, поэтому их изменение может происходить согласованно.

```
typedef struct _fx_rwlock_t {
  fx_sync_waitable_t readers;
  fx_sync_waitable_t writers;
  ...
```

Примитив является продвинутым вариантом мьютекса, который допускает одновременный вход в критическую секцию некоторой группы потоков, тогда как для другой группы этого не допускается.
Могут произойти следующие ситуации во время работы с RW-блокировкой. Со стороны писателя (пользователя функции wr_timedlock) возможны следующие варианты:

- объект свободен
- объект занят другим писателем
- объект занят читателями.

Со стороны читателей:

- объект свободен
- объект занят читателями и есть ожидающий писатель
- объект занят писателем.

Если объект свободен, этот случай не очень интересен. Если писатель обнаруживает, что объект занят кем-либо, объект работает аналогично мьютексу. После того, как этот писатель покидает критическую область, он должен возобновить выполнение одного из ожидающих потоков и тут возникает неоднозначность, кому следует отдавать объект. Традиционно приоритет предоставляется писателям.
Наибольший интерес представляет поведение объекта со стороны читателей. Если объект не занят писателем, читатели допускаются в критическую область и могут работать над бщими данными параллельно. В случае, если объект занят другим писателем, как читатели так и писатели должны ждать, поэтому в этом случае они блокируются. Если читатель видит, что объект ожидается писателем, он также блокируется, что позволяет избежать голодания писателей из-за чередования читателей.
Как и любые объекты, которые предполагают семантику владения и удержания некоторого объекта, RW-блокировки подвержены проблеме инверсии приоритета, поэтому некоторые ОСРВ реализуют для них, так же как и для мьютексов, различные механизмы наследования.

6.10.4 Блоки памяти

При написании приложений часто используется динамическая память. Это может потребоваться при реализации сетевых протоколов, когда требуется выделение памяти

под приходящие пакеты, а также для многих других случаев. Выделение блоков произвольного размера вносит непредсказуемость из-за возможной фрагментации, а также требует поиска подходящего блока, что приводит к зависимости времени работы функции от ситуации, поэтому часто используется компромиссное решение - выделение блоков фиксированного размера. Обычно выделение памяти работает неблокирующим образом: при невозможности выделить память, функция выделения возвращается немедленно с ошибкой. Например, подобным образом работают функции стандартной библиотеки С malloc и free. Для некоторых задач подобный интерфейс является не очень удобным, поскольку в случае ошибки и возврата нулевого указателя у вызывающей функции в общем-то немного вариантов дальнейших действий: неизвестно когда появится память и появится ли она вообще. Блокирующее выделение памяти является более удобным, поскольку позволяет не обрабатывать такие ошибки, а заблокировать выполнение потока до тех пор, пока кто-то не освободит память. Интерфейс выглядит так:

```
int block_pool_timedalloc(block_pool_t* bp, void** alloc_block, uint32_t timeout);
int block_pool_release(void* blk_ptr);
```

Используется особый объект, называемый пулом блоков памяти (memory block pool), при инициализации которого ему передается участок памяти, из которого будут браться блоки фиксированного размера. Далее этот объект используется аналогично примитиву синхронизации: при необходимости выделить память, приложение вызывает соответствующую функцию и либо получает запрошенный блок, либо, в случае пустого пула, блокируется до тех пор, пока другой поток не вернет свой блок в пул с помощью функции block_pool_release.
Поскольку выделение блоков фиксированного размера может быть реализовано таким образом, что все функции будут работать за строго определенное время, а также будут свободны от фрагментации, использование такого механизма выделения памяти является предпочтительным для систем реального времени.

6.10.5 Очереди сообщений

Еще один широко распространенный тип объекта синхронизации, который появился как решение одной из типичных задач - очередь сообщений. Выше уже рассматривался вопрос синхронизации доступа к буферу на основе семафоров и мониторов. Очередь сообщений, это примитив, который объединяет в себе, собственно, сам буфер и логику его синхронизации таким образом, что буфер оказывается скрыт от приложения. Последние обращаются к объекту непосредственно за данными и если таковые в буфере имеются они передаются запрашивающему потоку. Как и в случае циклического буфера, попытка извлечь данные из пустой очереди или поместить в заполненную приводят к блокировке потока.

```
int msgq_front_timedsend(msgq_t* msgq, uintptr_t msg, uint32_t tout);
int msgq_back_timedsend(msgq_t* msgq, uintptr_t msg, uint32_t tout);
int msgq_timedreceive(msgq_t* msgq, uintptr_t* msg, uint32_t tout);
```

Две первые функции используются для того, чтобы поместить элемент в начало или в

конец очереди соответственно, а третья - для извлечения элемента. Для очереди могут применяться оптимизации: например, если поток пытается поместить данные в пустую очередь, у которой есть заблокированные ожидатели, данные могут быть помещены непосредственно в приёмный буфер ожидающего потока, без предварительной буферизации внутри очереди, что позволяет добиться лучшей производительности (такая очередь называется очередью без копирования, zero copy queue).
Отдельным вопросом является размер элемента очереди. Хотя его можно указывать при создании объекта, использование слишком больших элементов (например, вмещающих в себя целый сетевой пакет в несколько десятков килобайт) не очень практично, потому что потребует в лучшем случае копирования данных к принимающему потоку, а в худшем случае (когда используется буферизация) потребуется два копирования. Выходом из этой ситуации является использование указателей на выделенные буферы и передача через очередь сообщений только указателей, которые имеют небольшой размер (тип uintptr_t). Некоторую трудность представляет вопрос, кто должен управлять памятью, освобождением и выделением памяти для буфера. Этот вопрос решается с помощью подхода, известного как семантика владения: в каждый момент времени только один поток должен владеть указателем на данный буфер, он же и должен его освобождать. Таким образом, с помощью очереди возможна передача владения другому потоку. Удобно использовать очередь совместно с примитивом "блоки памяти", рассмотренным ранее. После того, как получены данные, указатель на буфер помещается в очередь, с этого момента поток теряет владение буфером и не должен более к нему обращаться и запоминать указатель. Поток, который читает из очереди, прочитав из него следующий элемент, может быть уверен в том, что он хозяин этого буфера, то есть он обладает единственным указателем. После работы с данными поток-владелец возвращает память в пул и перестает быть его владельцем. Далее буфер может быть снова выделен другим потоком и вновь помещен в очередь сообщений и т.д (Рис. 75)

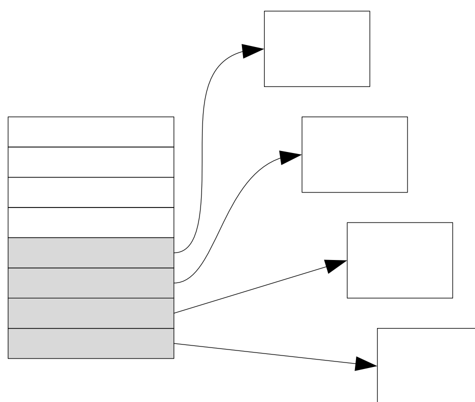

Рис. 75 Очередь сообщений с указателями в качестве данных

Несмотря на то, что очередь сообщений не относится к стандартным и универсальным объектом синхронизации, такие объекты предоставляются большинством ОСРВ. Дело в том, что очередь сообщений очень удобно использовать для обмена данным между обработчиками прерываний и потоками. Поскольку использовать мьютексы в прерываниях нельзя, а семафоры и подобные объекты могут потребовать ожидания в

обработчиках прерывания, реализация взаимодействия между потоками и обработчиками прерываний может стать довольно сложной и нетривиальной задачей, так как описанные универсальные примитивы создавались в основном все-таки для синхронизации взаимодействия именно потоков. В этом свете очередь сообщений это один из важнейших примитивов, предоставляемых ОСРВ. Так как пул блоков памяти также можно использовать внутри обработчиков при условии указания нулевого таймаута, вместе с очередью сообщений они предоставляют продвинутый механизм для эффективного обмена данными с обработчиками прерываний.
Описанный механизм обмена данными между потоками называется **асинхронными сообщениями** из-за того, что выполнение отправителя и получателя не синхронизировано, а данные буферизируются внутри объекта. В противоположность этому существует метод синхронного обмена сообщениями, в этом случае буферизация отсутствует. Можно рассматривать это как частный случай, при котором размер буфера равен нулю и перемещение данных происходит только между источником и приемником непосредственно. В этом случае их выполнение синхронизируется в точке обмена.

6.11 Резюме

Различные задачи связанные с синхронизацией требуют различных подходов. Использование объектов синхронизации требует атомарности, поэтому очень трудно обобщить эти объекты. Это является причиной того, что для разных задач существует набор примитивов, которые удобно использовать в той или иной ситуации.
При программировании встроенных систем используется два основных механизма синхронизации: реализация взаимного исключения и условная синхронизация. В простых системах, особенно в том случае, когда ресурсы существенно ограничены, допустимо использование для синхронизации таких методов, как запрет прерываний и прямое увдомление потоков из обработчиков прерываний. К сожалению, эти методы сложны в использовании, а также по-разному реализуются в разных ОС, что затрудняет портирование, поэтому использовать такие подходы можно только в порядке исключения, для всех остальных ситуаций существуют предназначенные для этого объекты.
Несмотря на то, что семафор также пригоден для реализации взаимного исключения, в большинстве случаев ему следует предпочесть мьютексы, которые, помимо прочего, обеспечивают дополнительные проверки корректного использования, а также защиту от инверсии приоритета. Семафоры лучше всего подходят для уведомления о событиях, таких как прерывания. С его помощью можно относительно несложно реализовать в том числе обработку таких прерываний, которые могут вновь возникать даже в то время, когда одно из таких прерываний обрабатывается. Кроме того, ввиду широчайшей поддержки семафоров в различных операционных системах и открытых стандартах, он также может использоваться с целью достижения максимальной переносимости.
В остальных случаях, особенно в ситуациях, когда требуется синхронизация с учетом множества переменных и условий можно рекомендовать использование мониторов или условных переменных совместно с мьютексами. Программирование с использованием мониторов менее подвержено ошибкам, в отличие от использования семафоров, а также

проще поддается модификации. К сожалению, не все встроенные ОСРВ предоставляют интерфейсы для использования условных переменных, поэтому могут возникнуть трудности с портированием, но если используемая ОС поддерживает эти объекты и менять ее не планируется, то это разумный компромисс.

В том случае, когда ОСРВ поддерживает специфические примитивы и они хорошо подходят для конкретной задачи, целесообразно их использовать для устранения дублирования кода и переписывания логики таких примитивов на стороне приложения. Особо стоит отметить блоки памяти и очереди сообщений. Для большинства встроенных систем требуется коммуникация между потоками и обработчиками прерываний с передачей данных, поэтому не следует пренебрегать возможностью использования готовых решений.

[1] Alpern, Shneider, "Recognizing safety and liveness" 1987
[2] Maurice Herlihy, Nir Shavit, "The Art of Multiprocessor programming" Elsevier 2008 (p.2)
[3] Anderson, Dahlin, "Operating Systems. Principles and Practice" (c5.1.3, p.184)
[4] T. Harris, "A pragmatic implementation of non-blocking linked lists" 2001
[5] A. Downey "The Little book on semaphores" 2005
[6] Reed, Kanodia, "Synchronization with event counts and sequencers" 1979
[7] L. Sha, R. Rajkumar, J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization" 1990
[8] Per Brinch Hansen," The programming language Concurrent Pascal" 1975
[9] Silberschatz et al., "Operating system concepts" John Wiley & Sons 2009 (6.7.3, p.250)

7 Защита и изоляция

Темы главы:

- Процессы как механизм изоляции приложений
- Обзор различных вариантов реализации процессов
- Особенности реализации системных вызовов в ОС с разделяемым ядром
- Реализация ввода-вывода для процессов без прямого доступа к устройствам

7.1 Постановка задачи

Системы, включающие реализацию потоков и механизмов синхронизации, рассмотренные ранее, позволяют строить довольно сложные встроенные системы, для многих устройств этого функционала вполне достаточно. Тем не менее, по мере усложнения встроенного ПО, проявляется такой недостаток упомянутого подхода, что всё приложение работает в одном домене защиты, то есть потенциальные ошибки в одной из подсистем влияют на всё устройство. Поэтому наметилась тенденция добавления во встроенные ОСРВ механизмов защиты, сходных с теми, которые применяются в ОС общего назначения, где изначально требовалось изолировать несколько независимых приложений друг от друга таким образом, чтобы ошибки в одном приложении не могли повлиять на другие.

Встроенное ПО может содержать много достаточно сложного кода, такого как реализация файловых систем или коммуникационных протоколов, причем это ПО может использоваться для вспомогательных функций устройства. Например, устройство может содержать веб-интерфейс для сбора статистики, что требует наличия в нем стека TCP/IP, который сам по себе является довольно громоздким. Большое количество кода потенциально приводит к большему количеству уязвимостей и ошибок. Так как все эти компоненты работают как единое приложение в одном адресном пространстве, ошибки во второстепенных частях приложения приводят к тому, что к каждой части системы предъявляются требования надежности такие же, как и к самому ответственному коду, что повышает стоимость разработки, сертификации, а также увеличивает цену ошибки.

Необходимость в изоляции компонентов встроенного ПО друг от друга возникает также по экономическим причинам. Для ответственных систем может применяться физическое разделение различных функций, то есть использование отдельного как аппаратного так и программного обеспечения (такие системы называются федеративными (federated)). Однако это решение довольно дорого, и в ряде случаев было бы целесообразно использовать одну аппаратную базу для запуска сразу нескольких встроенных приложений, особенно в том случае, если каждое приложение требует относительно немного вычислительных ресурсов [1]. Если каждое из таких приложений использует свои аппаратные интерфейсы для взаимодействия с внешним миром, то эффект будет такой же, как если бы каждое приложение работало на своем собственном контроллере, притом что единый контроллер, разумеется, дешевле. Кроме того, использование выделенной аппаратуры сильно усложняет коммуникацию между компонентами ПО, а также может потребовать изменения аппаратной части, при необходимости изменить существующий функционал. Возможность запуска нескольких приложений на одном

устройстве упрощает коммуникацию между ними, так как, в отличие от случая, когда каждому процессу выделено отдельное оборудование, здесь имеется общая память, через которую можно организовать обмен данными.

Отдельно стоит упомянуть проблему безопасности. Все те же рассуждения, которые касались ошибок и их влияния на все устройство в целом, могут быть распространены и на уязвимости, когда через возможные бреши в безопасности второстепенных компонентов может быть скомпрометирован основной функционал устройства и перехвачено управление им.

Наконец, изоляция частей приложения удобна также для инженеров, разрабатывающих ПО, поскольку позволяет быстрее обнаруживать и локализовывать некоторые виды ошибок, таких как, например, обращение по неверным указателям. Повышается надежность, поскольку разный функционал изолирован, в случае критической ошибки устройство в целом может продолжать работу.

Таким образом, подытоживая сказанное, можно указать следующие аргументы в пользу использования аппаратных механизмов защиты:

- Разделение кода с разным уровнем ответственности (изоляция ошибок)
- Использование одного устройства для параллельной работы нескольких приложений
- Упрощение разработки и тестирования
- Разграничение доступа с целью обеспечения безопасности

Из этих задач следуют требования к реализации системы защиты: должна быть возможность (хотя бы гипотетически) использовать одни и те же приложения как в составе сложной системы с несколькими изолированными приложениями, так и в качестве единственного приложения в классической встроенной системе, то есть прикладные интерфейсы не должны сильно различаться; требуется возможность задания ограничений на использование системных ресурсов для каждого из приложений; должна быть возможность перехватывать и анализировать возникающие ошибки. Грубо говоря, весь работающий в составе встроенной системы код, можно разделить на доверенный, и недоверенный. Первый работает как и прежде, с полным доступом к аппаратуре и памяти, а второй работает в режиме урезанных привилегий и ограничений доступа в память. В контексте ОСРВ, это означает, помимо прочего, также невозможность второстепенных компонентов ПО повлиять не только на корректность работы и безопасность, но также и на временные характеристики прочих компонентов. Поскольку непривилегированная часть приложения предположительно подвержена ошибкам и различным проблемам с безопасностью, следует при проектировании исходить из того, что она, в процессе работы, может делать (или пытаться сделать) буквально всё, что угодно - некорректно использовать API ОС, обращаться по любым адресам памяти, использовать знания о механизмах работы ОС с целью ее компрометации и прочее и прочее. Любые такие действия должны оставаться допустимыми и не приводить к нарушению работы как прочих приложений, так и ядра ОСРВ.

Хотя, в общем случае, предполагается сосуществование нескольких, не сильно связанных между собой, приложений в рамках одного аппаратного обеспечения, допустимо

разделение на несколько частей с разным уровнем ответственности и одного приложения. Например, если некоторая ответственная система управления технологическим процессом использует логирование в файл, логично отделить логирование и код, связанный с файловой системой, от управляющего приложения, поскольку эти части являются достаточно сложными и в то же время не очень важными для функционирования устройства. Другой пример: приложение может принимать команды извне (например, по сети) и выполнять их, проверяя при этом корректность полученных команд в контексте логики устройства и отказываясь выполнять заведомо некорректные команды. Дополнительный контроль и отказ от выполнения некорректных команд, пусть даже и приходящих по штатным каналам данных, может обеспечить дополнительную устойчивость и безопасность. То есть, даже в случае, если в результате эксплуатации возможных уязвимостей в реализации сложных сетевых протоколов будет возможно выполнение кода на атакуемой встроенной системе, полный контроль над ней захватить не удастся.

Наконец, существуют ситуации, когда требуется ограничить доступ приложения к периферийным устройствам, даже в том случае, если приложение не нуждается в защите и использование единого домена безопасности допустимо. Некоторые контроллеры содержат возможности по обновлению ПО, которая используется для удаленного обновления прошивки. При этом используемый чип может содержать механизмы защиты интеллектуальной собственности, такие как CRP (code read protection), которые позволяют на аппаратном уровне включить защиту от чтения и обновления содержащегося в флеш-памяти ПО [2]. Уязвимости в коде приложения, потенциально позволяют злоумышленнику, путем использования уже содержащегося в прошивке кода для обновления флеш-памяти, перепрограммировать контроллер таким образом, чтобы предотвратить дальнейшее обновление, то есть, по сути, программным образом вывести из строя оборудование. Такая атака может повлечь значительное время и материальные ресурсы для своего устранения.

7.2 Концепция процесса

Ключевой концепцией, которая лежит в основе отделения приложений друг от друга, является концепция **процесса**. Процесс - это экземпляр работающей программы, а также связанных с ней ресурсов, среди которых могут быть адресное пространство, регионы памяти, потоки, которые выполняют код, права доступа к устройствам и прочие сведения подобного характера [3]. То есть, процесс является своего рода контейнером ресурсов, ассоциированных с программой. Процесс содержит все атрибуты приложения, такие как код, данные, стек одного или более потоков и так далее.

Поддержка процессов типична для настольных и серверных ОС. Во встроенных системах идея примерно такая же, хотя есть и некоторые технические различия. Например, все настольные системы ориентированы на использование MMU и отдельных адресных пространств для каждого из приложений, тогда как в области встроенных систем возможно использование MPU и единого адресного пространства.

Важно понимать, что сам по себе процесс является пассивной сущностью, которая не выполняет код, выполнение кода всегда происходит с помощью потока, который

принадлежит определенному процессу. В ранних ОС (включая первые реализации UNIX) процесс мог содержать только один поток, поэтому можно было говорить о выполнении кода процессом, поскольку они соответствовали друг другу как один к одному. Из-за этого впоследствии возникло много путаницы, так как в литературе речь часто идет об операционных системах семейства UNIX, поэтому могут обсуждаться вопросы "синхронизации процессов". В общем случае процесс является многопоточным, и, разумеется, говорить о выполнении кода процессом уже не совсем корректно, по крайней мере в контексте разработки ОС. Для краткости под "действиями процесса" далее будут подразумеваться действия одного из его потоков.

Рассмотренные выше варианты с единым встроенным многопоточным приложением можно рассматривать как системы с единственным процессом, который содержит все потоки, а также все аппаратные ресурсы. В многопроцессном окружении эти ресурсы некоторым образом распределяются между несколькими процессами, то есть процессу предоставляется только ограниченное подмножество ресурсов и контролируютсяпопытки выйти за пределы этого подмножества.

Изоляция процессов друг от друга достигается путем использования аппаратной защиты, а также соответствующих интерфейсов HAL, рассмотренных в главе 4. Хотя возможны различные подходы, но, как правило, процессы являются непривилегированными, то есть выполнение содержащихся в них потоков происходит в пользовательском режиме процессора, что позволяет контролировать поведение процесса и учитывать выделяемые ему ресурсы. Разумеется, реализовать поддержку процессов можно только в том случае, если аппаратное обеспечение поддерживает механизмы изоляции и защиты памяти.

Хотя ранее ядро ОС было определено как сущность, управляющая потоками и их синхронизацией, в контексте обсуждения процессов, существует и другое определение. Поскольку теперь используются различные режимы работы процессора, бывает целесообразно называть ядром ту часть системы, которая работает в привилегированном режиме процессора, который из-за этого и получил свое название - "режим ядра" (kernel-mode). Начиная с этой главы и далее под ядром будет подразумеваться именно это, если явно не оговаривается иное. Для избежания путаницы, реализация потоков, планировщика и базовых механизмов синхронизации, рассмотренных в предыдущих главах, будет называться **наноядром**[3].

С точки зрения программирования, процесс ничем не отличается от прочих объектов рассмотренных ранее: он имеет свяанную с ним структуру, а также набор методов. По аналогии с потоками, объект процесса называется PCB (process control block). Хотя процесс и не является выполняемой сущностью, для него определено понятие **переключения процесса**. В отличие от переключения потоков, в результате переключения процесса меняется не поток управления и регистры общего назначения, а контекст защиты. То есть, например, в случае наличия MMU, переключение процесса может включать переключение текущего каталога страниц, для того чтобы обеспечить правильный

---

[3] В некоторых источниках (см. [7]) реализация потоков и планировщика называется микроядром. В микроядерной архитектуре этот же термин используется для обозначения части системы, работающей в режиме ядра. Для избежания неоднозначности был выбран менее устоявшийся, но более однозначный термин.

контекст защиты памяти (Рис. 76).

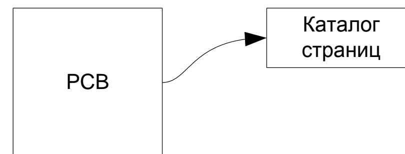

Рис. 76 Process control block

Код, исполняемый потоками процесса, выполняет прикладные задачи, обращаясь к ядру ОС для выполнения привилегированных действий. ОС не знает, чем занимается процесс и ограничивает его только в тех действиях, которые могут повлиять на другие процессы или саму ОС.

Так как теперь ОС должна уметь защищать себя от некорректных действий процесса, следует вкратце рассмотреть такие действия, которые могут приводить к нарушению штатного функционирования системы, и которые, следовательно, нужно контролировать. К таким дйствиям относятся:

- Обращения к памяти. Процесс должен иметь возможность обращаться только в явно разрешенные для него регионы памяти.
- Вызовы функций ОС. Поскольку функции ОС, вызываемые через механизм системных вызовов, работают в привилегированном режиме, неверные аргументы функций ядра могут спровоцировать крах системы.
- Обращение к устройствам. Так как некоторые устройства имеют прямой доступ к физической памяти, с их помощью можно обойти защиту памяти, если разрешить процессу программировать устройство непосредственно.

Проверка аргументов также должна быть устроена сложнее, чем в однопроцессном случае. Количество аргументов, которое можно передать с помощью системного вызова ограничено, поэтому довольно часто используются указатели и передача данных по ссылке. В таком случае возможна такая уязвимость, что, после того, как функции ядра проверили корректность аргументов, пользователь может изменить аргументы и добиться либо использования некорректных аргументов.

Если ядро содержит ограниченное количество ресурсов, должны быть предусмотрены квоты, в противном случае процессы могут намеренно вызвать исчерпание ресурса, что, потенциально, может сделать невозможной работу других, возможно более важных процессов. Ядро должно учитывать эти ресурсы независимо от кода процесса, поскольку он может не освобождать их намеренно, и, в таком случае, после завершения процесса эти ресурсы могут быть утрачены.

Очевидно, что реализация всех этих механизмов защиты требует довольно существенного объема кода, поэтому ОС с поддержкой процессов в целом более тяжеловесны, чем ОСРВ с поддержкой только потоков. Распределение ОСРВ по целевым аппаратным платформам выглядит, как показано на Рис. 77:

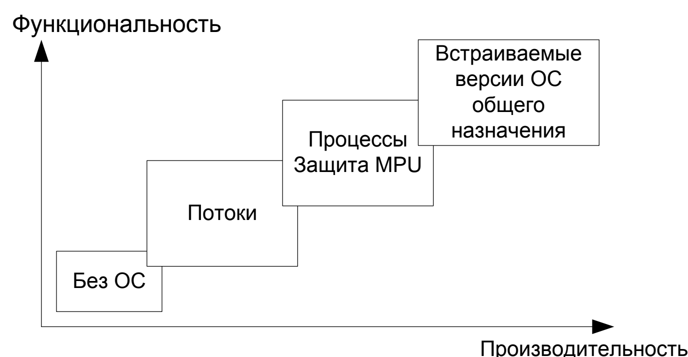

Рис. 77 Зависимость функциональности ОС от производительности процессора.

Совсем простые системы, как правило, обходятся вовсе без ОС. По мере роста производительности процессора и сложности встроенного ПО становится оправданным применение потоков. Далее появляется необходимость в изоляции частей системы друг от друга, поэтому системы, обладающие достаточной для этого производительностью, используют ОСРВ, поддерживающие один или несколько процессов и защиту памяти. Нужно понимать, что, хотя такие системы и поддерживают некоторые возможности, свойственные ОС общего назначения, в целом, такие ОС занимают промежуточное положение и сильно урезаны в своих возможностях с целью соответствия требованиям работы в реальном времени, а также требованиям производительности. Наконец, когда появляется необходимость в динамическом запуске во время работы нескольких приложений, динамическая загрузка драйверов устройств и тому подобное, целесообразно использовать модификации ОС общего назначения, которые адаптированы для работы в составе встроенных систем, например Windows embedded.

По мере роста функциональности ОС, растет ее сложность, поэтому зависимость времени реакции встроенной системы от наличия механизмов изоляции ошибок выглядит уже противоположным образом. Системы без ОС могут обеспечить наилучшее время реакции, поскольку всё может быть подчинено этой единственной цели. Наличие потоков привносит некоторые дополнительные расходы, связанные с переключением контекста, отдельными стеками и так далее. В дальнейшем ситуация все более усугубляется, приходя в итоге к системам общего назначения, время реакции которых варьируется в широких пределах.

7.2.1 Модель защиты с одним процессом

Довольно частым требованием к встроенной системе является не столько возможность запуска сторонних приложений, сколько необходимость ограничить определенные сложные части встроенного ПО в их возможности воздействовать на систему. Поэтому одной из самых простых моделей защиты, является модель с одним непривилегированым процессом. В этом случае, приложение разделяется на доверенную и недоверенную часть, которые работают в соответствующих режимах процессора. Предполагается, что доверенное приложение хорошо протестировано и может работать с оборудованием напрямую. Поскольку механизм взаимодействия непривилегированного кода с ядром ОС похож на механизмы, использующиеся в ОС общего назначения, можно условно считать часть, выполняемую в пользовательском режиме единственным

приложением, которое общается с ядром ОС через системные вызовы. Ресурсы для процесса выделяются при сборке образа статически и не освобождаются, предполагается, что процесс запускается и работает все время пока работает система. Хотя это по прежнему единый образ и разделение на ядро и приложение условно, такая система уже может явным образом определить, что допустимо делать непривилегированному коду внутри процесса, а что нет.
Привилегированная часть приложения в такой архитектуре по прежнему статически компонуется с ядром и пользуется его сервисами напрямую, путем вызова функций (Рис. 78).

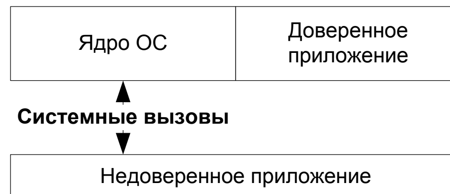

Рис. 78 Структура системы с одним процессом.

Выделение в отдельный процесс потенциально опасного кода дает возможность доверенному приложению и ядру ОС пережить фатальную ошибку в недоверенной части. Тем не менее, для полного восстановления работоспособности необходим перезапуск. Хотя это требование, на первый взгляд, сводит на нет все преимущества использования процессов - ведь в случае отсутствия процессов, в случае ошибки, устройство точно так же должно быть перезапущено - такой подход может быть полезным для вынесения второстепенной логики в непривилегированный режим. В случае выполнения процессом некорректных действий, все его потоки принудительно останавливаются. Привилегированная часть приложения может проанализировать причины ошибки и запротоколировать их, для последующего изучения, после чего перезапустить устройство в штатном режиме, если это возможно.
Такие системы по скорости работы и требованиям к аппаратным ресурсам сравнимы с встроенными ОС без поддержки процессов и могут успешно работать в системах с несколькими десятками килобайт памяти, то есть накладные расходы относительно невелики.

7.2.2 Модель защиты с несколькими процессами

Предыдущая описанная модель предполагает только два уровня ответственности, по которым могут быть распределен выполняющийся код. Дальнейшим развитием этой идеи и ее обобщением является модель с несколькими процессами (Рис. 79).

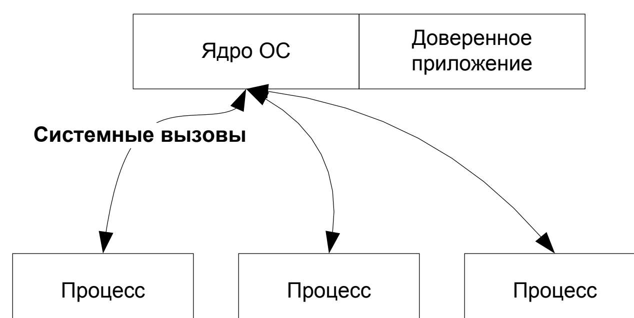

Рис. 79 Структура системы с несколькими процессами.

Ресурсы по прежнему распределяются между ними статически, и точно так же, в случае возникновения ошибок, аварийный процесс останавливается (останавливаются все принадлежающие ему потоки) и необходим перезапуск всей системы. Перезапуск отдельных процессов невозможен, но они изолированы как от ядра так и друг от друга. Отсутствие возможности перезапуска позволяет исключить из ядра сложную логику отслеживания используемых процессами ресурсов, поэтому накладные расходы минимальны и сравнимы с предыдущим случаем.

В отличие от настольных компьютеров, во встроенных системах, количество процессов, которые будут в них выполняться, как правило, известно заранее, на этапе проектирования системы. Это позволяет распределить между ними память и прочие ресурсы статически, во время инициализации системы, и удалить из ядра ОСРВ возможности, связанные с управлением памятью. Отказ от динамического управления памятью позволяет добиться большей надежности и скорости работы, так как ядро получается более компактным и содержтит меньше кода. В таком случае, ПО встроенной системы представляет собой набор предопределенных заранее бинарных модулей, каждый из которых запускается в созданном для него процессе.

Во время старта и инициализации системы происходит запуск процессов и их дальнейшая работа. Если система поддерживает перезапуск процессов, то он происходит с повторным использованием уже выделенных ресурсов, поэтому динамического перераспределения памяти во время работы системы не происходит. На первый взгляд, это лишает программиста возможности выделять и освобождать память во время работы, что может показать слишком радикальным упрощением, но на самом деле ситуация не так плоха. Дело в том, что процессу выделяется некоторая фиксированная область памяти, которую он может использовать, а далее, уже внутри процесса, в непривилегированном режиме процессора, с этим выделенным регионом работает локальный менеджер памяти, который реализует стандартные функции управления памятью. То есть, с точки зрения программиста, все это выглядит так же, как и окружение, предоставляемое приложению в настольных ОС. Процессы лишены только возможности выделять память сверх того максимума, который был определен на этапе сборки системы.

Поскольку в системе отсутствует менеджер памяти, который мог бы распределять память между процессами динамически, эта работа возлагается на специализированное приложение, используемое совместно с инструментами компиляции и сборки, которое использует в качестве входных параметров набор модулей, которые хотелось бы запустить в виде изолированных процессов. Дополнительно может сообщаться также и другая информация о ресурсах, которая не может быть определа путем анализа бинарного модуля. После анализа этих модулей и выяснения количества памяти, которое им нужно с учетом всех поправок, имеющаяся физическая память распределяется между ними и генерируется структура данных, описывающая данное распределение. Эти данные в дальнейшем в каком-то виде содержатся в образе, загружаемом в устройство. После старта ядра ОС, эти данные анализируются и создается соответствующее распределение с настройкой параметров защиты еще до запуска пользовательских процессов.

Использование такого подхода позволяет запустить несколько незавимых приложений в пределах одного устройства, а также разделить несколько частей одного приложения по уровням ответственности. Например, некоторое устройство может принимать по сети команды, а также также вести журнал событий в файле (для чего нужна реализация файловой системы). Возможность получать из сети команды можеть быть намного важнее способности писать журнал, поэтому логично отделить файловую систему от реализации сетевого стека и исключить влияние одного на другое.

Как и в предыдущем случае, в такой системе может существовать привилегированная часть приложения, стстически компонуемая с ядром, которое, при необходимости, может проанализировать причины аварии и перезапустить устройство. Главное здесь то, что аварийное завершения второстепенного процесса делает ситуацию нештатной, но не делает ее аварийной, так как основной функционал устройства продолжает работать, а перезапуск устройства нужен лишь для восстановления необязательных функций.

На первый взгляд, можно отказаться от прикладного кода в режиме ядра и получить тем самым систему, похожую на системы общегоназначения, в котором существуют только ядро и набор процессов, без возможности запуска приложений в режиме ядра. Это упростило бы систему и позволило бы использовать один бинарный компонент ядра с любым набором приложений. Тем не менее, возможность существования доверенных приложений, на которые не распространяются механизмы защиты, является важной с точки зрения обеспечения минимального времени реакции, так как передача управления, сопряженная с переключением процесса занимает намного больше времени, чем вызов ISR, который мог бы быть реализован в первом случае.

Главной проблемой описанного метода является необходимость перезапускать устройство всякий раз, когда аварийно завершается хотя бы один из процессов. Поэтому дальнешее развитие такой модели заключается в том, чтобы позволить перезапуск отдельных процессов, что позволило бы обеспечить непрерывную работу устройства даже в случае отказов отдельных программных компонентов. Это наиболее сложная модель, сходная с той, которая используется в настольных и серверных ОС. Реализация такой логики довольно сложна, поскольку необходимо учитывать используемые процессом ресурсы и корректно их освобождать в случае его завершения, поэтому вариант с перезапускаемыми процессами используется в более продвинутых системах обладающих достаточными ресурсами.

Использование статического анализа также повышает безопасность приложений, поскольку в системе физически отсутствует код, который может менять карту памяти во время работы, поэтому отсутствует возможность получить доступ к памяти другого процесса.

7.2.3 Планирование процессов

Несмотря на то, что процесс сам по себе не может выполнять код, а является лишь контейнером различных ресурсов, в некоторых системах он может выступать в качестве планируемого объекта, объединяющего в себе группу потоков. В системах повышенной надежности, таких как авионика, часто применяется двухуровневая схема планирования, вначале планируются процессы, затем, внутри промежутка времени, отведенного для

процесса планируются его потоки. Механизм планирования процессов и потоков при этом существенно различается. Например, один из подходов, применяющийся там, где необходимо в первую очередь обеспечить выполнение нескольких независимых процессов на одной аппаратуре, предполагает жесткое разделение времени между процессами в определенной пропорции. Всё время работы системы выглядит как бесконечная последовательность так называемых "главных кадров" (major frame) [4]. В свою очередь, главный кадр состоит из совокупности времен выполнения каждого из процессов системы. Например в системе с тремя процессами, время может быть распределено в пропорции 50% - 25% - 25% (Рис. 80):

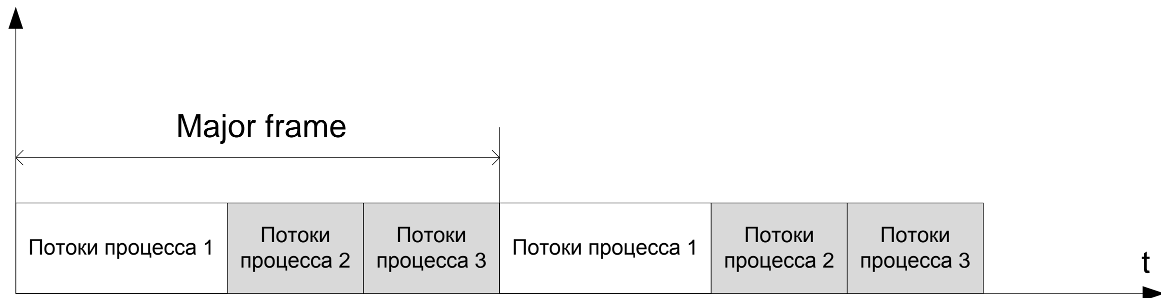

Рис. 80 Двухуровневая схема планирования.

Если главный кадр выполняется, например 100 мс, то первому процессу предоставляется 50 мс времени для выполнения его потоков, потом 25 мс следующему, и, наконец 25 мс работают потоки третьего процесса. Расписание составляется на этапе разработки системы и не может изменяться во время работы. Такой подход обеспечивает жесткое разделение времени и гарантирует, что ни один процесс не сможет выполняться дольше чем отведенный ему временной интервал, что обеспечивает хорошую предсказуемость. Негативным свойством такого механизма является то, что если потоки данного процесса завершили работу до истечения его времени, никакой другой процесс все равно не получит процессорное время. То есть приоритет потока влияет только на планирование внутри кванта времени, отведенного процессу.

Альтернативный подход похож на применяемый в системах общего назначения. Процесс не является планируемой сущностью, а приоритет потока имеет общесистемный смысл, то есть потоки планируются без учета принадлежности к процессам. Такой подход является более распространенным в связи с тем, что он обеспечивает хорошую скорость реакции, в отличие от предыдущего случая, где потокам любого процесса, независимо от его важности, вначале нужно дождаться следующего выделенного ему промежутка времени. Кроме того, подход с кадрами плохо применим в многопроцессорном окружении.

Поскольку потоки выполняются по мере их перехода в состояние готовности, картина распределения процессорного времени по процессам полностью непредсказуема и определяется во время работы системы (Рис. 81).

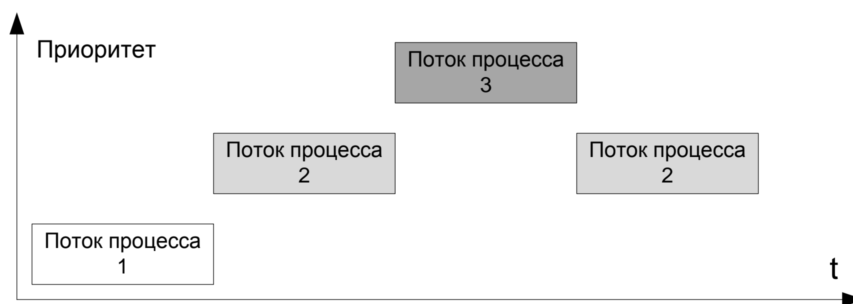

Рис. 81 Сквозное планирование потоков принадлежащих разным процессам.

Поскольку каждый процесс работает в своем собственном контексте защиты памяти, переключения процесса возникают одновременно с переключением потока, в том случае если новый поток принадлежит другому процессу.

Поскольку такая реализация процессов является наиболее распространенной, в дальнейшем будет рассматриваться именно этот подход.

7.3 Подходы к реализации процессов

7.3.1 Виртуальные машины

Так как концепция процессов служит, в том числе, для того, чтобы запускать несколько независимых приложений на одном физическом устройстве, самый логичный механизм заключается в том, чтобы предоставить приложению иллюзию того, что оно полностью владеет устройством. Тогда несколько приложений могли бы работать так же, как они работали бы на разных физических устройствах. В этом случае ядро ОС выполняет функцию **виртуальной машины** (виртуальная машина - ПО, эмулирующее аппаратное обеспечение).

Ядро ОС при этом получается относительно простым. Каждому пользовательскому процессу соответствует один поток ядра, после того, как этот поток получил управление, ему периодически посылаются "прерывания", которые обрабатываются уже пользовательским кодом. Из-за исключительной простоты - часть системы, работающая в привилегированном режиме процессора минимальна и осуществляет только мультиплексирование процессов - такие методы используются в системах, где требуется повышенная надежность.

Что касается эмуляции аппаратного обеспечения и прерываний, то обычно это происходит следующим образом. Поскольку прерывание, с точки зрения аппаратуры, это сохранение текущего контекста и передача управления обработчику, можно потребовать от процесса иметь структуры, аналогичные тем, которые требуются от приложения реальным процессором: таблицу векторов прерываний IVT. После получения "настоящего" аппаратного прерывания, ядро ОС может изменить сохраненный в стеке пользовательский контекст таким образом, чтобы после возврата из прерывания вернуться не в прерванный код, а в установленный пользователем виртуальный "обработчик прерывания", адрес которого можно узнать из некоторых глобальных структур данных. Все происходит по аналогии с тем, как обработчик прерывания должен быть сообщен процессору. Такой механизм называется **upcall** (вызов "вверх"), из-за того,

что в данном случае ядро является инциатором вызова некоторой пользовательской функции (Рис. 82). Таким образом, для кода процесса создается полная иллюзия того, что он работает на некоторой машине, которая время от времени вызывает обработчики прерываний и делает все прочие действия, свойственные "настоящей" аппаратуре.

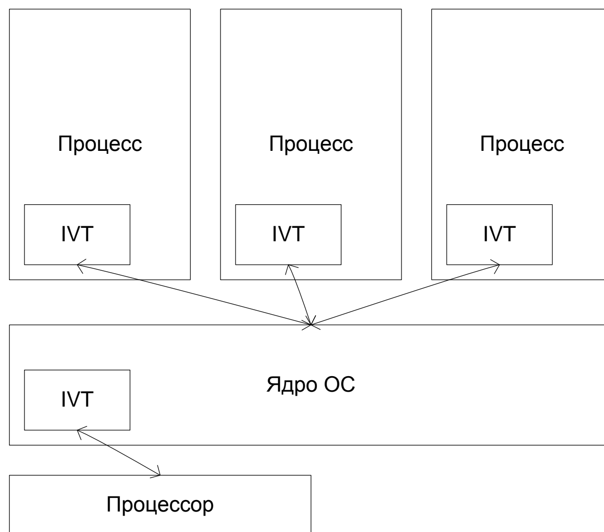

Рис. 82 Ядро ОСРВ в роли виртуальной машины.

Будучи похожей на аппаратное обеспечение, виртуальная машина позволяет реализовать все подходы, обсуждавшиеся в главе 3, такие как суперцикл, циклический планировщик и прочие. Соответственно, несколько процессов могут быть представлены несколькими суперциклами, время между которыми распределяется в соответствии с некоторым расписанием (то есть используется двухуровневое планирование с кадрами), как показано на Рис. 83.

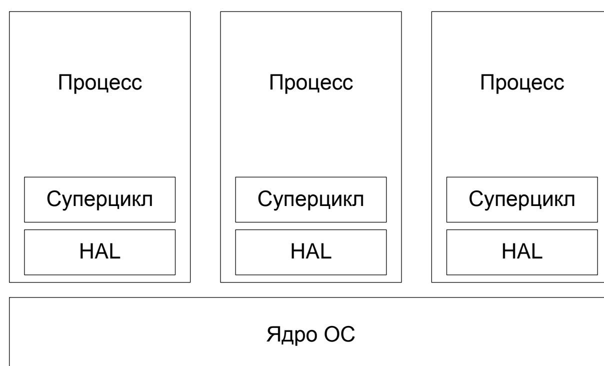

Рис. 83 Разделение с помощью процессов нескольких циклических приложений.

С точки зрения ядра, все пользовательские потоки (соответствующие процессам) всегда являются активными.

Так как предоставляемая виртуальной машиной абстракция пригодна для запуска суперцикла и при этом имеет возможность доставки прерываний в пользовательский режим, внутри виртуальной машины можно запустить "ядро" ОС, рассмотренное в предыдущей главе, и программировать приложения обычным образом с потоками и всеми прочими возможностями предоставляемыми ядром. Возможен даже запуск внутри

каждого из процессов разных ОС пользовательского уровня (Рис. 84):

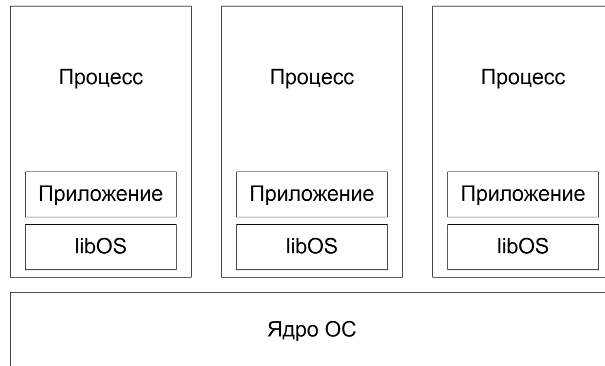

Рис. 84 Отдельная реализация потоков в каждом процессе.

Для ОС, работающей внутри процесса, разумеется должен быть реализован HAL, соответствующий интерфейсу виртуальной машины, поэтому окончательная структура выглядит как показано на Рис. 85:

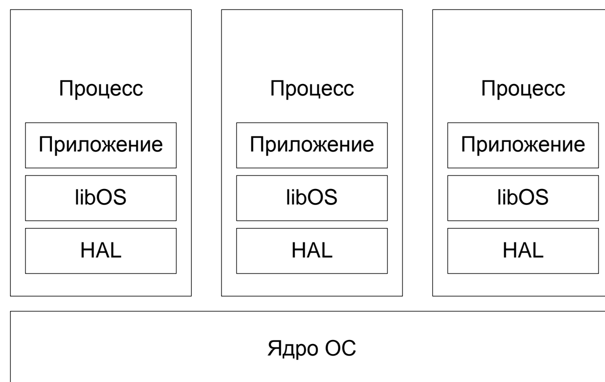

Рис. 85 HAL для реализации потоков внутри процесса.

Система на основе виртуальных машин отличается надежностью: интерфейс между ядром и приложением приближен к аппаратному, то есть является простым и через него очень сложно повлиять на другие процессы. Это является решающим фактором при построении систем с особыми требованиями к надежности. Из-за того, что пользовательские потоки, также как и планировщик, и все остальные компоненты, работают в пользовательском режиме, при этом достигается высокая производительность, то есть накладных расходов на реализацию виртуальной машины относительно немного. Однако, несмотря на все эти плюсы, такой подход обладает также серьезными минусами. Поскольку все пользовательские потоки планируются, фактически, внутри одного потока с точки зрения ядра, при этом исчезает возможность использования многопроцессорных систем. Запуск нескольких потоков ядра с целью эмуляции многопроцессорной системы не сильно помогает, поскольку для эффективного использования спинлоков использующий их код должен уметь запрещать вытеснение, а так как процесс лишен возможности запрета собственного вытеснения с целью безопасности, его время может быть отобрано ядром в любой момент, в том числе и после захвата спинлока, что приводит к неоправданным расходам временных ресурсов на активное ожидание.

Поскольку реализация потоков и планировщика у каждого из процессов может быть своя, затрудняется также коммуникация между ними. Из-за отсутствия разделяемых

примитивов синхронизации, приходится использовать для общения между процессами, по сути, только общую память и атомарные операции. Наконец, для эффективной работы "внутрипроцессного" планировщика, может потребоваться доставка в него прерываний таймера, что в случае высокого разрешения последнего может привести к существенным накладным расходам, поскольку действия по программному перенаправлению прерываний должны выполняться сотни и тысячи раз в секунду.

7.3.2 Разделяемое ядро

Из рассуждений выше понятно, что поддержка процессов в виде набора виртуальных машин хороша только тогда, когда предъявляются особые требования к надежности, ради которых стоит свести разделяемый между всеми процессами код к минимуму (код ядра ОС или виртуальной машины). Несмотря на некоторые достоинства, для прикладного программиста такая система видится как набор копий некоторой аппаратной платформы, что не всегда удобно с точки зрения программирования, особенно в том случае, когда процессы объединены некоторой общей целью и им необходимо эффективно общаться между собой.

Вследствие этих причин большее распространение получил другой подход. Хотелось бы, чтобы ядро предоставляло более удобную для использования абстракцию, нежели копия аппаратуры. Поэтому дальнейшее развитие идеи изоляции заключается в том, чтобы повысить уровень абстракции для сервисов, предоставляемых через системные вызовы: ввести в них понятие потока и позволить пользователю их создавать, наделить ядро знанием об этих потоках, чтобы оно могло их планировать в том числе и на мультипроцессорных системах и т.д. Это позволило бы также создавать примитивы синхронизации, о которых ядро было бы в курсе, что позволило бы использовать их, в том числе, и для общения между процессами (Рис. 86).

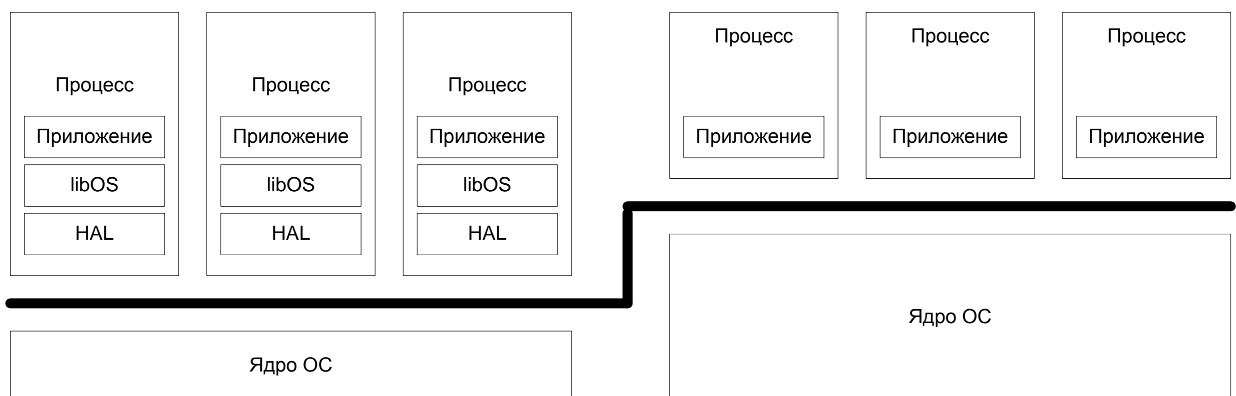

Рис. 86 Уровни абтракции виртуальной машины и разделяемого ядра.

Ядро такого типа называется **разделяемым** из-за того, что оно, в данном случае, как и оборудование, является разделяемым между всеми процессами ресурсом. Такое ядро является также некоторой абстрактной виртуальной машиной, но ее интерфейс определяется уже не в "аппаратных" терминах регистров и прерываний, а в терминах потоков, примитивов и так далее, поэтому, как правило, ядра такого типа, виртуальными машинами или гипервизорами уже не называются.

Единственным минусом такого подхода является многократно возрастающая сложность ядра. Оно должно предоставлять приложениям абстракцию потока, примитива синхронизации и увязывать эти абстракции с потенциально многопроцессорным оборудованием, взяв на себя всю низкоуровневую синхронизацию. Такое ядро сложно в разработке и тестировании, но обеспечивает более удобную коммуникацию между процессами и устраняет дублирование кода.

7.3.3 Корректность ядра при некорректном поведении процесса

В том случае, когда используются виртуальные машины, планировщик потоков пользователя выполняется с теми же привилегиями, что и само приложение. Если приложение ведет себя некорректно, нормальная работа внутренних механизмов планирования и обслуживания примитивов синхронизации может быть нарушена и процесс аварийно завершается. Менеджер виртуальной машины, выполняемый в привилегированном режиме, при этом не страдает и может перезапустить приложение и его внутренний планировщик вместе с ним. Совсем иначе обстоит дело с разделяемым ядром. Поскольку оно теперь разделяется между всеми процессами и работает в привилегированном режиме, некорректные действия любого из процессов могут вызвать ошибки в работе самого ядра и, следовательно, всех остальных процессов, что недопустимо. Причем ядро должно корректно обрабатывать в том числе и намеренные некорректные действия любого из процессов. В частности, это усложняет проектирование прикладного интерфейса ядра. Например, если какие-то функции предполагают определенную последовательность вызовов, ядро должно быть готово к тому, что пользователь намеренно эту последовательность не выполнит. Не освободит выделенную память, не размаскирует прерывания, если ему предоставить возможность их маскировки, не разрешит вновь вытеснение, после его запрета и так далее. Поскольку ядро является общей сущностью, практически все его функции имеют побочные эффекты, поэтому задача проектирование интерфейса ядра довольно сложна, многие операции, которые допустимы для приложений без разделения привилегий и ядра, компонуемого вместе с приложением, оказываются недопустимы в изолированных контекстах, потому что предполагают определенное поведение пользователя и зависят от него.

Более того, процесс может даже не иметь возможности что-то сделать согласно API. В случае разделяемого ядра, процесс, в общем случае, многопоточен. При этом ошибка, приводящая к аварийному завершению процесса может произойти только в одном потоке, а остальные должны быть принудительно завершены операционной системой. Завершение происходит в случайных точках кода потока, поэтому, даже в случае корректного написания кода, у процесса может попросту не быть возможности выполнить протокол, из-за завершения потока между двумя некоторыми вызовами. Это приводит к идее о том, что все ресурсы, используемые процессом, должны учитываться и контролироваться ядром, причем не только ресурсы памяти и процессорного времени, но и различных системных объектов.

7.4 Структура системы с разделяемым ядром

Несмотря на наличие пользовательских процессов, не всякое приложение должно

работать в непривилегированном режиме. В этом встроенные системы отличаются от ОС общего назначения, которые, как правило, не допускают выполнение никакого прикладного кода в режиме ядра, а все приложения запускаются внутри создающихся для этого процессах. В области встроенных систем, по соображениям производительности бывает целесообразно разместить часть приложения в привилегированном режиме, разумеется, при условии достаточной надежности данного кода. Дело в том, что хотя концептуально возможно сделать так, чтобы в привилегированном режиме работало только ядро ОС, на переключения между режимами тратятся драгоценные нано- и микросекунды, поэтому многие встроенные ОС допускают использование приложений двумя способами - прямой компоновкой с ядром, а также в виде непривилегированных процессов [5].

Поскольку в различных устройствах одно и то же приложение может быть как доверенным так и недоверенным, желательно использовать одно и то же API, как для приложений работающих в режиме ядра, так и в режиме пользователя. Поэтому, в первом приближении, структура системы с разделяемым ядром выглядит следующим образом (Рис. 87):

Рис. 87 Структура системы с разделяемым ядром.

Весь образ, включает в себя привилегированное приложение, статически скомпонованное с кодом ядра и выполняющееся в привилегированном режиме, а также набор из одного или более процессов, выполняющихся в непривилегированном режиме.

Так как в этом случае, ядро ОС знает о каждом пользовательском потоке и управляет ими, то исчезает необходимость в планировщике внутри каждого процесса. При этом, все рассуждения, касающиеся обработки прерываний, взаимодействия прерываний с потоками, схемы синхронизации и прочих тем, которые рассматривались в предыдущей главе, являются актуальными и в случае разделяемого ядра. В связи с этим, было бы желательно использовать имеющийся код ядра, а заодно использовать и соответствующее API.

Применяя те же рассуждения, которые использовались для обоснования того, почему потоки должны выполнять ядро непосредственно, можно прийти к выводу, что и пользовательские потоки также должны выполнять ядро в том же потоке. Такое поведение определяется в том числе и HAL. Системный вызов не приводит к переключению потока, а переключает режим процессора, и дальнейшее выполнение того же потока происходит уже в привилегированном режиме. За счет того, что системный вызов передает управление только в контролируемые ядром точки, пользовательские процессы лишены возможности выполнять в ядре произвольный код. Так как пользовательские потоки переключаются в режим ядра на время выполнения системных вызовов, с точки зрения ядра, все потоки выглядят единообразно и планируются одинаковым образом, одним и тем же планировщиком. Теоретически, возможна даже такая экзотическая ситуация, когда часть потоков, принадлежащих одному процессу выполняется в режиме ядра, а другая часть - в непривилегированном режиме. Таким образом, имеются два основных вида исполняемых сущностей (помимо обработчиков прерываний и DPC) - потоки, выполняемые целиком в режиме ядра и потоки, выполняющиеся в изолированных процессах, выполняющие код ядра в моменты выполнения системных вызовов.
Более детализированная структура системы принимает следующий вид (Рис. 88):

Рис. 88 Структура разделяемого ядра.

В случае разделяемого ядра, общение между ядром и приложениями происходит с помощью системных вызовов, реализация которых зависит от процессора и не может использоваться приложениями напрямую, поэтому приложение использует библиотеку, интерфейс которой имеет API используемого ядра, что позволяет использовать один и тот же интерфейс как для приложений выполняемых в режиме ядра, так и для непривилегированных процессов. Если функция не требует обращения к ядру, она может быть целиком реализована в библиотеке.

Рассмотренная реализация потоков предполагает, что памятью, используемой для объектов потока и примитива синхронизации, управляет пользователь. В то же время, пользовательские процессы не могут сами управлять памятью по соображениям безопасности, в противном случае можно с помощью некорректной работы с памятью вызвать нарушение работы планировщика. Поэтому функции управления потоками и примитивами, как правило, не могут использоваться приложениями напрямую. Между системным вызовом и реализацией потоков и прочих сущностей, должен находится слой, управляющий ресурсами. Как было показано ранее, этот слой называется исполнительной подсистемой. Управление памятью, используемых для системных структур данных, является критически важной работой, поэтому исполнительная система работает в привилегированном режиме и разделяется между всеми процессами. Хотя функции исполнительной подсистемы в большинстве своем предназначены для их вызова из пользовательского режима, некоторые из них могут использоваться как из потоков ядра, так и из привилегированного приложения.

Реализация управления ресурсами может быть реализована как дополнительный слой поверх реализации потоков, для этого не требуется изменения в, собственно, самих потоках и планировщике. Вопрос реализации исполнительной подсистемы будет отложен до следующего раздела, а пока стоит сосредоточиться на изменениях в наноядре для поддержки процессов и разделения режимов, при условии, что в HAL соответствующая поддержка имеется.

7.5 Поддержка непривилегированных потоков

Очевидно, должна иметься возможность создавать как потоки ядра, так и потоки пользователя, принадлежащие определенному процессу. При этом у кода, работающего в режиме ядра должна быть возможность создавать потоки обоих типов, тогда как у кода процессов должна иметься возможность создания только потоков в своем собственном процессе. Желательно также, чтобы эти изменения могли быть сделаны так, что в случае отключенной поддержки непривилегированного режима, весь связанный в нею код отключается бы на уровне исходных текстов и не привносил никаких накладных расходов.

Создание потоков, выполняющихся в пользовательском режиме процессора целиком реализуется с помощью HAL и не требует никаких изменений в реализации потоков. Как описывалось в главе, посвященной HAL, для того, чтобы поток выполнялся в непривилегированном режиме, после создания его контекста, этот контекст должен быть помечен как непривилегированный с помощью соответствующей функции. То есть гипотетический код для создания пользовательского потока мог бы выглядеть так:

```
/* Создание потока ядра. */
int error=kernel_thread_init(&thread, func, NULL, 0, kern_stack, kern_stack_sz, true);
if(!error)
{
  /* Модификация контекста потока для выполнения в пользовательском режиме. */
  cpu_context_create_unprivileged(
       &thread->context, user_entry, user_arg, user_stack_top);
  thread_resume(&thread);
}
```

С помощью функции thread_init создается поток, который будет выполняться в режиме ядра на указанном стеке. Далее его контекст модифицируется с помощью функций HAL, устанавливается пользовательский стек и новая точка входа. В подавляющем большинстве процессоров точка входа ядра func перезаписывается новой точкой входа в пользовательском режиме и не используется, хотя гипотетически можно представить ситуацию, когда поток начинает работу в режиме ядра и лишь затем переключается в пользовательский режим. Стек ядра не перезаписывается, а используется в качестве стека, на котором будет происходить выполнение системных вызовов.

Поскольку все отличия такого потока от потока выполняемого в режиме ядра заключены только в его контексте, с точки зрения планировщика, механизмов ожидания и всех прочих компонентов ОСРВ, пользовательские потоки ничем не отличаются от потоков ядра, возможные различия в переключении контекста на такие потоки, переключение стеков и так далее, заключены внутри HAL, в таких точках, как вход и выход из прерывания.

Отличие пользовательских потоков от потоков ядра заключается в том, что последние работают в условиях ограничений доступа к памяти, поэтому вместе с переключением потоков, потенциално может потребоваться переключение процесса, в том случае, если потоки принадлежат разным процессам. Необходимость выполнять это одновременно с переключением контекста потоков требует единственного изменения в коде реализации потоков. Кроме того, для выполнения операций переключения, ядро должно знать о принадлежности потока какому-либо процессу.

Управление процессами осуществляется с помощью компонента наноядра, реализующего функцию переключения:

```
void process_switch(process_t* next, process_t* prev);
```

В структуру потока также вводится указатель на процесс, которому этот поток принадлежит.

```
typedef struct _thread_t
{
  process_t* parent;
  ...
}
thread_t;
```

Для заполнения этой структуры может использоваться модифицированная функция создания потока, принимающая дополнительный аргумент.

```
int thread_init(process_t* parent, thread_t* thread, ...);
```

Так как, в этом случае, все потоки обязаны иметь родительский процесс, вводится понятие системного процесса. Этот процесс создается при инициализации системы и содержит потоки, выполняемые в режиме ядра. Он не имеет доступных адресов в пользовательском режиме и заменяется на загрушку в случае отсутствия поддержки

процессов.

С учетом вышесказанного, теперь легко понять, каким образом должно происходить переключение процесса. Это происходит в функции-обработчике программных прерываний, переключающей контекст.

```
void dispatch_handler(void) {
  /* Планирование потоков и обработка очереди DPC. */
  ...

  /* Если поток изменился ...*/
  if(next != context->current_thread) {
    /* Переключение процесса ...*/
    process_switch(next->parent, prev->parent);

    /* Переключение контекста потока. */
    ...
  }
  /* Доставка асинхронных процедур для текущего потока. */
  ...
}
```

Функция, переключающая потоки, перед переключением котекста анализирует процессы-владельцы потоков, и, если они различаются, вызывает функцию переключения процесса. Поскольку планировщик работает в режиме ядра, и его код отображен во все процессы, часть адресного пространства, содержащая планировщик, совпадает у всех процессов. Поэтому, в момент переключения меняется только отображение пользовательской части адресного пространства, которая станет актуальной только после возвращения в пользовательский режим и выхода из планировщика.

Для использования ядра без поддержки процессов, реализуется модуль-заглушка.

Таким образом, если поддержка процессов отключена, накладные расходы на его поддержку отсутствуют, если сравнивать с обсуждавшейся ранее реализациией потоков.

7.5.1 Переключение процесса

В том случае, если поддерживается всего один процесс, система защиты памяти (MMU или MPU) может быть настроена однократно во время инициализации системы и в дальнейшем никаких переключений не требуется, а значит, функция переключения может быть заменена заглушкой. Нетривиальная функция переключения процессов требуется только тогда, когда поддерживается выполнение нескольких процессов одновременно.

Механизм переключения отличается, в зависимости от использования MPU или MMU. В первом случае с каждым процессом должен быть ассоциирован набор регионов, описывающих права доступа к памяти. В момент переключения процессов, в MPU загружаются регионы из нового процесса.

```
void process_switch(process_t* new_process, process_t* old_process)
{
  if(current_process[get_current_cpu()] != new_process)  {
    current_process[get_current_cpu()] = new_process;
```

```
    // Перепрограммирование MPU и установка новых атрибутов доступа к памяти...
    mpu_region_set_access(
      code_index, new_process->code.addr, new_process->code.size, ...);
    ...
  }
}
```

Каждый процесс использует некоторый ограниченный набор регионов MPU, например два: для описания региона кода и данных. В MPU резервируется два региона с определенными индексами (в коде выше это code_index, data_index) и ядро ОС обеспечивает правильное перепрограммирование MPU всякий раз при переключении процесса.

В зависимости от программной модели MPU, его программирование может включать довольно много работы, поэтому в системах с MPU в целом рекомендуется использование только одного процесса с однократной настройкой защиты памяти.

Более распространенным, хотя и отличающимся меньшей предсказуемостью и производительностью, вариантом организации защиты памяти является использование MMU. В этом случае, с каждым процессом ассоциирован набор таблиц, описывающих отображение виртуального адресного пространства на физическое, а также права доступа. Если используется аппаратно-поддерживаемая трансляция страниц, то переключение обычно происходит путем замены указателя на текущую таблицу трансляции в соответствующем регистре процессора. Какая именно часть адресного пространства будет переключена, а также неизменяемые при переключениях области, в которых должен находиться код планировщика и ядро ОС, определяется HAL.

В простейшем случае, без использования всевозможных оптимизаций и ASID, переключение процесса с использованием MMU выглядит так:

```
void process_switch(process_t* new_process, process_t* old_process)
{
  if(current_process[get_current_cpu()] != new_process)  {
    current_process[get_current_cpu()] = new_process;
    mmu_switch(new_process->pagedir);
  }
}
```

В обоих случаях, как с MMU так и с MPU возможна также дополнительная оптимизация связанная с системным процессом. Поскольку его адресное пространство присутствует во всех других процессах, при переключении на поток ядра (поток у которого системный процесс является родительским процессом) можно не переключать процесс так что в представленных примерах есть большое пространство для оптимизаций. Тем не менее, суть остается неизменной независимо о используемой ОС.

7.5.2 Обработчики завершения потока

Наконец, последним механизмом, который стоит рассмотреть до начала обсуждения исполнительной подсистемы - cleanup-обработчики. По большому счету, необходимость в их использовании может возникнуть и без использования процессов, так что они являются самоценным механизмом, имеющим ценность даже для прикладного ПО. В

частности, такие обработчики включены в стандарт POSIX [6]. Тем не менее, если использование таких механизмов совместно с небольшими встроенными ядрами, это скорее исключение, нежели правило, то при поддержке процессов и системного управления памятью, данный механизм становится обязательным, поэтому будет рассматриваться в данной главе.

В процессе работы поток может выполнять действия, которые имеют побочные эффекты и могут привести к некорректному поведению системы, если поток неожиданно завершится. Например, если была выделена память, это предполагает, что кто-то должен будет ее освободить. Если поток, по каким-либо причинам, был штатно либо аварийно завершен до этого момента, память не будет освбождена. Подобные рассуждения распространяются также и на некоторые примитивы синхронизации, такие как мьютексы. Было бы довольно полезно, в момент совершения действия с побочным эффектом, сопоставлять с потоком некий обработчик, который выполнялся бы в случае завершения потока. Подобный функционал имеется в некоторых языках программирования в виде блоков try-catch-finally, которые дают возможность выполнить какие-то дополнительные действия, в случае возникновения исключений. Возможность выполнения пользовательских действий при завершении потока также оказывает помощь в отладке, поскольку можно проверить корректность действий и отсутствие ошибок в освобождении ресурсов. Так как поток может завершиться в любой точке своего выполнения, то подобная поддержка с учетом возможности асинхронного завершения потока, может быть реализована только на системном уровне, то есть внутри функции ядра, выполняющейся при завершении потока. Поскольку добавление и удаление таких обработчиков, как и их выполнение может производить только один поток, то никакая синхронизация не требуется, необходимо только запрещать вытеснение на время манипуляций со структурой данных. Это нужно, чтобы исключить завершение потока во время работы со списком обработчиков, что может повлечь попытку выполнить обработчики, пользуясь потенциально некорректной структурой данных. Интерфейс выглядит достаточно тривиально:

```
void cleanup_init(
  cleanup_handler_t* handler,
  void (*func)(cleanup_context_t*, cleanup_handler_t*, void*), void* arg);

void cleanup_add(cleanup_context_t* target, cleanup_handler_t* handler);
bool cleanup_cancel(cleanup_handler_t* handler);
```

Обработчики формируют стек, выполняемый в порядке, обратном порядку их добавления, то есть обработчики, установленные позже, выполнятся раньше.

На первый взгляд, создается впечатление, что обсуждавшиеся ранее процедуры отмены активных таймеров и отмены ожидания, выполняемые при завершении потока, также логически являются такими же cleanup-обработчиками, и, следовательно, все эти действия могут быть реализованы через единый механизм. На самом деле, несмотря на некоторое сходство, существует также и важное отличие: отмена таймеров и ожидания, а также прочих системных побочных эффектов, должна выполняться не только в случае

завершения потока, но и в случае, например, доставки асинхронной процедуры или сигнала, обработчик которого должен работать в контексте, в котором все побочные эффекты потока отменены. Напротив, cleanup-обработчик определяется именно как обработчик, выполняющийся только в случае завершения потока, поэтому обработка цепочки таких обработчиков не выполняется в случае доставки асинхронных процедур, в чем и состоит главное различие.

Так как стек потока остается пригодным для использования все время работы потока, включая процедуру удаления, объекты cleanup-обработчиков могут быть размещены на стеке потока, что позволяет приложению создавать такие объекты и пользоватся ими по мере надобности.

Что касается реализации, то в наноядре требуется всего несколько модификаций: во-первых, список таких обработчиков нужно добавить в структуру потока, так как каждый поток имеет свой собственный список; во-вторых, в APC-процедуру, выполняющую удаление потока из планировщика, перед удалением необходимо добавить функцию, выполняющую обработчики:

```
void thread_termination_handler(...)
{
  cleanup_handle(thread_as_cleanup_context(me));

  sched_lock(...);
  thread_exit_sync(...);
  sched_unlock(...);
}
```

Поскольку предполагается, что выполнение отменяющих действий для потока может потребовать обращения к системному API, обработчик должен работать в том же контексте, что и поток. То есть, он должен выполняться в контексте потока, на его стеке и так далее. Здесь может возникнуть трудность, связанная с оптимизациями. Как говорилось в предыдущей главе, с целью повышения производительности в однопроцессорных системах, может применяться оптимизация, связанная с выполнением кода асинхронной процедуры внутри любого потока, при условии заблокированного планировщика. Это дает возможность избавиться от сложного механизма доставки асинхронных процедур. Функции, используемые при завершении потока, такие как извлечение его из планировщика, могут использоваться на любом SPL, как на DISPATCH так и на LOW, что и позволяет использовать данную оптимизацию. Поскольку cleanup-обработчики работают в составе той же самой асинхронной процедуры, то использование упрощенной схемы работы APC приводит к тому, что обработчики будут вызваны на повышенном уровне SPL, и, возможно, в контексте того потока, который пытается завершить выполнение другого, с которым связаны обработчики. Хотя с синхронизацей в данном случае все в порядке, так как упрощенные механизмы APC используются только в однопроцессорных системах, могут возникнуть проблемы с использованием функций ожидания на повышенных уровнях SPL. Поэтому, стоит проанализировать, какие обработчики могут быть вызваны. Если допускаются вызовы только такого API, которое допустимо использовать на уровне SPL как LOW, так и DISPATCH, то допустимо

использовать упрощенную схему и выполнять обработчики на повышенном SPL. В противном случае следует использовать механизм доставки APC, предполагающий выполнение обработчика в контексте потока на уровне LOW.

Для большей универсальности, если предполагается использование cleanup-обработчиков, рекомендуется использовать полноценную реализацию APC, выполняющую обработчики в контексте потока. Использование cleanup-обработчиков с упрощенной схемой не рекомендуется еще и потому, что требует выполнения в невытесняемом контексте заранее неизвестного количества работы, что негативно влияет на время реакции и прочие характеристики работы в реальном времени.

7.6 Структура исполнительной подсистемы

Как и в случае с многими другими терминами из области разработки ОС, с понятием исполнительной подсистемы связано довольно много путаницы. В данной книге под исполнительной подсистемой будет подразумеваться обобщенное понятие: слой API, работающий в режиме ядра, который предоставляет любые более высокоуровневые операции, по сравнению с традиционным функционалом наноядра, занимающегося реализацией потоков, планировщика и примитивов синхронизации. В случае монолитной архитектуры ядра, среди этих сервисов могут быть поддержка файлов, драйверов устройств, сетевых интерфейсов и тому подобного. В случае микроядерной архитектуры, исполнительная система, в основном, занимается управлением ресурсами для объектов ядра, таких как потоки, а также предоставляет интерфейс для приложений, работающих в непривилегированном режиме. Из этих задач следует то, что исполнительная система, помимо прочего, должна обеспечивать корректное использование ресурсов и интерфейсов потоков и HAL и обрабатывать попытки их некорректного использования пользовательскими процессами. То есть, в задачи этой подсистемы входит подсчет ссылок на объекты, проверка аргументов, приходящих из непривилегированных процессов, обработка системных вызовов и так далее.

Исполнительная система содержит два вида API. Одно подмножество функций доступно только для использования из режима ядра и предоставляется привилегированым приложениям. Например, в рассматриваемых примерах к этому виду функций относятся возможность запускать и останавливать процессы. Функции, предназначенные для использования привилегированными приложениями режима ядра реализуется так же, как и все другие функции компонентов ядра: вызовы функций связываются во время сборки образа. Другая категория функций доступна непривилегированным приложениям через интерфейс системных вызовов. Реализацией пользовательского API занимается компонент исполнительной системы называемый диспетчером системных вызовов. Он является сердцем инфраструктуры поддержки процессов и содержит обработчик системных вызовов, который вызывается HAL в ответ на использование системных вызвов процессом.

Типичный обработчик системных вызовов обычно использует один из доступных аргументов в качестве идентификатора функции, которую нужно вызвать, поэтому код выглядит следующим образом:

```
register_t syscall_handler(
  register_t a, register_t b, register_t c, register_t d, register_t e, register_t f)
{
  static register_t ( * const syscall_table[] )
    (register_t,register_t,register_t, register_t,register_t) =   {
    ...
  };

  static const uint32_t syscall_n = sizeof(syscall_table) / sizeof(syscall_table[0]);
  register_t result;

  if(a < syscall_n)  {
    result = syscall_table[a](b,c,d,e,f);
  }
  ...
  return result;
}
```

В таблице системных вызовов содержатся указатели на функции-обработчики. Поэтому в общих чертах структура исполнительной подсистемы выглядит как показано на Рис. 89:

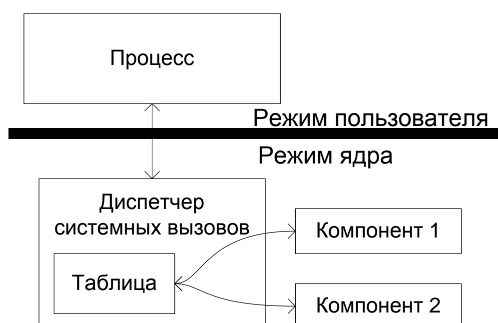

Рис. 89 Структура исполнительной подсистемы.

Все компоненты, работающие в режиме ядра, составляют набор компонентов, которые занимают определенные ячейки в таблице системных вызовов.
Когда приложение совершает системный вызов, первоначально управление получает HAL, затем, после выполнения зависящих от оборудования действий, таких как переключение стеков, вызывается кросс-платформенная функция-обработчик системных вызовов из исполнительной подсистемы. Обычно, отображение пользовательских запросов на функции ядра происходит с помощью таблицы. В качестве аргумента системного вызова указывается некоторый идентификатор функции, например индекс в таблице, и набор аргументов этой функции. После того, как управление передано в ядро, диспетчер системных вызовов может проанализировать переданный идентификатор и определить является ли он корректным, то есть соответствует ли он какой-то функции. Если указан некорректный идентификатор, то приложение обычно аварийно завершается, хотя, в некоторых случаях, в зависимости от целей, он может быть и проигнорирован. С помощью такого механизма реализуется контроль точек входа в ядро со стороны пользователя. Пользователь может вызывать только те функции, которые перечислены в таблице и вызывать их именно по тому адресу, который указан.
Некоторые ОС позволяют динамически добавлять системные вызовы в таблицу, хотя такие возможности требуются редко. Тем не менее, при необходимости, это может быть

реализовано путем создания таблицы фиксированного размера, большего, чем количество системных вызовов, то есть таблица будет содержать пустые ячейки. Если вызов происходит в тот момент, когда ячейка пуста, это может обрабатываться так же, как и некорректный системный вызов, если же там находится указатель на функцию, системный вызов считается легитимным.

Таблица системных вызовов позволяет решить и другую задачу - задачу трансляции адресов. Если функции ОС вызываются напрямую по адресу, как это происходит в случае libOS, образ ПО нужно пересобирать всякий раз, когда изменяется распределение памяти. В частности, нельзя просто так заменять приложения и распространять их в виде бинарных модулей, поскольку перед запуском приложению необходимо узнать адреса функций ОС, которые оно собирается вызывать. Вызов функций по идентификатору вводит дополнительный слой косвенности, который позволяет отвязать приложения от конкретных адресов в памяти, что делает их более абстрактными и позволяет переносить приложения в виде бинарных модулей без необходимости каждый раз компоновать их с ядром ОС при любых изменениях. Это является причиной того, что иногда механизм, аналогичный системным вызовам, применяется и в том случае, когда не предусмотрено разделения привилегий, это бывает необходимо для реализации обновлений и необходимостью отвязать бинарный модуль от конкретных адресов функций.

Поскольку путем манипуляций с памятью, используемой для системных объектов, можно нарушить стабильность работы ядра и даже вызвать его крах, память под системные объекты не предоставляется пользователем, а выделяется системой. Поэтому исполнительная подсистема, как правило, содержит как минимум средства для управления памятью, выделения и освобождения памяти под системные объекты.

Так же, как и реализация потоков, диспетчер системных вызовов предоставляет инфраструктуру, поверх которой могут добавляться дополнительные компоненты, по аналогии с тем, как интерфейс потоков содержит инфраструктуру для реализации ожидания, позволяющую добавлять неограниченное количество примитивов.

7.7 Реализация системных вызовов

Простейшие системные вызовы, особенно те, которые не имеют аргументов, выполняют только функцию трансляции пользовательского запроса к соответствующим системным сервисам, например системный вызов для приостановки потока мог бы выглядеть так:

```
register_t syscall_thread_suspend(
  register_t p0, register_t p1, register_t p2, register_t p3, register_t p4)
{
  return thread_suspend();
}
```

В некоторых случаях, в целях упрощения синхронизации внутри ядра, используются невытесняемые системные вызовы. То есть, перед началом выполнения системного вызова, происходит захват некоторой блокировки (спинлока или мьютекса) и дальнейшее выполнение происходит в невытесняемом контексте (подход, известный как BKL: big kernel lock [7]). В системах реального времени такой подход применять нельзя, поэтому,

когда поток выполняется в режиме ядра, он ничем не отличается от прочих потоков и может вытесняться потоками с более высоким приоритетом как из текущего процесса так и из других. Если вытеснение потока происходит в тот момент, когда он выполняется в пространстве пользователя, сохранение контекста происходит на стек ядра данного потока. Таким образом, обработчик системного вызова не должен делать никаких предположений относительно собственной вытесняемости.

7.7.1 Использование объектов ядра

Системные вызовы не использующие аргументы просты, но являются меньшинством. Большинство функций ОС все-таки работают с передаваемыми в них объектами. Поэтому, другой важный вопрос, после вопроса о том, как обеспечить контроль точек входа в ядро: как гарантировать корректность аргументов. Поскольку ряд функций ядра принимают указатели на объекты, с которыми они работают, такие как, например, потоки или примитивы синхронизации, вся ответственность за корректность объекта и само его существование возлагается на пользователя. В случае изолированных процессов, это допущение, очевидно, приводит к проблемам с безопасностью. Можно случайно либо целенаправленно вызвать функцию, передав ей некорректный объект и, тем самым, вызвать нарушение работы ядра и остальных процессов. Поэтому пользователь не должен иметь возможность:

- доступа к памяти объектов ядра
- удаления объекта в то время, как он используется
- передачи ядру некорректного объекта.

Далее будут рассмотрены методы реализации этих гарантий.

7.7.2 Выделение памяти

Прежде чем использовать объект ядра, он должен быть определенным образом выделен и проинициализирован. В том случае, когда памятью управляет пользователь, это выглядит довольно тривиально, если же используются процессы, нужны дополнительные меры, направленные на защиту ядра от возможной злонамеренности процесса.

По соображениям обсуждавшимся выше, память под объекты ядра должна выделяться независимо от пользователя и этот процесс должен происходить в режиме ядра, для предотвращения компрометации этой памяти и состояния объекта из режима пользователя. Один из вариантов - использование специального пула памяти, который инициализируется во время старта ядра. Его размер может задаваться конфигурационными методами, либо вычисляться в процессе работы в зависимости от объема имеющейся памяти.

Интерфейс для выделения памяти под объекты похож на интерфейс блоков памяти обсуждавшийся ранее. Некоторое количество памяти передается в пул при его инициализации, в дальнейшем пул используется для "нарезки" этой памяти на блоки некоторого размера.

```
void obj_pool_init(obj_pool_t* pool, void* pool, size_t pool_sz);
void* obj_alloc(obj_pool_t* pool, size_t obj_sz);
bool obj_free(void* obj);
```

При необходимости, интерфейс можно свести к глобальному пулу. Так как при инициализации указывается требуемый размер блока, можно использовать функции работы с пулами памяти в качестве обёрток вокруг некоторого универсального аллокатора.

```
void* obj_alloc(fx_obj_pool_t* pool, size_t obj_sz) {
  return malloc(obj_sz);
}
```

Так как этот механизм используется для выделения объектов, представленных некоторой структурой, целесообразно ввести тип заголовка, хранящий данные, связанные с пулом и определить интерфейс таким образом, чтобы всякий объект, который выделяется с помощью пула, содержал такой заголовок в качестве первого элемента структуры. Подобные заголовки неявно используются и в случае с примитивом "блоки памяти".

```
typedef struct _some_allocatable_object_t
{
  obj_rsrc_t header;
  ...
};
```

Поскольку выделение памяти происходит в ядре и пользователь лишен возможности доступа к этой памяти, это решает проблему с вмешательством пользователя во внутреннюю структуру объектов.

7.7.3 Подсчет ссылок на объекты

Второй упомянутый вопрос - обеспечение должного времени жизни объекта и лишение пользователя возможности удалить объект в то время, когда он используется. Существует несколько подходов к решению этой задачи, например, в однопроцессорной ОС может использоваться запрет вытеснения на время работы с объектами ядра. Более распространный подход, который можно уже считать классическим, для решения такого рода задач - подсчет ссылок.

Суть этого метода заключается в том, что каждый объект содержит счетчик ссылок на себя, который увеличивается перед использованием объекта и уменьшается после использования. Объекты создаются с начальным значение счетчика равным единице. Фактическое удаление в таком случае происходит в тот момент, когда счетчик обнуляется, то есть, когда объект никто не использует, а также после того, как было запрошено его удаление.

Данная техника имеет несколько слабых мест, которые стоит вкратце обсудить. Во-первых, требуется, чтобы в момент увеличения счетчика ссылок объект был в корректном состоянии. То есть тот поток, который производит увеличение счетчика, должен быть уверен, что счетчик больше нуля. Во-вторых, механизм зависит от корректного использования, то есть не может быть доступен пользователю, так как, в противном случае, можно искусственно вызвать удаление объекта просто уменьшив счетчик ссылок несколько раз. В-третьих, есть проблема, когда прямо или косвенно объект может

удерживать ссылку на самого себя, что будет препятствовать его удалению, поэтому система в целом должна быть спроектирована так, чтобы исключить возможность такого сценария.
Поскольку каждый объект, доступный пользователю, должен использовать подсчет ссылок, а также одновременно является объектом, выделяемым с помощью описанного выше механизма работы с памятью, целесообразно определить заголовок абстрактного объекта, включающий как счетчик ссылок, так и заголовок, связанный с памятью.

```
typedef struct _object_header_t
{
  ...
  volatile uint32_t ref_count;
  void (*dtor)(void*);
  unsigned int type;
}
object_header_t;
```

Объекты исполнительной подсистемы, таким образом, приобретают следующий общий вид:

```
typedef struct _some_object_t
{
  object_header_t header;
  ...
}
some_object_t;
```

Многие реализации предполагают хранение внутри объектов еще ряда дополнительной информации, такой как, например, указатель на деструктор. Что касается интерфейса, относящегося к самим объектам, он выглядит также довольно просто:

```
void object_init(object_header_t* object, ...);
void object_ref(object_header_t* object);
bool object_deref(object_header_t* object);
```

Функция инициализации используется для установки заголовка любого объекта в корректное состояние, а также установки счетчика ссылок в начальное значение равное 1. Две другие функции используются для увеличения счетчика ссылок и его уменьшения соответственно.
Всякий раз, когда счетчик ссылок на объект достигает нуля (это значит что данный декремент - последний и в системе больше нет ссылок на данный объект), его нужно удалять. Вызов деструктора объекта может производиться несколькими путями. Основных способа можно выделить два: в первом случае вызов деструктора осуществляется синхронно с функцией декремента ссылок (то есть, эта функция вызывает деструктор в том контексте, в котором она сама была вызвана). Описанный интерфейс предполагает именно такое поведение. Второй случай предполагает асинхронный вызов, то есть существует дополнительный поток, или какая-то иная выполняемая сущность, которой передаются объекты, которые нужно удалить. Например можно включать

удаляемый объект в список, который просматривался бы потоком-чистильщиком. Оба подхода имеют свои достоинства и недостатки, поэтому бывает сложно сделать какой-то однозначный выбор. В первом случае накладывается ограничение на контекст, в котором может быть вызвана функция уменьшения ссылок. Важно понимать, что в процессе работы деструктора, может потребоваться ожидание и блокировка потока, как в описанном выше примере. Поэтому, если деструктор устроен таким образом, то, очевидно, нельзя освобождать объекты в контексте обработчиков прерываний или DPC.
С другой стороны, подход с дополнительными потоками вносит в систему накладные расходы и добавляет неопределенности. Существование только одного такого потока может привести (в многопроцессорной системе) к тому, что он станет узким местом в случае одновременного завершения нескольких процессов и необходимости удаления большого количества ресурсов. Неопределенность связана с тем, что вызвавшему деструктор коду становится неизвестно, когда этот деструктор фактически отработает, а память будет освобождена.
На фоне общей проблемы подсчета ссылок на объекты, особняком стоят потоки. С одной стороны, поток является таким же объектом, как и все прочие: ему соответствует блок памяти, выделяемый с помощью сооветствующих сервисов, у него есть конструктор, деструктор и все прочие атрибуты, свойственные объектам. Вместе с тем, поток имеет существенное отличие от всех остальных объектов исполнительной системы: он является активной исполняемой сущностью. В то время, пока поток выполняется, очевидно, освобождать используемую им память нельзя, даже в том случае, когда в системе больше нет ссылок на этот объект и когда было запрошено удаление. То есть получается, что поток, пока он выполняется или находится в готовом к выполнению состоянии, как бы удерживает ссылку на самого себя. Следующий вопрос, который немедленно возникает после решения об удержании потоком ссылки на самого себя, заключается в том, кто и когда должен удалять объект потока, раз сам поток сделать этого не может. После завершения потока, последней сущностью, которая с ним взаимодействует внутри ядра ОС это планировщик. Именно он и должен обеспечить финальный декремент количества ссылок на поток, так как в этом контексте удаление потока уже безопасно. Для этого может быть предусмотрен специальный вызов внутри обработчика программного прерывания, который вызывался бы всякий раз, когда очередной поток завершился (это известно планировщику по состоянию потока).
В методе подсчета ссылок на потоки, обсуждавшемся выше, требуется модификация нижележащих слоев ПО, с целью обеспечить вызов некоторой дополнительной функции в контексте обработчика программного прерывания после завершения потока. Данное решение оптимально в том смысле что обеспечивает особождение ресурсов сразу, как только это стало возможным. Тем не менее, есть и альтернативный взгляд на эту проблему, не требующий модификаций в реализации потоков. В этом случае, удаляемый поток, в то время когда он еще работает, но уже известно, что он должен быть завершен, вносится в некоторую глобальную структуру данных, например список. Это может происходить в системных вызовах, реализующих синхронное или синхронное завершение потока. Важно понимать, что в этом списке могут находить потоки как уже завершившиеся, так и те, которые еще работают. Далее, в функции создания нового

потока, когда для него требуется выделить память под объект, извлекается элемент из глобального списка завершающихся потоков и ожидается завершение с помощью fx_thread_join или ее аналога, после чего извлеченный объект переиспользуется для нового потока.

7.7.4 Идентификаторы объектов

Подсчет ссылок решает проблему существования объекта и неограниченного продления времени его жизни, при условии, что в момент увеличения счетчика ссылок объект существовал. Здесь существует неочевидная на первый взгляд проблема: как гарантировать существование объекта в момент увеличения счетчика. После создания объекта пользователю может передаваться указатель на объект ядра (который можно использовать только как идентификатор объекта, поскольку доступ к памяти ядра отсутствует), но ничто не мешает пользователю попытаться удалить объект, а потом попытаться использовать указатель на уже удаленный объект, или же просто передать в функции ядра заведомо некорректный указатель.

Один из подходов к решению этой проблемы - проверка существования в системе запрошенного объекта. В этом случае, вместе с созданием объекта, он помещается в некоторую структуру данных (например, в хэш-таблицу или список), содержащую все объекты данного типа. Когда через механизм системного вызова передается указатель на объект, полученный указатель используется как ключ поиска в соответствующей хэш-таблице. Если результат положительный, то есть в таблице присутствует такой объект, можно не сомневаться, что объект корректен, поэтому можно увеличить счетчик его ссылок. В противном случае вызов завершается ошибкой.

Этот механизм может быть реализован на основе описанного интерфейса подсчета ссылок. Вначале блокируется хэш-таблица, в то время пока она заблокирована, объекты не могут быть из нее удалены, а значит, их память корректна. Найдя объект, его счетчик ссылок увеличивается, таблица разблокируется и объект используется как обычно. Удаление объекта предусматривает предварительное удаление его из таблицы. После этого не может возникнуть новых ссылок на объект, а корректное удаление и освобождение памяти произойдет после того, как завершатся все текущие работы с объектом и счетчик его ссылок будет необходимое число раз декрементирован.

Интерфейс такого контейнера (или базы данных) корректных объектов в системе мог бы выглядеть следующим образом:

```
void handle_db_init(handle_db_t* db);
void handle_db_close(handle_db_t* db);

bool handle_insert(handle_db_t* db, unsigned type, unsigned int* handle, void* obj);
void* handle_ref_object(handle_db_t* db, unsigned type, unsigned int handle);
bool handle_close(handle_db_t* db, unsigned int handle);
```

Первые две функции используются как конструктор и деструктор хранилища и поэтому не очень интересны.

После того, как память объекта выделена и он сам инициализирован и готов к работе, он должен быть помещен в таблицу, для чего используется функция handle_insert. Она

возвращает некоторый идентификатор, который используется в качестве ключа поиска. Это может быть сам указатель, а может быть и иное значение, в зависимости от используемой структуры данных. В качестве входных данных функция использует указатель на объект и его тип. Информацию о типе можно хранить и внутри объекта, этот вопрос остается на усмотрение разработчика.

Когда возникает необходимость проверить корректность объекта данного типа, используется следующая функция. По указанному идентификатору и ожидаемому типу объекта она пытается найти в хранилище и увеличить счетчик ссылок на объект, вернув в вызывающую функцию указатель, который гарантированно указывает на существующий объект данного типа (либо NULL, если такого объекта не нашлось).

Наконец, последняя функция используется для удаления объекта из таблицы с уменьшением счетчика ссылок на него. Используется она тогда, когда пользователь решает удалить объект и освободить его идентификатор.

Хотя вариант с хэш-таблицами в целом приемлем и используется в коммерческих ОС, с ним связаны очевидные проблемы соответствия критериям работы в реальном времени. Время работы ограничено сверху, но вызовы одной и той же функции для двух объектов могут довольно сильно отличаться по времени выполнения из-за случайных факторов, таких как фактические адреса указателей и текущее заполнение хэш-таблиц. Поэтому, в системах реального времени большее распространение получил вариант основанный на применении таблицы идентификаторов (handle table).

В этом случае проблема решается по аналогии с контролем точек входа в системные вызовы - введением дополнтельного уровня косвенности, который дает возможность проверить корректность переданного объекта. Для объектов существует таблица, по аналогии с таблицей системных вызовов (Рис. 90). При создании объекта он заносится в эту таблицу, а пользователю возвращается идентификатор (индекс в таблице).

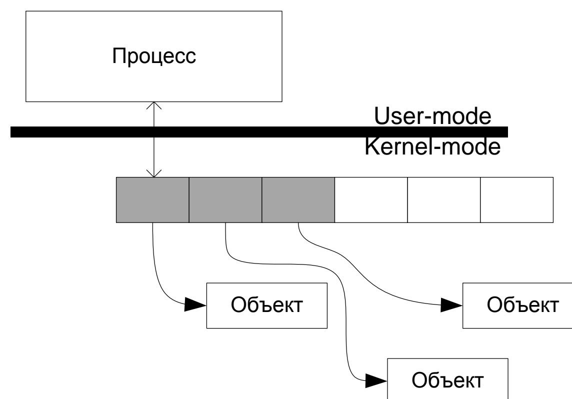

Рис. 90 Таблица идентификаторов для доступа к объектам ядра.

При этом улучшаются параметры производительности: поиск элемента и выяснение корректности идентификатора всегда выполняется за константное время. Но есть и недостатки, такие как ограниченный размер таблицы, и, как следствие, ограниченное количество объектов, к которым может обращаться приложение. Существуют способы динамического роста размера таблицы: идентификатор может разделяться на несколько частей, например, индекс таблицы и индекс в таблице и тому подобные решения.

Такой подход позволяет наращивать количество записей в таблицах и количество самих таблиц в зависимости от потребностей приложения. Тем не менее, все эти методы связаны с непредсказуемыми заранее задержками: нельзя сказать, когда именно потребуется инициализация новой таблицы, поэтому для системы жесткого реального времени лучше пойти на некоторый компромисс - ограничить размер таблицы числом объектов, достаточным для большинства применений, либо подстраивать ее размер под конкретное приложение с помощью конфигурирования.

С учетом всего вышесказанного, код для создания и работы с объектами ядра в общем виде выглядит, как показано ниже:

```
/* Выделение памяти для объекта. */
object = obj_alloc(...);
...

/* Инициализация заголовка и тела объекта. */
object_init(object);

/* Вставка идентификатора в таблицу. */
error = handle_insert(..., type, &handle, object);

/* Обработка ошибок и возврат идентификатора созданного объекта. */
...
```

Важным моментом здесь является добавление объекта в таблицу именно после полной и корректной инициализации, так как непосредственно после добавления, объект теоретически может быть использован другими потоками, выполняющимися на других процессорах, поэтому попадание в таблицу не до конца сконструированного объекта потенциально способно спровоцировать некорректную работу ядра.

После добавления в таблицу возможно использование объекта другими потоками в этом процессе. Проверка корректности аргумента выполняется так:

```
/* Проверка корректности идентификатора и получение ссылки на объект. */
void* object = handle_ref_object(&handle_db, type, handle);

if(object != NULL)
{
  /* Использование объекта. */
  ...

  object_deref( object);
}
```

Таблицы идентификаторов, помимо проверки корректности объектов, выполняют также другую важную функцию. С их помощью проверяется принадлежность объекта запрашивающему процессу, а также может реализовываться контроль доступа. Например, в случае использования глобальной таблицы, один из процессов может попытаться "угадать" идентификатор объекта, который создал и использует другой процесс, то есть попытаться вмешаться в его работу, и, возможно, помешать корректному функционированию. По этой причине таблицы идентификаторов привязываются к

процессу, а не делаются глобальными. То есть при создании объекта, он попадает в таблицу только этого процесса и попытки использования некорректных идентификаторов заранее обречены на провал.
Идентификаторы объектов активно используются пользовательскими потоками. Любой доступ к любому объекту должен проходить через проверки, реализованные в этом слое, поэтому реализация таблиц идентификаторов имеет критическое значение для производительности. Этим обусловлено больше количество оптимизаций, применяемых при реализации этих таблиц.
Многие встроенные системы не предполагают завершения работы устройства. В таких условиях целесообразно удалить из системы возможности удаления объектов, так как с ними связаны проблемы синхронизации. При таком подходе единожды созданный объект существует все время работы системы и его идентификатор всегда остается корректным. Поскольку использование объекта до того, как он добавлен в таблицу, невозможно, то необходимо синхронизировать только доступ к хранилищу для добавления в него записей. После того, как запись добавлена, она не может ни измениться, ни быть удаленной, то есть для проверки корректности достаточно обычных операций чтения и сравнения с нулем.
Если же создание и удаление объектов путем закрытия идентификаторов возможно во время работы, необходимо должным образом синхронизировать доступ к таблицам с учетом одновременного доступа к ним многих потоков. С целью снижения конкуренции между потоками, имеет смысл использовать не блокировку уровня всей таблицы, а снабжать индивидуальными блокировками каждую ячейку. Работает это так: поскольку таблица предполагает размещение внутри нее только указателей, младшие их биты всегда будут равны нулю, поскольку указатели на объекты ядра всегда должны быть выровнены как минимум на размер машинного слова. Можно использовать младший бит в виде некоторого подобия спинлока, защищающего данную ячейку таблицы. Увеличение счетчика ссылок на объект это, как правило, очень быстрая операция, включающая буквально несколько инструкций, поэтому выполнение ее под защитой спинлока вполне оправдано. Если предположить, что таблица идентификаторов индексируется идентификатором объекта, то содержательная часть функции handle_ref_object могла бы выглядеть так:

```
/* Повышение SPL до DISPATCH, проверка нахождения идентификатора в пределах. */
...
bool success = false;
uintptr_t ptr;

do {
  ptr = (uintptr_t) db->table[handle];

  /* Ячейка содержит NULL, идентификатор некорректен. */
  if (!ptr)
    break;

  /* Если младший бит установлен, переход на следующую итерацию цикла (ожидание).*/
  if ((ptr & 1) == 0) {
```

```
    /* Если младший бит сброшен, попытка его установить с помощью CAS.*/
    uintptr_t prev = atomic_cas_ptr(&db->table[handle], ptr, ptr | 1);
    if (prev == ptr)
      success = true;
  }
} while (!success);

if (success) {
  object_header_t* object = (object_header_t*) ptr;
  object_ref(object);

  /* Сброс младшего бита, разблокировка ячейки таблицы. */
  db->table[handle] = (void*) ptr;
}
```

С помощью операции CAS функция пытается установить младший бит, при условии, что данная ячейка занята (не равна NULL). Если в процессе этого идентификатор закрывается, из функции возвращается ошибка и на этом выполнение завершается. Если же бит удалось установить, это гарантирует, что все остальные функции, пытающиеся что-то сделать с данной ячейкой в других потоках, будут ждать, поэтому можно увеличить счетчик ссылок и сбросить бит занятости ячейки. Сразу же после этого идентификатор может быть закрыт, но его удаления не произойдет до тех пор, пока счетчик ссылок не будет уменьшен. Из этих рассуждений становится понятно, почему идентификатор должен соответствовать ровно одной ссылке на объект и его нельзя использовать для того, чтобы сделать несколько декрементов счетчика ссылок.

Закрытие идентификатора работает по аналогии. После того, как удалось освободить ячейку таблицы, владение ссылкой на объект переходит к данному потоку, который и делает декремент.

```
/* Сброс ячейки таблицы в 0, с ожиданием, если ячейка заблокирована. */
do {
  ptr = (uintptr_t) db->table[handle];

  if(!ptr)
    break;

  if ((ptr & 1) == 0) {
    uintptr_t prev = atomic_cas_ptr(&db->table[handle], ptr, NULL);
    if (prev == ptr) {
      success = true;
    }
  }
} while(!success);

if (success) {
  object_header_t* object = (object_header_t*) ptr;
  db->table[handle] = NULL;
  ...
  object_deref(object);
  ...
}
```

Пока ячейка заблокирована, она не может быть закрыта. Как только младший бит устанавливается в 0, может начаться процесс закрытия ячейки. Несколько потоков могут

пытаться одновременно закрыть одну и ту же ячейку, в этом случае необходимая синхронизация обеспечивается процессором аппаратно в реализации инструкции CAS. После того, как ячейка сброшена в 0, если это действие выполнил именно текущий поток, происходит вызов деструктора. Ввиду того, что в деструкторе могут быть вызваны блокирующие операции, декремент счетчика (а следовательно и синхронный вызов деструктора) происходят в контексте с разблокированным планировщиком.

7.7.5 Квоты

Рассмотренные выше механизмы обеспечивают корректное использование объектов ядра, однако надо понимать, что любые разделяемые ресурсы могут становиться потенциальными целями для атак, направленных на исчерпание ресурсов. Хотя память для каждого процесса распределена статически и во время работы никакой процесс не может изменить это распределение, память под объекты ядра выделяется из единого пула, который разделяется между всеми процессами, поэтому, если не защищаться от такого сценария, возможна атака такого типа, что один процесс может выделить большое количество объектов ядра, например семафоров, исчерпав при этом всю доступную для них память. Другой процесс, попытавшись создать семафор, потерпит неудачу и тем самым будет нарушена возможность работы этого процесса, что идет вразрез с начальными требованиями, предполагающими изоляцию и невозможность одних процессов нарушить работу других. Те же самые рассуждения применимы и для приоритетов потоков: процесс может увеличить приоритет своим потокам до максимального уровня и помешать работе других процессов в их конкуренции за разделяемый ресурс - процессорное время.

Все это приводят к идее о том, что обеспечить корректное использование ресурсов, это только одна из задач, стоящих перед разработчиком ОСРВ, поддерживающей изоляцию процессов. Другая не менее важная задача - предотвратить неограниченное поглощение любых ресурсов одним из процессов, то есть предусмотреть механизм квотирования. Ответственный процесс всегда должен быть уверен, что необходимый, выделенный для него, набор ресурсов всегда доступен и прочие процессы не смогут вызвать дефицит этих ресурсов. Для решения этой проблемы, с каждым процессом сопоставляется определенное количество ресурсов (квота), необходимых ему для работы. Эти ограничения устанавливаются разработчиком системы в момент ее конфигурирования, хотя могут быть и варианты установки квот во время работы. Процесс не может выделить количество объектов большее, чем было для него предусмотрено, поэтому атаки такого типа как описано выше становятся невозможны.

Ограничение приоритета потоков выполняется довольно просто - соответствующая проверка вставляется в функции создания потока, с ограничением же количества объектов связано несколько тонкостей.

Квоты могут быть реализованы различными способами, например, с помощью таблиц идентификаторов. Ограничив количество идентификаторов для объектов заданного типа, можно запретить процессу создавать объектов больше, чем идентификаторов. Если квота исчерпана, создание объекта завершится неудачей при попытке вставить в таблицу новый идентификатор. Если же все сработало без ошибок, можно быть уверенным, что у

процесса есть возможность создать объект данного типа.

Такой дизайн используется во многих ОС и вполне приемлем для систем общего назначения. К сожалению, в системах повышенной надежности такой подход не дает 100% гарантии выполнения квоты. Вот как это может произойти: проблема возникает в момент освобождения идентификатора. Если не рассматривать систему, в которой существуют глобальные блокировки, позволяющие сделать освобождение идентификатора и памяти объекта атомарно, то эти два действия можно рассматривать как независимые. При этом возможны две ситуации, вначале освобождение памяти, потом идентификатора, и наоборот. Первый вариант, очевидно, некорректен, так как существует время, в течении которого доступный пользователю идентификатор ссылается на некорректный объект ядра.

Альтернативный вариант выглядит лучше: вначале происходит освобождение идентификатора и лишение пользователя возможности обращаться к данному объекту, затем происходит освобождение памяти в результате декремента счетчика ссылок. Нетрудно увидеть, что существует момент, когда идентификатор уже освобожден, то есть считается, что процесс может снова выделить такой объект, без превышения квоты, но еще не освобождена память. Поэтому, если в этот момент, какой-то другой поток этого процесса попытается выделить снова такой объект, проверка квоты будет пройдена и из пула будет выделен еще один объект, то есть процесс будет кратковременно владеть двумя объектами, хотя идентификатор был выделен только для одного. Если количество объектов в пуле исчерпано, другой процесс с большей квотой может в этот момент потерпеть неудачу при создании объекта. При неблагоприятном стечении обстоятельств, особенно в многопроцессорной системе, возможно превышение квоты и более чем на 1, хотя вероятность такого события очень низка, но тем не менее отлична от нуля.

Существует несколько способов решения этой проблемы. Например, можно ввести дополнительное API в таблицу идентификаторов, и сделать удаление объекта трехстадийным - закрытие идентификатора без изменения квоты, освобождение памяти, увеличение квоты. Альтернативный подход заключается в выделении процессу приватных пулов памяти. Каждый процесс обладает набором пулов, которыми он может пользоваться, выделение памяти происходит только из этих пулов и возвращается в них же. Квота, таким образом, указывается в качестве размера пула. Очевидно, что в этом случае превышения квоты не может произойти ни при каких условиях, другие процессы просто не разделяют ресурсы памяти, поэтому, если процессу выделен пул, он всегда сможет выделить ровно столько объектов, сколько находится в этом пуле.

7.7.6 Проверка указателей

После рассмотрения вопроса о проверке корректности объектов ядра, следующий по важности вопрос - проверка структур данных, получаемых из пользовательского режима по ссылке, то есть по указателю. В отличие от пользовательского процесса, ядро имеет доступ к памяти других процессов, поэтому вокруг использования указателей может быть построена эксплуатация уязвимостей, если ядро не проверяет их должным образом.

Следует учитывать также тот факт, что пользователь может модифицировать значения по указателю в то время, когда они используются ядром ОС. Атаки такого типа называются в

некоторых источниках [3] атаками TOCTOU (Time of check, time of use). Для их предотвращения, как правило, содержимое пользовательских данных копируется на стек ядра перед использованием, поэтому проверка и использование выполняются для значений, которые уже не могут быть модифицированы из непривилегированного режима.

Что касается проверки корректности самого указателя, здесь существует две основные техники. В ядрах статического типа, распределение памяти которых не может меняться в процессе работы системы, проверку указателя можно организовать с помощью описанных ранее интерфейсов HAL. В случае MMU, это может быть вызов HAL, для проверки того, что пользователь действительно имеет доступ к той памяти, указатель на которую он передает в системный вызов. Довольно редко передаются объемы в мегабайты и гигабайты, поэтому накладных расходов немного и в большинстве случаев они сводятся всего к одной проверке, так как данные помещаются в одну страницу. Системный вызов при этом выглядит следующим образом:

```
register_t some_syscall(
  register_t p0, register_t p1, register_t p2, register_t p3, register_t p4) {
  void* user_ptr = (void*) p0;
  struct user_data_buf some_buffer;

  if(mmu_translate(..., user_ptr, sizeof(struct user_data_buf))) {
    memcpy(some_buffer, sizeof(struct user_data_buf));

    ...
  }

  return ...
}
```

В том случае, когда данных может передаваться много, а также в случае, если распределение адресного пространства процесса может меняться во время работы, применяется другой метод. Системный вызов пытается копировать данные без проверок, допустимость этого определяется процессором аппаратно, в зависимости от текущего загруженного каталога страниц. Разумеется, при передаче некорректного указателя, а также в том случае, если структура адресного пространства была изменена другими потоками данного процесса, возникнет исключение защиты.

```
register_t some_syscall(
  register_t p0, register_t p1, register_t p2, register_t p3, register_t p4)
{
  void* user_ptr = (void*) p0;
  struct user_data_buf some_buffer;

  copyin(some_buffer, user_ptr, sizeof(struct user_data_buf));
  ...
  return ...
}
```

Это исключение обрабатывается специальным образом так, что если оно возникло, это вызывает возврат из функций копирования ошибки. Реализация такого механизма

довольно сложна, так как требует участия ассемблера. То есть требуется поддержка такого интерфейса со стороны HAL. В случае x86, например, эта функция может выглядеть так:

```
copyin:
       /* Проверки, что адрес и размер находятся в допустимом диапазоне. */
       /* Копирование... */
known_fault: rep movsb
       /* Установка статуса успешного завершения */
       xorl %eax, %eax
       ret
```

И есть альтернативное продолжение этой функции:

```
copyin_fault:
        movl <ERROR>, %eax
       ret
```

Обработчик страничного исключения проверяет, не является ли адрес, по которому произошло исключение, одним из известных адресов, находящихся внутри функций типа copyin. Если адрес известный, происходит модификация адреса возврата в стеке на copyin_fault. То есть, если при копировании произошло исключение в функции copyin, то после исключения возврат из него произойдет в альтернативное продолжение функции, выполнение которого приведет к возврату кода ошибки из функции.
Подобным образом реализуется и копирование информации из режима ядра в режим пользователя по указателю, функция в этом случае называется copyout.

7.7.7 Завершение потока во время выполнения системного вызова

На первый взгляд, описанных выше мер достаточно для обеспечения полной изоляции процессов: контролируется точки вызова системы, аргументы всех функций проверяются таким образом, что передача недействительных объектов исключается, выполнение происходит на выделенном стеке, а ресурсы ограничены пулами, приватными для каждого процесса. Однако существует еще одна неочевидная проблема, которая может препятствовать корректному использованию ресурсов. Эта проблема связана с завершением потока во время выполнения системного вызова.
Поскольку системный вызов ничем не отличается от обычного выполнения потока (за исключением работы в привилегированном режиме), он, так же как и обычный код потока, может быть завершен в любой момент и в любой точке своего выполнения. Например, если возникает ошибка в одном из потоков процесса и процесс должен завершиться, все его потоки принудительно останавливаются даже в том случае, если в этих потоках ошибок не возникало. Проверка аргументов с использованием идентификаторов предполагает, что любой системный вызов должен доработать до конца и правильно использовать функции object_ref/object_deref. К тому же, внутри системного вызова потенциально могут использовать общесистемные мьютексы и прочие примитивы синхронизации. Если же поток может быть завершен посреди выполнения системного вызова, он, очевидно, не сможет корректно его обработать и выполнить

необходимые действия, такие как освобождение идентификатора, уменьшение счетчика ссылок на объект, освобождение мьютекса и так далее.

Можно искать различные выходы из этой ситуации, например, каждый раз, в случае запуска или перезапуска процесса, инициализировать его заново и обновлять указатели на буферы из какого-то сохраненного заранее места. Но в целом крайне сложно писать алгоритмы, которые должны оставаться корректными с точки зрения использования ресурсов, притом что алгоритм может быть остановлен в произвольном месте. Можно решить проблемы с памятью, но она возникнет с выделением идентификаторов или какими-либо другими побочными эффектами. Воевать с проблемой таким образом, это все равно что бороться с многоголовой гидрой: отрубив одну голову, будут вырастать две новые. Очевидно, что корень зла находится в самом утверждении, что поток может быть в произвольном месте завершен. Поэтому рабочее решение заключается в том, чтобы гарантировать потоку возможность доработать до того момента, когда все ресурсы будут освобождены. Этот момент известен только самому обработчику системного вызова, поэтому нужно использовать дополнительное API со смыслом "запретить уничтожение потока" и разрешить.

Для реализации такого функционала не потребуется изменений в наноядре, весь необходимый для этого функционал уже в нем содержится. Достаточно вспомнить, как происходит асинхронное удаление потока - с помощью APC. При этом в системе уже есть API, позволяющее маскировать APC. Поэтому системный вызов должен маскировать APC для того, чтобы предотвратить собственную асинхронную остановку. Несмотря на всю мощь этого инструмента, позволяющего решить проблему корректного использования ресурсов и алгоритмов, использовать этот механизм следует осторожно. В то время, когда поток находится в критическом регионе, он не может быть завершен, а следовательно, процесс не может быть перезапущен, что может быть критично. Хорошая новость заключается в том, что в отличие от регионов с повышенным SPL, потоки остаются вытесняемыми даже в то время, когда находятся в критическом регионе. Но следует учитывать, что длина региона влияет на латентность перезапуска процесса в случае ошибки, точно так же, как время запрета прерываний влияет на время реакции. Системные вызовы, использующие объекты, при этом приобретают следующий вид.

```
/* Запрет APC и асинхронного завершения потока. */
enter_critical_region();

/* Проверка корректности идентификатора и получение ссылки на объект. */
void* object = handle_ref_object(&handle_db, type, handle);

if(object != NULL) {
  /* Использование объекта. */
  error = ...
  object_deref(object);
}
leave_critical_region();
```

Запрет планирования (повышение SPL до уровней >= DISPATCH) также предотвращает асинхронное завершение потока по причине блокирования доставки APC, но, в отличие от

описанного метода, у этого способа есть множество других нежелательных ограничений. Вместе с невозможностью завершения потока, блокируется также планирование, что накладывает ограничения на время выполнения кода, страдает также время реакции на прерывания. Но главное ограничение связано с доступным API: невозможность блокирования и ожидания примитивов синхронизации. Напротив, критический регион на основе запрета APC сохраняет поток вытесняемым и блокируемым, то есть доступен весь набор методов синхронизации и прочего, поэтому, при прочих равных, использование в системных вызовах критических регионов на основе APC предпочтительно.
В разных ОС для обеспечения этих гарантий используются различные механизмы, вплоть до запрета остановки или принудительного завершения потоков, которые находятся в режиме ядра, однако это может повлечь за собой дополнительные накладные расходы, так как не всякий вызов должен быть защищен таким образом.

7.7.8 Блокирующие вызовы из режима пользователя

Обрамление системного вызова функциями запрета APC и создание критического региона на все время выполнения, молчаливо предполагают, что системный вызов неблокирующий. Большинство вызовов действительно являются таковыми и с ними никаких проблем нет. С блокирующими вызовами ситуация обстоит иначе. Блокирующий системный вызов как раз и предполагает, что поток блокируется на определеяемое пользователем время (пока он не активирует примитив или иным образом не прервет ожидание). Избежать создания критического региона при этом нельзя, потому что сам примитив синхронизации, который используется для ожидания, должен быть предварительно проверен, и, как следствие этого, на него будет создана ссылка, которую нужно освободить после завершения ожидания. Соответственно, оба возможных варианта решения проблемы имеют связанные проблемы и поэтому не подходят для решения: если выполнять все в критическом регионе - пользователь сможет создать незавершаемый процесс, начав ждать примитив и не активируя его намеренно; если выполнять ожидание вне критического региона, возможна утечка памяти, после завершения потока без декремента счетчика ссылок.
Для решения этой проблемы, в блокирующих вызовах используется дополнительный механизм, основанный на обсуждавшихся ранее cleanup-обработчиках. Первоначально поток входит в критические регион и выполняет необходимую проверку аргументов. Далее, в этом же регионе, инициализируется cleanup-обработчик, в который передается информация о том, какие объекты он должен освободить. Затем поток выходит из критического региона и приступает к ожиданию. Поскольку счетчик ссылок объекта ранее был увеличен, объект является гарантированно корректным. Завершение же потока на любом этапе ожидания вызовет обработчик, который уменьшит счетчик ссылок и удалит объект. Поскольку объект обработчика располагается на стеке, он также остается корректным все время существования потока. После завершения ожидания, поток вновь входит в критическую область. Если выполнение потока дошло до этого момента, значит, очевидно, завершения потока не произошло в той точке где APC были разрешены, а следовательно, обработчик не был вызван. Обработчик удаляется из потока и происходит декремент счетчика ссылок обычным образом, затем поток выходит из критического

региона и покидает системный вызов.

```
/* cleanup-обработчик умеьшающий счетчик ссылок для заданного объекта и типа. */
static void sem_cleanup(..., void* arg) {
  ... object = (...) arg;
  object_deref(object);
}

/* Блокирующий системный вызов. */
register_t syscall_sem_timedwait(
 register_t p0, register_t p1, register_t p2, register_t p3, register_t p4) {
  ...
  /* Вход в критическую область и проверка аргументов... */
  enter_critical_region();
  ... object = handle_ref_object(...);

  if(object != NULL) {
    /* Инициализация и вставка cleanup-обработчика в стек обработчиков потока. */
    cleanup_handler_t cleanup;
    cleanup_init(&cleanup, sem_cleanup, object);
    cleanup_add(..., &cleanup);
    leave_critical_region();

    error = sem_timedwait(&object->internal_sem, timeout);

    /* Повторный вход в критическую область. */
    enter_critical_region();
    cleanup_cancel(&cleanup);
    object_deref(object);
  }
  leave_critical_region();
  return error;
}
```

Если поток был асинхронно завершен в момент ожидания (предполагается, что синхронных завершений потока не будет), это приведет к выполнению cleanup-обработчика, который уменьшит счетчик ссылок.

Ожидание одного примитива это классический случай, но похожий подход может использоваться и в более сложных системных вызовах, требующих множества ожиданий или использования нескольких объектов. Возможность устанавливать произвольное продолжение потока, выполняющееся в случае его завершения, позволяет гарантировать корректную работу с ресурсами даже в том случае, если потоки будут завершаться в произвольных местах.

7.7.9 Динамическое создание процессов

Рассмотренные выше методы реализации статических процессов позволяют реализовать и их динамический вариант. То есть запуск и завершение процессов во время работы системы. Этого можно добиться путем добавления системных вызовов, которые динамически выделяют и инициализируют объект процесса и его первого потока. Некоторую трудность здесь может представлять только поддержка систем с MPU. Поскольку последний предполагает использование единого адресного пространства для всех процессов, это приводит к тому, что необходимо либо выполнять связывание

символов в бинарном образе процесса после его запуска, либо использовать MMU для того, чтобы с точки зрения процесса адресное пространство всегда выглядело одинаково.

7.8 Оптимизация производительности

7.8.1 Обработка прерываний в пользовательском режиме

В некоторых случаях бывает необходимо обеспечить обработку аппаратных прерываний в непривилегированном режиме. Целью может быть минимизация кода, работающего в привилегированном режиме, изоляция кода драйвера и тому подобное. При этом нужно понимать, что в случае потенциальной злонамеренности драйвера устройства, даже работа в непривилегированном режиме не может помешать ему вызвать крах ядра. Многие устройства предполагают использование DMA, поэтому программируя устройство определенным образом, можно обходить защиту памяти. С другой стороны, если драйвер не пытается специально обойти защиту, то перенос его в непривилегированный режим может помочь выявить в нем ошибки или изолировать их от ядра и других процессов.

Так как "общение" с большинством устройств в настоящее время происходит с помощью регистров отображенных на память, то открыть к ним доступ определенном процессу несложно - требуется только настроить должным образом таблицы страниц или MPU. Главная трудность заключается в прерываниях: многие процессоры предполагают выполнение обработчиков прерываний только в привилегированном режиме, к тому же, обработчики выполняются на едином стеке. В случае, если требуется поддержка разделяемых между устройствами векторов прерываний, то иного выхода нет - в таком случае ядро должно предусмотреть возможности для вызова обработчиков в режиме ядра. В большинстве же случаев, современные процессоры поддерживают достаточное количество векторов и в их разделении нет необходимости. Используется в основном следующий подход: ядро предоставляет возможность ассоциировать с прерыванием некоторый примитив синхронизации, который в дальнейшем может использоваться для ожидания прерываний и их обработки в пользователькомрежиме. Например, подобным образом устроены ОС Windows CE и QNX. В первом используется функция InterruptInitialize, которая соединяет вектор прерывания и объект "событие". Во втором случае используется механизм, похожий на сигналы: поток присоединяет к вектору прерывания себя, и может пользоваться функцией InterruptWait, которая используется для блокирования потока до прихода прерывания, то есть, даже сам примитив синхронизации и нижележащие механизмы его ожидания скрыты от непривилегированного кода.

В отличие от случая с ISR, где маскирование вектора осуществляется во многих случаях аппаратно, в случае переноса всей обработки в потоки, потенциально возникает проблема с приходом новых прерываний в то время, когда еще не обработаны предыдущие. Для ее решения ядро обычно маскирует вектор, а размаскировка должна выполняться уже из пользовательского потока, для чего предоставляется специальное API.

Что касается реализации то она довольно очевидна: системный вызов предполагающий присоединение примитива синхронизации к вектору выделяет объект, содержащий

объекты ISR и DPC, а также хранящий ссылку на объект, переданный пользователем. При возникновении прерывания, вектор маскируется, после чего активируется указанный примитив.

7.8.2 Обработка исключений в пользовательском режиме

Исключения так же, по аналогии с прерываниями, могут обрабатываться в пользовательском режиме. Это может быть полезно тогда, когда приложение может обрабатывать свои ошибки и продолжать работу, так что нет необходимости выполнять его остановку и рестарт.

В связи с тем, что исключения это синхронные события, использовать подход, применяемый для прерываний не получится. Механизм обработки исключений обычно работает так: после возникновения исключения, его обработка всегда начинается в ядре. Обработчик может определить, что исключение возникло в непривилегированном режиме (определить это он может по сохраненному фрейму прерывания, к которому обработчик исключения как правило имеет доступ). Поскольку в этом же фрейме содержится информация о пользовательском стеке, в случае, если исключение возникло в этом режиме, это дает возможность обработчику модифицировать стек таким образом, чтобы после возврата из первичного обработчика исключения управление передавалось в пользовательский обработчик, который должен быть предварительно установлен с помощью системных вызовов, либо указатель на него должен располагаться в определенном месте в адресном пространстве процесса. В целом механизм похож на механизм доставки прерываний в случае использования виртуальных машин. В POSIX для этого используются сигналы, которые с помощью системных вызовов сообщают ядру адреса обработчиков. В дальнейшем ядро выполняет вставку этого обработчика в поток в случае возникновения синхронных исключений.

7.8.3 Реализация примитивов синхронизации в пользовательском режиме

Существуют способы реализовать примитивы синхронизации таким образом, чтобы сочетать скорость их реализации в библиотеке поддержки с удобством использования примитивов и потоков на уровне ядра ОС.

Один из примеров такого подхода: мьютекс. Его реализация в пользовательском режиме предполагает существование двух объектов: очереди ожидания, которая может использоваться только в ядре, и переменной, которая хранит текущего владельца мьютекса, доступной из непривилегированного режима. То есть речь идет об разделении целостного объекта, такого как fx_mutex_t на две части. Используется также два специфических системных вызова WAIT и WAKE.

Когда поток пытается захватывать мьютекс, он, используя атомарные операции вроде CAS, работает с переменной в пространстве пользователя.

Переменная в этом случае должна иметь как минимум 3 условных значения:

- Мьютекс свободен
- Мьютекс занят и очередь его ожидания пуста (нет конкуренции за мьютекс)
- Мьютекс занят и есть ожидающие его потоки

До тех пор, пока с объектом работает только один поток, все происходит очень быстро и

целиком в пользовательском режиме. Состояние мьютекса меняется между первыми двумя значениями.
При попытке захватить мьютекс когда он уже захвачен, поток пытается (с помощью CAS-loop) перевести его в третье состояние. Если это удается - происходит системный вызов WAIT, который получает в качестве аргумента адрес переменной.
Псевдокод системного вызова можно представить следующим образом:

```
wait(volatile int* addr) {
	wait_block wb;
	int status = skip;
	spinlock_acquire(...);
	int value = *addr;
	if (val == <mutex_busy_contested>) {
		enqueue(&wb, hash(addr));
		status = wait;
	}
	spinlock_release(...);
	if (wait) {
		...
	}
	...
}
```

После перехода в режим ядра, поток находит очередь, соответствующую данной переменной. Затем, уже под спинлоком очереди, переменная проверяется еще раз и если она не изменилась - поток помещается в очередь и приостанавливается как и при обычной реализации мьютекса. Поскольку в ядре текущий поток оказался, предварительно обозначив это, поток-владелец, когда он решит освободить мьютекс, также будет вынужден зайти в ядро и проанализировать очередь. Поскольку ядро ничего не знает о пользовательских мьютексах, возникает вопрос, откуда берется эта очередь и ее элементы? Здесь существует два подхода. В первом из них элементы в виде блоков ожидания создаются на стеке ожидающих потоков, поскольку не завершив ожидание, поток не может начать другое, каждый поток нуждается не более чем в одном блоке ожидания. Очереди создаются на этапе инициализации системы и определение, в какую их них попадет мьютекс, определяется обычно с помощью хэширования. Спинлоки в данной архитектуре связаны не с объектом ядра, как в случае реализации мьютексов в ядре, а с этой очередью. Другой подход предполагает явное создание объекта ядра типа "фьютекс" который имеет идентификатор и выступает в качестве очереди ожидания.
Теперь рассмотрим реализацию операции WAKE. Она вызывается всякий раз, когда владелец мьютекса решает его освободить и видит, что у мьютекса есть ожидающие потоки после обновления значения переменной в пользовательском режиме.

```
wake(volatile int* addr) {
	spinlock_acquire(...);
	wait_block* wb = dequeue(hash(addr));
	spinlock_release(...);
	if (wb) {
		...
	}
```

}

По адресу мьютекса вычисляется некоторый хэш, после чего, под спинлоком соответствующей очереди происходит попытка найти блок ожидания, вставленный ожидающим потоком.

При этом может оказаться, что ожидающий поток еще не успел вставить блок ожидания в очередь. Такое происходит, если освобождение мьютекса произошло между обновлением состояние переменной и вызовом WAIT на стороне ожидающего потока. В этом случае WAIT закончится без ожидания увидев изменившееся состояние переменной. Если же блок ожидания вставлен в очередь, происходит его извлечение и пробуждение соответствующего потока.

Можно сделать обобщение этого интерфейса: вместе с адресом в функцию WAIT можно передавать прочитанное значение, чтобы после чтения в ядре иметь возможность определить, что значение изменилось. В таком случае нет коду ядра нет необходимости знать о конкретных значениях, и функция получается более абстрактной, пригодноя для реализации не только мьютексов. Подобным образом можно реализовать большинство примтивов, включая семафоры и условные переменные [8].

Несмотря на очевидные достоинства, связанные с производительностью, некоторую сложность вносит необходимость использования хэш-таблиц и потенциальные непредсказуемые заранее задержки. Худший случай будет состоять в просмотре под спинлоком всех мьютексов созданных данным процессом (либо всех мьютексов в системе, если мьютексы могут разделяться между процессами), поэтому масштабируется такая архитектура хуже, по сравнению с реализацией мьютексов в ядре и сопоставлением спинлоков только с объектами. В связи с этим в системах реального времени предпочтительно использовать подход, где фьютексы имеют соответствующий объект ядра, что хоть и связано с дополнительными затратами на создание этого объекта, ограничивает сверху время захвата спинлока и запрета прерываний.

7.8.4 Расширение интерфейса ядра

Описанные сервисы, такие как потоки и примитивы синхронизации, создают впечатление, что процессы это некоторые вещи в себе, полностью изолированные от внешнего мира. На самом деле, если процесс выполняет какую-то полезную функцию, он должен каким-то образом общаться с внешним миром, получать данные из него и отправлять наружу. Поэтому, в дополнение к базовым сервисам процессов потоков и синхронизации, ядро ОС должно предоставлять еще некоторые дополнительные интерфейсы, которые могут использоваться для обмена информацией.

В микроядерных ОС для этих целей обычно используются механизмы межпроцессного взаимодействия, которые реализуются ядром. Эти механизмы вкратце будут рассмотрены в следующей главе. Применение "микроядерного" подхода привносит некоторые дополнительные накладные расходы, которые могут быть не всегда приемлемы, особенно для устройств с ограниченными ресурсами либо с особыми требованиями к производительности. В распространенных ОС, как встроенных так и общего назначения, обычно предполагается, что помимо процессов потоков и синхронизации, ядро

предоставляет ряд дополнительных сервисов, в состав которых входят файловая система, поддержка драйверов устройств, стек сетевых протоколов и тому подобное.

В контексте рассматриваемой архитектуры ядра это может быть сделано дополнительными компонентами, которые устанавливают свои обработчики в таблицу системных вызовов (Рис. 91):

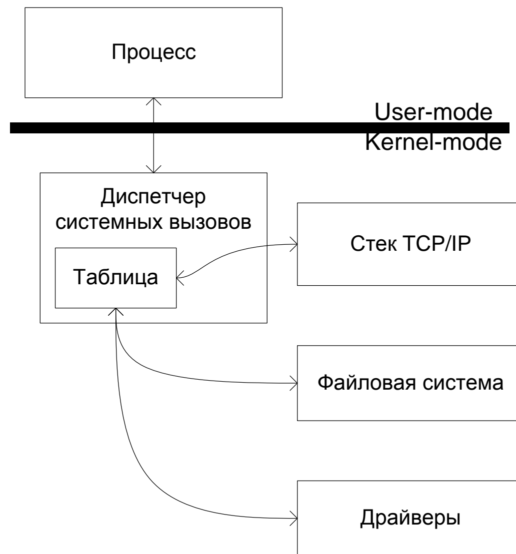

Рис. 91 Расширение ядра компонентами исполнительной подсистемы.

При необходимости реализовать микроядерную архитектуру, в исполнительную подсистему добавляются только механизмы IPC, с помощью которых процессы могут общаться между собой, так что ряд компонентов могут быть реализованы в пользовательском режиме в виде процессов (Рис. 92).

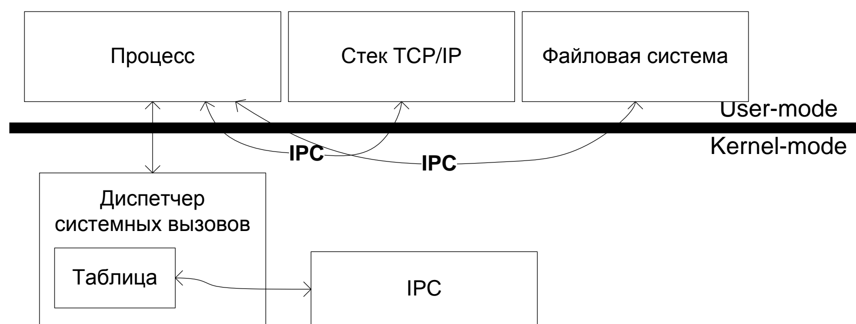

Рис. 92 Расширение интерфейса ядра с помощью IPC.

Существует еще более простой вариант: дать возможность разработчику привилегированной части приложения определять свои собственные системные вызовы, через которые процессы могли бы взаимодействовать как с устройствами так и с привилегированным приложениям. В FX-RTOS такой интерфейс называется PFC (privileged function call). Этот интерфейс состоит из единственной функции, которая является просто оберткой вокруг соответствующего пользовательского вызова. Все аргументы передаются без изменений и их проверка возлагается на разработчика PFC-вызова:

```
register_t syscall_fx_pfc_call(
  register_t p0, register_t p1, register_t p2, register_t p3, register_t p4) {
  return fx_pfc_handler(p0, p1, p2, p3, p4);
}
```

Можно сказать, что PFC это компонент, который определяет специальный системный вызов и берет на себя некоторые инфраструктурные вопросы, транслируя обращения к системному вызову к вызову внешней функции, которую должно предоставить привилегированное приложение.

7.9 Ввод-вывод

Важным вопросом при разработке непривилегированных приложений является способ их общения с ОС и возможно с другими процессами. Приложения практически никогда не являются вещами в себе, им нужно взаимодействие с устройствами. В простейших случаях может применяться интерфейс вроде специальных системных вызовов PFC, реализуемых для конкретного приложения. Эти системные вызовы можно нагружать произвольной функциональностью и скрывать за ними, в том числе, драйверы устройств. То есть привилегированная часть приложения содержит обработчики системных вызовов, которые вызываются по запросу непривилегированного приложения (Рис. 93)

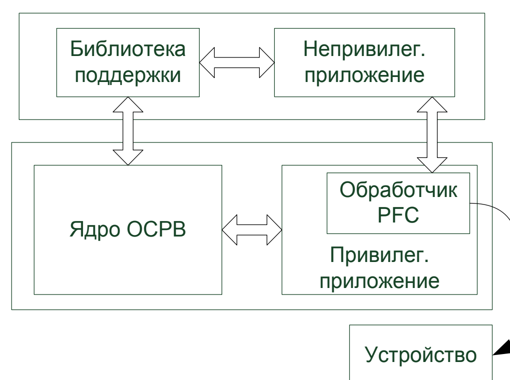

Рис. 93 Ввод-вывод через интерфейс PFC.

При этом, однако, надо учитывать возможные некорректные аргументы и прочее, проектировать привилегированую часть так, чтобы через нее нельзя было скомпрометировать всю систему. В этом случае все интерфейсы полностью определяются приложением, никакая их часть не является стандартной. Из плюсов такой организации ввода-вывода только исключительная простота. Если приложению требуется только доступ к простым функциям вроде мигания светодиодами, то довольно тяжеловесно использовать для этого целую подсистему дрейверов.

Интерфейс может быть расширен до поддержки универсальнго интерфейса драйверов уровня ядра с поддержкой стандартного API, такого как POSIX. Это ведет к усложнению кода ядра, что негативно скзывается на его надежности, но зато относительно просто и обеспечивает хорошую производительность. В этом случае ядро содержит некоторый стандартный интерфейс для обращения к драйверам, которые подключаются в виде отдельного модуля (Рис. 94). Такая организация ввода вывода более тяжеловесна, но зато обеспечивает стандартные интерфейсы, драйверы отныне могут разрабатываться и

подключаться независимо от приложения и ОС.
По этому пути в настоящее время идет большинство ОС, в том числе настольных. Драйверы в режиме ядра могут вызываться в контексте вызывающего процесса, что происходит очень быстро.

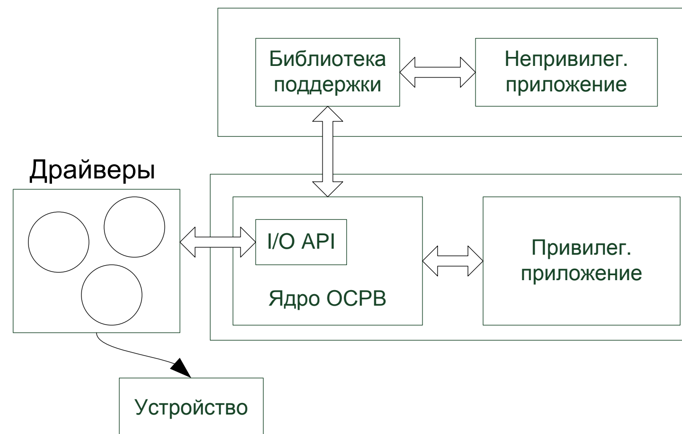

Рис. 94 Драйверы в режиме ядра.

Можно разрешить пользовательскому процессу обращаться к какому-либо устройству напрямую, добавив в его адресное пространство разрешенный регион, где расположены регистры устройства. Разумеется, не всякое устройство можно сделать доступным для пользовательских процессов. Скажем, "открыв" таким образом сетевой адаптер можно дать возможность писать в любую физическую память, что недопустимо. Но для простых устройств вроде светодиодов, которые не содержат даже прерываний, вполне допустимо управлять такими устройствами полностью из пользовательской программы.

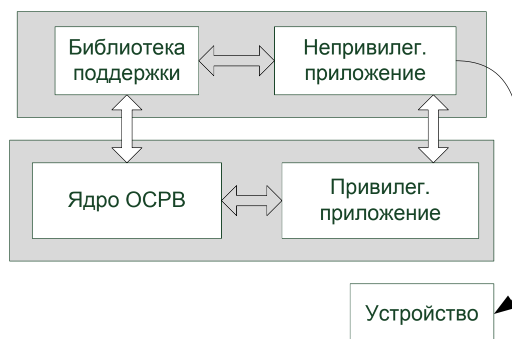

Рис. 95 Прямое управление устройством из пользовательской программы.

Этот метод хорош производительностью, но, очевидно, плохо портируем, подходит не для всякого устройства и требует реализации слоев абстракции внутри приложения, если требуется достичь совместимости и возможности работы с разными устройствами.
Наконец, третий путь, следует прямо из концепции микроядра. Драйверы устройств группируются в процессы, общение с которыми происходит через механимы межпроцессоного взаимодействия (interprocess communication, IPC) (Рис. 96). В этом случае ядро не страдает от возможных ошибок и сложности наличия в нем драйверов, пользовательские приложения не могут напрямую управлять устройствами, все разделено по уровням ответственности. Вызовы IPC могут быть скрыты за стандартными интерфейсами обращения к драйверам внутри библиотеки поддержки, поэтому

приложения могут использовать их так же, как и драйверы режима ядра.

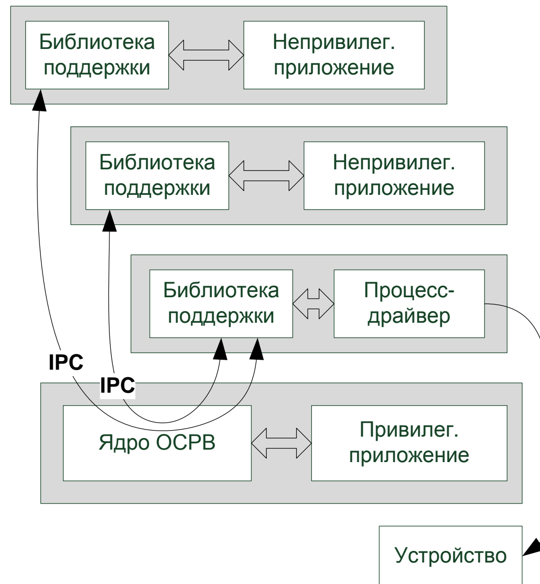

Рис. 96 Реализация драйверов в виде процессов.

К сожалению, этот метод, очевидно, страдает от накладных расходов связанных с IPC и для систем с ограниченными ресурсами применяется редко, хотя и обеспечивает максимальный уровень надежности.

Одним словом, каким образом устроены драйверы, определяется типом управляемого устройства и требованиями к производительности и надежности.

7.10 Пример реализации процессов: FX-RTOS

После рассмотрения основных составляющих для создания изолированных процессов, можно приступить к реализации.

Исполнительная система, так же как и наноядро, имеет свои внутренние представления о процессах и потоках. В FXRTOS, принята следующая политика именования, функции и типы данных, которые относятся к ядру имеют префикс fx_, относящееся к исполнительной подсистеме имеет префикс ex_. Объект, представляющий непривилегированный процесс, должен содержать следующие основные компоненты: адресное пространство, в котором будут выполняться его потоки, таблицу идентификаторов объектов, используемую для проверки аргументов, один или несколько пулов для объектов, которые процесс может создавать, список потоков, принадлежащих процессу, а также некоторые вспомогательные поля, такие как максимальный приоритет потока, который может работать внутри процесса и т.д.

```
typedef struct _ex_process_t {
  ex_object_header_t header;              /* Заголовок объекта. */
  fx_process_t process;                   /* Адресное пространство. */
  ex_handle_db_t handle_db;               /* Таблица идентификаторов объектов. */
  ex_obj_pool_t* pool;                    /* Память для локальных объектов. */
  unsigned int thread_prio_max;           /* Максимальный приоритет для потоков. */
  rtl_list_t threads;                     /* Список потоков. */
  volatile unsigned int thread_num;       /* Количество потоков в списке. */
  lock_t lock;                            /* Спинлок для защиты списка потоков. */
  ...
}
```

```
ex_process_t;
```

Потоки представлены объектами со следующей структурой:

```
typedef struct _ex_thread_t {
  ex_object_header_t header;                          /* Заголовок объекта. */
  fx_thread_t kthread;                                /* Объект "поток" наноядра. */
  int kstack[EX_THREAD_KER_STK_SZ /sizeof(int)];      /* Стек ядра потока. */
  ...
}
ex_thread_t;
```

Поскольку память, используемая для объектов потоков, должна управляться через механизм подсчета ссылок, первым элементом структуры является стандартный заголовок объекта. Далее расположен низкоуровневый объект ядра "поток", а также стек для него. В отличие от случая, когда в потоках ядра работает прикладной код, на стеке ядра потока работают только системные вызовы, количество которых ограничено, поэтому размер стека не зависит от сложности кода потока и может быть фиксирован для данной конфигурации ядра ОС.

Подобным образом определяются и примитивы синхронизации: объект ядра инкапсулируется в расширенную структуру, содержащую заголовок для подсчета ссылок и дополнительные поля, используемые исполнительной подсистемой.

Методы, связанные с запуском процессов не очень сложны.

```
int ex_process_init(ex_process_t* p);
ex_process_t* ex_process_self(void);

int ex_process_start(
  ex_process_t* p,void(*f)(void*),void* arg, unsigned int prio, void* stk, size_t sz);

void ex_process_kill(ex_process_t* p);
```

Первые две функции тривиальны: конструктор, инициализирующий перечисленные поля внутри объекта процесса, и функция получения указателя на текущий модуль. Работает она путем получения текущего потока, получения указателя на связанное с ним адресное пространство и получения по нему указателя на процесс. Важно понимать, что эта функция может использоваться только в системных вызовах, когда очевидно, что вызывающий поток действительно принадлежит какому-то модулю, результат вызова из потока ядра для этой функции не определен.

Функция запуска процесса используется для создания первого потока, который будет выполняться внутри указанного процесса. Она включает в себя следующие основные шаги:

- Выделение нового потока из пула потоков указанного процесса
- Инициализация потока и создание его контекста
- Вставка идентификатора потока в таблицу

После завершения этой функции созданный поток начинает выполнение в пользовательском режиме, обращаясь к ядру через интерфейс системных вызовов. Передаваемый в функцию стек должен быть доступен для доступа из пользовательского

режима.
В случае возникновения исключений или в результате явного запроса, процесс может быть остановлен. Остановка процесса предполагает принудительное завершение всех его потоков, а также освобождение всех ресурсов, которое происходит как результат завершения потоков. Функция остановки устроена довольно просто: в цикле, который перебирает все потоки, принадлежащие указанному процессу, вызывается функция fx_thread_terminate для всех потоков, после чего закрывается таблица идентификаторов процесса, уменьшая счетчик ссылок на все используемые объекты. Так как, после получения APC для принудительного завершения, у потока нет никакой возможности предотвратить его выполнение, рано или поздно все потоки будут завершены.

7.10.1 Запуск и завершение процесса

С точки зрения приложения, которое использует процессы, все выглядит также несложно. Первоначально управление получает функция инициализации приложения, которая инициализирует привилегированное приложение, создает потоки, примитивы синхронизации и все прочее. Написание этой функции ничем не отличается от описанного ранее в соответствующей главе. Далее эта функция может запустить один или более пользовательских процессов, а также потоков ядра, если они используются. Так как рассматривается модель со статическими процессами, предполагается что адресное пространство распределено на этапе компиляции, а у ядра есть некая конфигурационная информация, как именно следует создавать таблицы страниц. Создание процесса выглядит следующим образом:

```
void fx_app_init(void)
{
  static ex_obj_pool_t mempool;
  static ex_process_t my_process;

  ex_obj_pool_init(&mempool, ...);
  ex_process_init(&my_process);

  my_process.pool = &mempool;
  ...
  /* Настройка адресного пространства. */
  ...
  ex_process_start(&my_process, ...);
}
```

Поскольку процесс должен выделять память только из специально для него выделенного пула, вначале создается локальный пул памяти. Далее определяется сам процесс (my_process).
Так как процессу нужна доступная память, для размещения его кода и данных, в его адресном пространстве должен находиться хотя бы один доступный регион. Наконец, процесс запускается. Запуск процесса предполагает создание непривилегированного потока, который выполняется внутри процесса. Точку входа (потоковую функцию), можно либо узнать по какой-то информации, сообщаемой ядру в форме конфигурации, либо определить соглашением вида "первая инструкция в модуле - всегда переход на код его инициализации". То же самое касается стека, его размер и адрес определяются

соглашениями.
Далее эта функция может создать привилегированное приложение, потоки ядра, примитивы синхронизации, присоединить обработчики прерываний к векторам, а также перехватить исключения, для отслеживания ошибок в процессе. Доверенное приложение может также использовать API для завершения процесса, если решит, что он функционирует некорректно.

7.10.2 Библиотеки поддержки

После рассмотрения того, как реализуются процессы со стороны ядра, стоит уделить немного внимания тому, как выглядит и реализуется пользовательский интерфейс. Одним из целей проектирования было обеспечить одинаковый интерфейс для приложений работающих как в пользовательском режиме, так и режиме ядра, для возможности их прозрачного переноса между различными устройствами. К сожалению, обеспечить полностью идентичный интерфейс довольно сложно. Если приложение пользуется прерываниями в виде ISR, таймерами и тому подобным, то без изменений перенести его в непривилегированный режим нельзя, так как, это порождает проблемы с безопасностью. Но, если обрабатывать прерывания в потоках, и пользоваться только потоками и примитивами синхронизации, то такой интерфейс может быть сделан универсальным.
Все объекты, которыми управляет ядро, такие как потоки и примитивы синхронизации, заменяются на эквивалентные структуры, которые в пользовательском режиме содержат лишь идентификатор (а также опциональные дополнительные поля). Например семафор или мьютекс выглядят так:

```
typedef struct _fx_sem_t {  unsigned int handle; } fx_sem_t;
typedef struct _fx_mutex_t {  unsigned int handle; } fx_mutex_t;
```

Методы для работы с ними, в большинстве случаев, просто отображаются на соответствующий системный вызов без какой-либо дополнительной работы в библиотеке:

```
int fx_sem_timedwait(fx_sem_t* sem, uint32_t timeout) {
  return (int) hal_intr_scall(SYSCALL_ID_fx_sem_timedwait, sem->handle, timeout, ...);
}
```

Используя такой подход можно определить подмножество основного интерфейса, которое доступно в обоих режимах работы процессора и приложения, написанные на этом интерфейсе, становятся универсальными.
В том случае, когда приложение работает в режиме ядра, его структура выглядит так, как показано на Рис. 97:

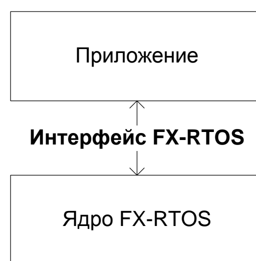

Рис. 97 Интерфейс приложения при отсутствии поддержки процессов.

При переносе его в пользовательский режим, функцию ядра выполняет библиотека поддержки, которая обращается уже к "настоящему" ядру (Рис. 98).

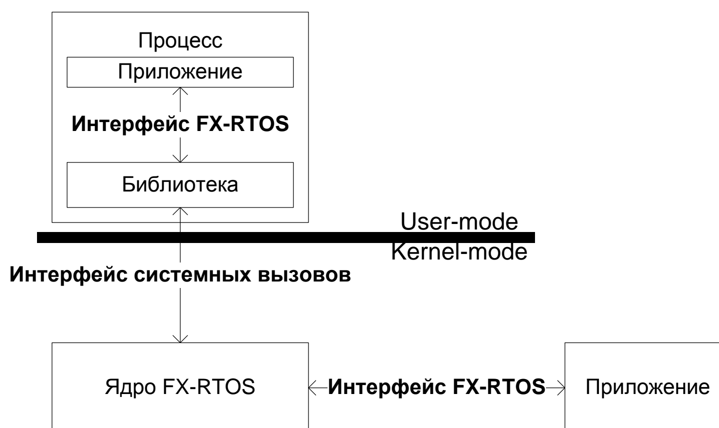

Рис. 98 Интерфейсы приложения при наличии процессов.

Доверенное и привилегированное приложение при этом работает так же, как и в первом случае. То есть процессы это "надстройка" над наноядром, которая не затрагивает доверенные приложения.

Динамическое создание процессов во время работы работает аналогичным образом, только все необходимые объекты не выделяются статически, а создаются с помощью механизма выделения памяти.

7.11 Резюме

По мере усложнения встраиваемых систем, большое значение имеет проблема изоляции частей приложения друг от друга, с целью повысить надежность. Для реализации этих возможностей используются аппаратные возможности ряда процессоров, позволяющие создавать изолированные процессы, которые имели бы ограничения на доступ к памяти и использование прочих системных ресурсов. В простых системах может использоваться модель с одним процессом, предполагающая разделение приложения на две части - привилегированную и непривилегированную. в более сложных случаях может использоваться и набор из нескольких процессов. В отличие от систем общего назначения, во встроенных системах обычно не предполагается создание новых процессов во время работы системы, их количество и функционалность определяются разработчиком системы на этапе проектирования, что позволяет упростить систему отказавшись от наличия в ядре ОС функций по динамичскому перераспределению памяти

между процессами.

В системах повышенной надежности может использоваться реализация процессов похожая на виртуальную машину: интерфейс ядра в этом случае похож на интерфейс предоставляемый оборудованием, а каждый процесс имеет собственную реализацию планировщика потоков и прочих механизмов. К сожалению, с такой реализацией связано довольно много накладных расходов, а также она не очень удобна для программирования в случае необходимости общения между процессами, поэтому большую популярность завоевал подход с разделяемым ядром, где процессам предоставляются более высокоуровневые сервисы, а ядро ОСРВ не запускается внутри каждого процесса, а разделяется между ними.

Реализация поддержки процессов с разделяемым ядром похожа во всех ОС. Она предполагает существование таблицы системных вызовов, с помощью которой контролируются точки входа в ядро. Используются также ряд других механизмов, таких как идентификаторы объектов для проверки аргументов. Реализация процессов может быть построена вокруг имеющегося наноядра, реализующего потоки и планировщик с минимальными изменениями.

[1] Wind River VxWorks 653 Platform 2010
[2] NXP Application note 10866: LPC1700 secondary USB bootloader
[3] Anderson and Dahlin "Operating systems. Principles and practice" ver.0.22 2011-2012 (p.43)
[4] Parkinson, Kinnan "Safety-Critical Software Development for Integrated Modular Avionics white paper. Temporal partitioning"
[5] INTEGRITY the most advanced RTOS technology brochure
[6] The Open Group. Portable Operating System Interface Base Specifications, Issue 7; 2.9.5.3 Thread Cancellation Cleanup Handlers 2008
[7] Robert Love "Linux Kernel Development" 3rd edition. Addison Wesley 2010 (p. 198)
[8] Ulruch Drepper, " Futexes are tricky" 2011

8 Взаимодействие процессов

Темы главы:

- Обзор механизмов межпроцессного взаимодействия
- Синхронизация процессов с использованием разделяемых объектов
- Обмен данными между процессами с использованием разделяемой памяти и сообщений

8.1 Необходимость взаимодействия процессов

В предыдущей главе была введена концепция процесса как изолированной вычислительной среды, предназначенной для изоляции ошибок и организации запуска нескольких независимых приложений на одном физическом устройстве. Хотя во главу угла выносится изоляция, и складывается впечатление, что основная задача в том, чтобы процессы, по возможности, не замечали присутствия друг друга, в реальности, в подобных системах, часто требуется возможность взаимодействия между процессами (inter-process communication, IPC).

Необходимость взаимодействия между процессами связана с тем, что, как правило, все они вовлечены в решение некоторой общей задачи. Например, системы авионики в самолетах требуют больших усилий по изоляции программных компонентов друг от друга, чтобы системы развлечения для пассажиров не могли повлиять на автопилот. В то же время, все эти системы работают над одной общей задачей - безопасным и комфортным для пассажиров полётом. В развлекательных системах для пассажиров могут отображаться данные о высоте и скорости полёта и тому подобное. Эти данные должны браться откуда-то извне, так как сам процесс, реализующий функции развлечения, очевидно, не должен иметь доступа к системам авионики и не может по собственной инициативе обращаться к структурам данных автопилота.

Другой пример связан непосредственно с тем, что формально независимые процессы, тем не менее, разделяют одну аппаратную платформу, поэтому им может требоваться синхронизация доступа к определенным ресурсам, которые используются несколькими процессами, но существуют в одном экземпляре. Это, правда, требует тщательного проектирования, чтобы некорректно работающий процесс не смог нарушить работу своего компаньона, но это отдельная тема.

Наконец, как уже говорилось в предыдущих главах, определенные достоинства имеет микроядерная организация операционной системы, в которой вся ОС представлена как набор взаимодействующих процессов, работающих поверх компактного и быстрого микроядра. Возможность быстрого обмена информацией между процессам критически важна для работы ОС с микроядерной архитектурой, так как практически все общение прикладных процессов с внешним миром осуществляется через механизмы межпроцессного взаимодействия: чтение файлов, взаимодействия по сети, работа с драйверами устройств и тому подобное.

Эффективные методы межпроцессного взаимодействия до сих пор являются предметом исследований (например, [1], [2]). К сожалению, начиная от микроядер первого поколения, таких как Mach и заканчивая современными реализациями, межпроцессное

взаимодействие содержит некоторые трудноустранимые накладные расходы, которые отсутствуют, когда необходимый функционал реализован в пространстве ядра. В последнем случае для обращения к сервисам ОС не требуется переключения процесса, поэтому такие методы в целом имеют более высокую производительность, благодаря чему микроядра, хотя и являются более простыми, защищенными и устойчивыми к атакам, не завоевали большой популярности в качестве ОС общего назначения. Иначе обстоит дело в системах повышенной надежности. В этом случае, ради достижения надежности, приходится идти на компромисс и мириться с некоторым снижением производительности, так что для встроенных систем, где существуют процессы с разным уровнем ответственности, микроядро в целом является более предпочтительным.

8.2 Обзор механизмов межпроцессного взаимодействия

После определения типичных задач, которые стоят перед разработчиками механизмов межпроцессного взаимодействия, прежде чем переходить к деталям, стоит вкратце рассмотреть различные способы взаимодействий, распространенные в существующих ОС. Очевидно, общение между процессами осуществляется через некоторую сущность, к которой оба (или более) участника взаимодействий должны иметь доступ. В качестве таких разделяемых между всеми процессами объектов могут выступать, например файлы на диске или сокеты в сетевой подсистеме. Что касается файлов, это не очень удачный выбор, так как, во-первых, их использование достаточно тяжеловесно (из-за того, что требуется участие файловой системы), а во-вторых, в случае, если такой механизм является базовым, он требует реализации файлов в ядре операционной системы, так как при размещении их в отдельном процессе, возникает порочный круг: для общения с процессом реализующим файлы, требуется процесс, реализующий файлы. Так что файлы не годятся в качестве универсального механизма IPC.

Подобные рассуждения справедливы и по отношению к сетевому стеку. Реализация, например, семейства протоколов TCP/IP требует довольно значительного объема кода и было бы крайне желательно выполнять этот код в изолированном процессе, особенно в системах повышенной надежности. Существует, впрочем, случай, когда использование сети как механизма для IPC оправдано. Это допустимо тогда, когда общающиеся процессы находятся на разных компьютерах. В таком случае использование универсального интерфейса для общения, как в предалах одного физического компьютера, так и в распределенной системе, может служить хорошим подспорьем, хотя высокой производительности от этого подхода ожидать не следует. Кроме того, в случае некорректно написанного приложения, можно случайно обеспечить доступ к нему не только для локальных процессов, но и внешних, через сеть, что может привести к проблемам с безопасностью.

Для общения между процессами находящимися в пределах одной вычислительной системы можно использовать более простое и очевидное решение - разделяемую или **общую память** (shared memory), которая показана на Рис. 99.

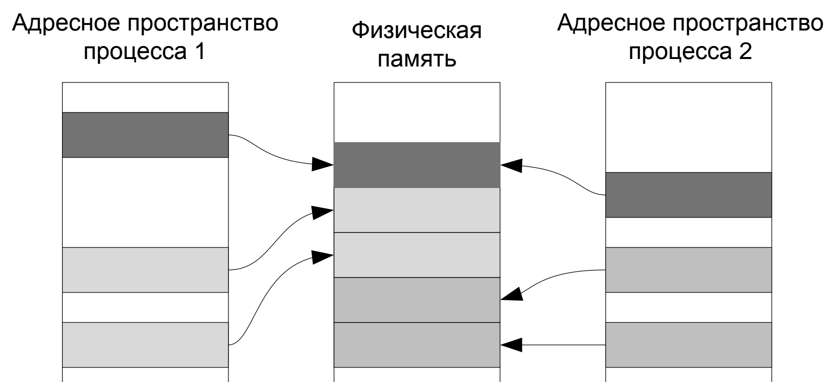

Рис. 99 Разделяемая память.

Хотя каждый процесс и имеет свое собственное адресное пространство, оно может быть настроено таким образом, что некоторые области виртуального адресного пространства отображены на одни и те же физические страницы, поэтому размещенная в них информация доступна всем процессам, которые имеют соответствующее отображение.

Многие стандартные API, в частности, POSIX, определяют наборы функций для создания разделяемых между процессами регионов памяти [3]. Подобные функции в ОС общего назначения часто подразумевают динамическое выделение региона памяти и правку таблиц трансляции участвующих процессов во время работы. Во встроенных системах, как правило, известно, какие процессы могут общаться и как именно, поэтому, в случае если необходимо иметь разделяемый регион памяти, это можно сделать статически. При таком подходе регионы разделяемой памяти описываются в виде конфигурационной информации, которая используется для настройки таблиц страниц на этапе начальной инициализации системы, и в дальнейшем размеры и адреса этих регионов не изменяются.

При условии правильно спроектированных процессов, когда все данные, получаемые от строннего процесса, тщательно проверяются перед использованием, можно организовать безопасный и быстрый обмен данными. К сожалению, обмен данными с помощью только общей памяти предполагает полинг, так как нету никакого способа "обратить внимание" другого процесса на то, что, например, появились новые данные для обработки. Иными словами, отсутствует механизм уведомления о событиях. В качестве механизма для отправки уведомлений процессу используется несколько средств, среди которых можно выделить два "классических": **сигналы** и **разделяемые объекты синхронизации**.

Сигналы - механизм, появившийся вместе с UNIX, использующийся для уведомления об асинхронных событиях. После отправки процессу сигнала, один из его потоков асинхронно выполняет некоторую процедуру (обработчик), которая была предварительно ассоциирована с этим сигналом. Поэтому, после получения сигнала, процесс-адресат может проанализировать разделяемую память и выполнить требуемые действия. Собственно, этот механизм повторяет логику работы аппаратуры, а сигнал выступает в роли обработчика прерывания. Из негативных сторон этого подхода: некоторая сложность программирования, так как сигналы прерывают выполнение потока в случайных местах, если они размаскированы, а также сложности с определением потока, которому будет доставлен сигнал. Существуют определенные правила, согласно которым сигналы доставляются потокам внутри процесса, но различные реализации ОС трактуют

эти правила достаточно свободно. К тому же, классические сигналы представлены, как и прерывания, одним битом, поэтому повторные уведомления вполне могут быть потеряны.
Более продвинутый механизм, который лишен недостатков сигналов - разделяемые примитивы синхронизации, например семафор. Он ничем не отличается от обычного семафора рассмотренного ранее, за исключением того, что ссылку на один и тот же объект могут иметь несколько процессов. Если поток в одном процессе начал ожидание такого семафора, то "разбудить" его могут как потоки в текущем процессе, так и потоки других процессов, которые имеют доступ к этому объекту.
Как и сигналы, интерфейс разделяемых семафоров является частью стандарта POSIX и более удобен в программировании. Это удобство заключается в том, что, во-первых, повторные уведомления накапливаются в счетчике семафора, поэтому не могут быть пропущены, во-вторых - "пробуждать" можно только те потоки, которые явно начали ожидание именно этого семафора, то есть уведомление не произойдет для них неожиданно, наконец, в третьих, в отличие от сигналов, семафоров может быть произвольное количество, в зависимости от нужд приложения.
Несмотря на то, что, теоретически, разделяемой памяти и семафоров достаточно для реализации любого механизма межпроцессного взаимодействия, эти объекты достаточно тяжеловесны. Разделяемая память хорошо работает тогда, когда нужно передать большие объемы данных, если же нужно передать пару килобайт или вообще несколько десятков байт, то правка ради этого таблиц страниц у нескольких процессов (с сопутствующей рассылкой межпроцессорных прерываний) это довольно существенные накладные расходы. Поэтому, для таких задач используются механизмы, которые передают данные непосредственно от отправителя к приемнику, без использования разделяемой памяти. Классическим примером такого механизма являются **каналы** UNIX (pipes).
Канал представляет собой небольшой буфер, для чтения и записи которого используются обычные операции для файлового ввода-вывода. Так как и чтение и запись - блокируемые операции, потоки-читатели могут блокироваться при попытке чтения из пустого канала, а потоки-писатели - при попытке записи в заполненный канал. Каналы являются однонаправленными, поэтому для коммуникации в обе стороны нужно использовать пару каналов. С точки зрения как отправителя так и получателя, канал, как и обычный файл, выдает просто последовательность байтов. Реализация какого-то протокола общения поверх этого потока возлагается на взаимодействующие стороны, что может быть не очень удобно. Очевидный недостаток каналов: возможность прерывания байтового потока от источника или приёмника "на полуслове", что существенно усложняет возможный протокол взаимодействия. Поэтому дальнейшее развитие этой идеи привело к средству, которое считается де-факто стандартом в реализации как микроядерных операционных систем, так и для общения между процессами. Этим средством является **обмен сообщениями**.
Сообщение - это порция данных определенной длины. В зависимости от реализации возможны как сообщения фиксированного размера, так и произвольного. В отличие от каналов, обмен сообщениями носит атомарный характер, то есть нельзя получить

сообщение частично, оно может либо прийти целиком, либо не прийти вообще и эта атомарность обеспечивается ядром.

В стандарте POSIX предусмотрен интерфейс для **асинхронных собщений**, то есть приемник и источник действуют полностью независимо друг от друга. Существует и альтернативный подход: **синхронный обмен сообщениями**. В этом случае, приемник и источник данных должны выполняться согласованно и быть в определенных точках выполнения, чтобы обмен мог состояться. Хотя реализация таких объектов сложнее, они, в целом, обеспечивают более высокую производительность, так как, в отличие от асинхронных сообщений, данные копируются напрямую от источника с приемнику, минуя промежуточную буферизацию в ядре.

Рассмотренные выше методы далеко не единственные и в различных ОС применяются и другие, в том числе довольно экзотические методы. Например, ОС Solaris реализует механизм межпроцессного взаимодействия называемый "двери" (doors), которые представляют собой средство для вызова функций в контексте другого процесса [4]. Так как в микроядерной архитектуре типичным сценарием обмена данными является обращение процесса-клиента к процессу-серверу, "двери" призваны оптимизировать именно такой сценарий. Дверь представляет собой системный объект, создаваемый сервером, причем с этим объектом ассоциируется некоторая функция. Когда процесс клиент обращается к "двери", происходит передача управления от клиента к одному из потоков сервера с вызовом в его контексте ассоциированной с "дверью" функции, которая выполняет запрос клиента и возвращает ему управление.

8.3 Синхронизация с помощью разделяемых объектов

Целесообразно начать с разделяемых объектов, как самого простого и эффективного способа синхронизации процессов без обмена данными. Главный вопрос здесь, каким образом обеспечить "разделение" объекта. Когда объекты создает сам процесс, он получает идентификатор, в данном случае, требуется как-то получить идентификатор на уже существующий объект, созданный другим процессом.

Здесь многое зависит от способа создания новых процессов. В UNIX и Windows, например, при создании нового процесса могут наследоваться идентификаторы объектов ядра. Разделяемые объекты при этом получаются за счет того, что два процесса ссылаются на один и тот же объект ядра, такой как семафор, поэтому родитель и потомок могут работать с ним практически так же, как это делают потоки внутри одного процесса.

В том случае, когда процессы нельзя создавать во время работы, используется другой подход. Так как теперь у процессов изначально нет никаких общих данных, нужен какой-то глобально уникальный идентификатор, который позволил бы каждому из них найти нужный объект синхронизации. В качестве такого идентификатора используется имя в виде строки. То есть с объектами, такими, например, как семафоры, может быть сопоставлено имя, с помощью которого их можно "открыть", то есть получить их идентификатор. Интерфейс выглядит следующим образом:

```
sem_t* sem_open(const char* name, int oflags, ...);
int sem_close( sem_t* sem );
int sem_unlink(const char* name);
```

С помощью первых двух функции можно открыть или закрыть семафор, имеющий определенное имя, а с помощью последней - удалить семафор с заданным именем. После получения идентификатора, с семафором можно работать используя функции sem_wait/sem_post, семантика которых была рассмотрена ранее. В FX-RTOS используется аналогичное API.

Так как ОС не может знать заранее, какие процессы с какими захотят взаимодействовать, уникальность имени полностью возлагается на пользователя (точнее, разработчика программ, использующих IPC). Межпроцессное взаимодействие с помощью разделяемых объектов идентично синхронизации потоков внутри одного процесса, различие заключается только в том, каким образом два процесса получают ссылку на объект.

В качестве разделяемых объектов могут использоваться любые объекты ядра, то есть, теоретически, можно получать ссылки не только на семафоры, но и на мьютексы и даже потоки, чтобы управлять ими с помощью прямых уведомлений с помощью функций типа thread_resume. Но нужно понимать, что при этом велик риск разрушить всю систему изоляции между процессами, если один из них будет иметь возможность нарушить работу другого. В том числе и по этой причине, POSIX допускает существование в виде именованных объектов только областей разделяемой памяти, семафоров и очередей (которые будут рассмотрены далее).

Наличие именованных объектов, как механизма взаимодействия между процессами, требует наличия в системе специального компонента, который занимается обслуживанием базы данных имён объектов - сервера именования или сервера имён (naming server). В некоторых ОС (например, Windows или Symbian) этот компонент тесно связан с таблицами идентификаторов [5]. Устроен он довольно просто, имеется некоторый контейнер, например список, и набор функций, которые позволяют добавлять в него объекты и искать их по имени. Именованные объекты также расширяются полем имени. Во время выполнения функций типа open, происходит поиск в контейнере. Если объект нашелся, счетчик ссылок на него увеличивается внутри процедуры поиска, далее требуется вставить этот объект в таблицу идентификаторов и так далее. Если ж объект не найден, то в зависимости от флагов создается новый объект также с последующей вставкой в таблицу идентификаторов объектов данного процесса. Так или иначе, процесс, имя некоторое строковое имя, получает идентификатор объекта с которым может далее работать.

Возможен и альтернативный путь, когда обслуживанием базы данных именованных объектов занимается один из пользовательских процессов. В таком случае с именем должен быть сопоставлен некоторый уникальный идентификатор, который можно использовать для создания ссылки на объект. В его роли может выступать указатель на объект ядра. Другие процессы могут обращаться к серверу имен используя IPC для получения этого идентификатора по имени, а затем использованием его для создания ссылки на объект.

С точки зрения приложений, работа с именованными объектами согласно, например, стандарту POSIX происходит как показано ниже:

```
sem_t* sem = sem_open("/my_semaphore", O_CREAT, 0600, 0);
```

Все взаимодействующие процессы должны "открыть" объект, после чего использовать его для синхронизации. Кроме имени указываются также флаги, определяющие, создавать ли семафор, если он не существует, а также права доступа и значение счетчика для вновь созданного семафора. Если семафор уже существовал на момент вызова, на него будет создана ссылка без создания нового объекта.

8.4 Коммуникация с общей памятью

Именованные объекты позволяют синхронизировать выполнение процессов и обеспечить совместное использование разделяемых ресурсов, но с их помощью нельзя организовать обмен данными. Одним из способов реализации обмена данными между процессами являтся использование именованных объектов совместно с разделяемой памятью.

Согласно стандарту POSIX, регионы разделяемой памяти могут быть такими же именованными объектами, как и семафоры, поэтому приложение может "открыть" оба объекта и использовать семафор для уведомлений, а общую память для коммуникации.

В ОС, которые не поддерживают управление памятью во время работы, предполагается, что все подобные области создаются на этапе конфигурации, но это не мешает реализовать такой же программный интерфейс.

При использовании этих механизмов важно обеспечить корректное функционирование процессов, такое, чтобы, например, в случае наличия одного сервера и нескольких клиентов, один из последних не мог вызвать ошибку на стороне сервера и помешать последнему обслуживать других клиентов.

Первое очевидное решение, когда все процессы имеют один общий ргеион разделяемой памяти, по понятным причинам не подходит, так как любой процесс видит всё то, что отправляется серверу любым другим процессом, а также ответы сервера. Для решения этой проблемы используется следующий подход: каждому процессу выделяется свой собственный регион разделяемой памяти, который виден серверу, но не виден другим процессам. Также для уведомления сервера используется один семафор, тогда как каждый клиент использует свой собственый семафор для уведомлений со стороны сервера (Рис. 100, семафоры, связанные с процессами, показаны окружностями).

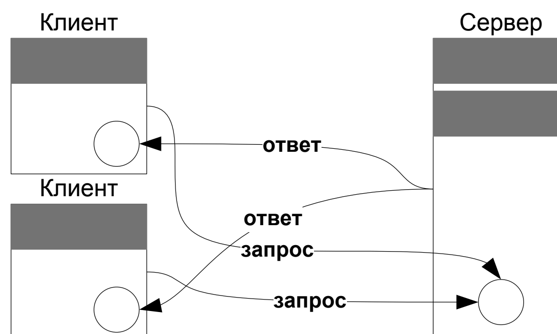

Рис. 100 Межпроцессное взаимодействие с использованием семафоров и разделяемой памяти.

Как можно увидеть, при реализации такой архитектуры взаимодействий, даже потенциальная злонамеренность клиента не сможет вызвать нарушение работы как сервера, так и других процессов. Каждый клиент видит только собственные запросы и

ответы, предназначенные для него же. Процесс не может послать запрос от имени другого процесса, а потенциальные излишние уведомления сервера не приведут к катастрофическим последствиям, ведь сервер остается работоспособен и будет продолжать обслуживать других клиентов, если они имеют более высокий приоритет, с точки зрения сервера.

Подобным образом можно организовать не только клиент-серверную модель приложения, но и другие механизмы. Несмотря на кажущуюся простоту, это требует тщательного проектирования. Вот некоторые вопросы, на которые должны быть ответы, если требуется разработка надежной встроенной системы: семафор не обеспечивает защиту от инверсии приоритета, а мьютекс небезопасен в том смысле, что владеющий им процесс может намеренно его не освободить и парализовать работу всех других процессов, использующих этот мьютекс; процессы работают независимо, а значит, любой из них может быть в произвольный момент времени завершен, поэтому, если процессы не доверенные, протокол общения должен быть спроектирован так, чтобы позволять продолжение работы после рестарта любого из процессов. Вопросы проектирования коммуникационных протоколов выходят за рамки данной книги, но это лишь один из примеров, иллюстрирующих сложность проектирования встроенных приложений.

Что касается использования именованных объектов из приложений, то для удобства обычно предполагается, что создавать объект могут как клиент, так и сервер (хотя возможно явно указать, кто именно должен создавать объект с помощью флагов). Если к моменту создания объекта одной из сторон, он уже существует, функция создания работает как функция открытия и создания ссылки на существующий объект.

8.5 Асинхронный обмен сообщениями

Использование разделяемой памяти и именованных примитивов хорошо для обмена большими объемами данных. Зачастую это требует реализации на стороне приложения довольно сложных протоколов взаимодействия. Вместе с тем, часто встречающейся практической задачей является обмен небольшими объемами данных. Другой пример: уведомления с передачей данных, когда одному из процессов надо уведомить другой о наступлении некоторого события, причем таких событий может быть множество и каждое из них требует отдельной реакции. Используя предыдущие методы, нужно либо создавать большой массив семафоров, каждый из которых соответствовал бы определенному событию, либо выделять общую память, синхронизировать доступ к ней, использовать отдельный семафор для уведомлений и так далее. Ни один из этих подходов нельзя назвать простым.

В отличие от синхронного обмена, асинхронный обмен сообщениями реализуется проще и исторически появился первым, поэтому его имеет смысл рассмотреть в первую очередь. По аналогии с тем, как приложение может создавать именованные семафоры и разделяемые регионы памяти, создается специальный объект - очередь сообщений. В отличие от каналов, очередь сообщений двунаправленная, а ее асинхронность предполагает, что, как приемник сообщений, так и их источник, работают независимо друг от друга. Это также означает буферизацию сообщений внутри объекта, поэтому очередь плохо подходит для обмена большими объемами данных, так как они должны

копироваться как минимум дважды: от источника во внутренний буфер очереди, а затем из буфера к приемнику (Рис. 101).

Рис. 101 Очередь сообщений.

Что касается размера самих сообщений, то, в зависимости от реализации очереди, они могут быть как фиксированного так и произвольного размера. В системах жесткого реального времени имеет смысл применять сообщения фиксированного размера для достижения большего детерминизма при выделении памяти. Достаточно реализовать передачу с помощью очереди объемов данных равных размеру указателя, тогда при необходимости с ее помощью можно передавать как данные по значению, так и буферы произвольного размера (Рис. 102).

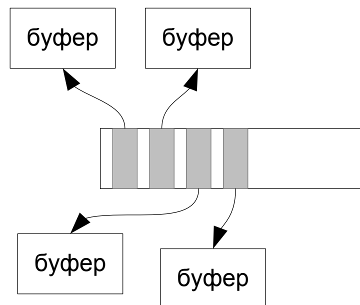

Рис. 102 Передача буферов с данными через очередь сообщений.

Так как данные всегда копируются в очередь и из нее, от пользователя скрыто внутреннее устройство буферов, поэтому очередь можно реализовать как комбинацию из объектов "очередь сообщений" и "блоки памяти", которые были рассмотрены ранее.

Некоторые интерфейсы, в частности, POSIX, определяют также ряд дополнительных возможностей. Например, возможность очереди посылать вместе с сообщением сигналы (когда очередь переходит из пустого состояния в непустое), а также связывать с сообщения приоритет, так что очередь престает быть просто FIFO-очередью, а становится очередью с приоритетами. Интерфейс выглядит так:

```
mqd_t mq_open(const char* name, int, ...);
int mq_close(mqd_t mq);
ssize_t mq_receive(mqd_t mq, char* buf, size_t sz, unsigned* prio);
int mq_send(mqd_t mq, const char* buf, size_t sz, unsigned prio);
int mq_unlink(const char* name);
```

Очереди FX-RTOS имеют аналогичный интерфейс, за исключением поддержки приоритетов.

Несмотря на то, что очереди сообщений позволяют реализовать взаимодействие между процессами без использования общей памяти, большие возможности имеет также совмещение этих двух механизмов. Пока сообщения имеют небольшой размер, либо являются уведомлениями о событиях, они могут целиком передаваться через очередь. В том случае, когда требуется передать большой объем данных, данные могут копироваться в разделяемую память, а затем указатель на эти данные передается с помощью очереди. Если такой подход позволяют соображения безопасности, можно разместить все данные процесса в разделяемой области данных, так что указатели на участки памяти могут

передаваться через очередь сообщений напрямую, то есть не требуется копирования данных и другой процесс может получить к ним доступ очень быстро.

В ранних микроядерных ОС, таких как Mach, асинхронный обмен сообщениями был основным способом IPC [6], для общения между сервером и клиентами. Из-за необходимости многократных копирований данных, IPC на основе асинхронных сообщений не отличалось производительностью, частично поэтому микроядра первого поколения и не получили распространения за пределами академической среды.

Из-за атомарности получения сообщений, а также из-за буферизации в пространстве ядра, очередь может сохранять сообщения даже в том случае, если один или даже оба участника обмена были аварийно завершены. Также, поскольку очередь одновременно является объектом синхронизации, не требуется никаких дополнительных усилий для синхронизации дотсупа к ней, как это было бы в случае разделяемой памяти, поэтому очередь сообщений может быть легко использования для разработки параллельных приложений с использованием только одного этого объекта и без какой-либо дополнительной синхронизации.

Несмотря на более удобные интерфейсы, если сравнивать с разделяемой памятью, реализация клиент-серверного взаимодействия с использованием очередей не отличается простотой. Хотя очередь допускает обмен данными в обоих направлениях, если отправить в очередь запрос, а потом попытаться прочитать из нее ответ, то можно прочитать лишь свой отправленный запрос, если к тому моменту сервер не успел его забрать. Поэтому используются как минимум две очереди, одна для отправки запросов, а вторая для получения ответов (Рис. 103).

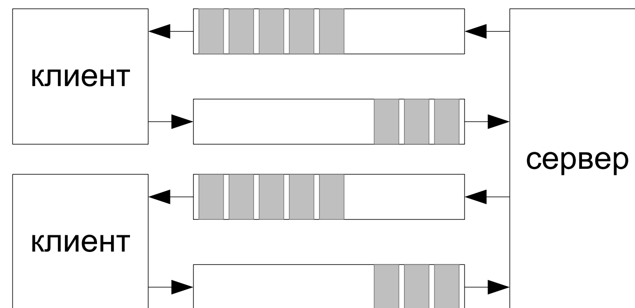

Рис. 103 Клиент-серверное взаимодействие с использованием асинхронных очередей сообщений.

Здесь возникает еще одна проблема: из-за асинхронности работы как клиента так и сервера, если были отправлены несколько запросов, то ответы на них могут приходить в различном порядке и от пользователя требуется снова реализовывать довольно нетривиальный коммуникационный протокол, например, снабжать как запрос так и ответ неким уникальным идентификатором, который позволил бы клиенту понять, ответ на что он получил. Применяется также частный случай очереди сообщений, называемый почтовым ящиком. Почтовый ящик это очередь сообщений которая содержит только один элемент. При таком дизайне исчезает асинхронность и возможность отправить несколько запросов сразу, но зато упрощается протокол.

Возможен также вариант реализации, когда очередь не интегрирована в ядро ОС таким образом, что является примитивом синхронизации, то есть не вызывает блокировки потоков. В этом случае возможно только проверять наличие в ней сообщений и

извлекать сообщения, которые там уже присутствуют на момент выполнения запроса. Такую очередь можно рассматривать как некоторое устройство, подобное, скажем, универсальным асинхронным приемо-передатчикам (UART). Из-за того, что такой объект может не знать о потоках, планировщике и так далее, на его основе можно организовать обмен данными не только в случае разделяемого между процессами ядра ОС, но даже и в случае виртуальных машин, когда обмениваться данными между собой могут потоки разных ОС и приложений, работающих в изолированном окружении.

8.6 Синхронный обмен сообщениями

Дальнейшие исследования в области ОС и межпроцессного взаимодействия, привели к появлению синхронного обмена сообщениями. Синхронный обмен устроен таким образом, что выполнение обоих участников взаимодействия должно быть согласовано. Если асинхронное сообщение можно отправить в очередь даже тогда, когда никто его не ожидает, то в случае синхронного обмена и клиент и сервер должны вызвать соответствующие функции, чтобы обмен состоялся.

Как и в случае с асинхронным обменом, существует некоторый объект, называемый портом, который является разделяемым объектом. Методы обычно включают три функции: send/receive/reply. Клиент вызывает функцию отправки, а сервер - функцию приема. Если в момент вызова нету ожидающего потока, как клиент так и сервер блокируются в этой точке. После того, как к порту приходит поток (клиент к серверу или наборот), сервер разблокируется и начинает обработку данных клиента. Данные от клиента копируются в буфер сервера напрямую из адресного пространства клиента, поэтому данный подход имеет существенное преимущество по скорости перед асинхронным обменом сообщениями. После получения запроса от клиента последний не разблокируется, а ожидает, пока сервер не ответит на запрос с использованием функции reply, которая копирует ответ от сервера клиенту, опять же минуя промежуточную буферизацию в ядре. Таким образом, устраняется необходимость реализации асинхронного протокола. Синхронные сообщения можно назвать обобщением понятия вызова функции, когда клиент, отправляя сообщение, вызывает функцию сервера, которая получает сообщение как входные аргументы, формирует ответ и точно так же возвращает результат запрашивающему потоку. Схематично этот процесс показан на Рис. 104. Вначале два потока работают независимо (1), предположим, что сервер вызвал функцию receive первым и блокируется из-за отсутствия клиентов. Далее, когда клиент вызывает функцию send, он блокируется и одновременно с этим разблокируется сервер (2). Так как клиет заблокирован, сервер может скопировать сообщение непосредственно из его адресного пространства и приступить к обработке запроса (3). Наконец, сервер формирует ответ и вызывает функцию reply, которая копирует ответ обратно в адресное пространство клиента и разблокирует его (4). Обмен состоялся, после этого оба потока продолжают работать независимо (5).

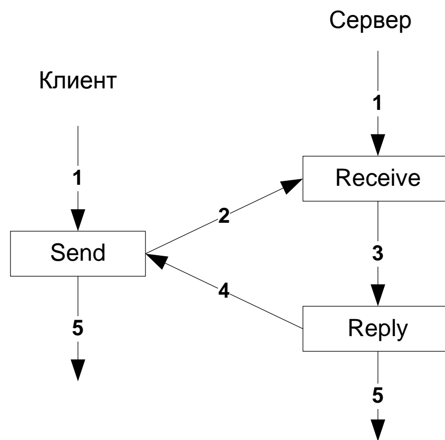

Рис. 104 Синхронный обмен сообщениями.

Программный интерфейс для синхронного обмена сообщениями не стандартизирован, поэтому в каждой ОС, поддерживающей такой тип взаимодействия, вводится свой собственный интерфейс. В FX-RTOS он выглядит так:

```
int fx_msg_port_open(fx_msg_port_t* port, const char* name, unsigned flags);
int fx_msg_port_unlink(const char* name);
int fx_msg_port_close(fx_msg_port_t* port);

int fx_msg_send(
	fx_msg_port_t* port,
	void* data, size_t buf_sz, size_t req_sz,
	size_t* reply_sz, bool block);

int fx_msg_receive(
	fx_msg_port_t* port,
	void* data, size_t buf_sz,
	size_t* max_reply_sz, size_t* rcvd_sz, bool block);

int fx_msg_reply(void* data, size_t sz);
```

Первые три функции служат для управления разделяемым именованным объектом. Функция send использует один и тот же буфер и для запроса и для получения ответа, поэтому в нее передаются две переменные определяющие размер: одна из них указывает на размер данных, которые нужно отправить, а вторая - максимальный объем данных для приема. Для упрощения синхронизации предполагается, что функцию reply должен вызвать тот же поток, который получил сообщение через receive, так что в функцию reply передается только буфер для ответа и его размер, вся остальная информация хранится во внутренних структурах данных потока.

Хотя механизм выглядит довольно простым, его реализация гораздо сложнее, чем асинхронный обмен. Главным образом из-за того, что при любом запросе дважды возникает обращение к памяти другого процесса и копирование данных, притом что в системе жесткого реального времени процессы могут завершаться асинхронно и в произвольный момент времени. То есть должны быть предусмотрены механизмы для предотвращения некорректных обращений к памяти, если, например процесс клиента аварийно завершается в тот момент когда сервер копировал данные в или из его буфера.

Вопрос о том, что выступает в качестве разделяемого между процессами объекта, реализующего передачу сообщений, является дискуссионным. Может применяться подход, использовавшийся для асинхронных сообщений, где имеется разделяемый объект "очередь". В случае синхронного обмена сообщениями, этот объект ничего не буферизует, а используется только для того, чтобы была какая-то точка, где могут встретиться источник и приемник (Рис. 105 справа).

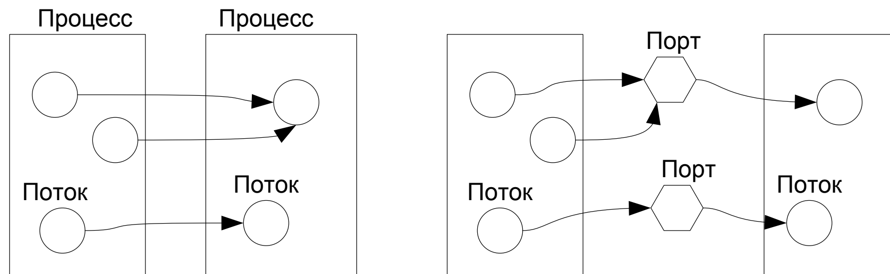

Рис. 105 Взаимодействующие объекты при синхронном обмене сообщениями.

Но такой подход не является единственным. В некоторых ОС, например в микроядрах семейства L4 [7], сообщения передаются напрямую между потоками, то есть не используется никаких промежуточных объектов (Рис. 105 слева). При этом, разумеется, отправитель сообщений должен как-то узнать идентификатор получателя. Каким образом это происходит, микроядро не определяет. Например, может существовать выделенный поток с предварительно определенным идентификатором, у которого регистрируются серверы. Так как узнать адрес сервера нужно только единожды, этот поток не будет являться узким местом во время работы. Возможны и третьи варианты. Например, в ОСРВ QNX, используется схема с большим количеством уровней. Помимо порта, существует еще объект "соединение". Процесс должен предварительно установить соединение с портом, затем сообщения посылаются в соединение. Такая схема позволяет "отключить" все потоки данного процесса от общения с сервером, путем одного вызова "отсоединения"(Рис. 106).

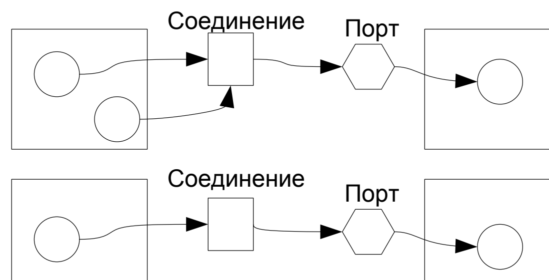

Рис. 106 Использование промежуточных объектов "соединение".

В отличие от случая с асинхронными сообщениями, в синхронной модели межпроцессного общения может быть использована оптимизация, использующая манипуляции с таблицей страниц. Пока сервер не ответил клиенту, предполагается, что клиент находится в заблокированном состоянии, поэтому страницы, содержащие буфер клиента, могут быть отображены на адресное пространство сервера без необходимости копирования. К сожалению, это возможно только в том случае, если все буферы выровнены на размер страницы. В противном случае, использовать отображения проблематично даже для страниц, которые находятся "в середине" сообщения, поскольку

это потребует дополнительного согласования смещений. Эта ситуация показана на Рис. 107. Сообщение занимает 4 страницы, причем область данных не выровнена (показана серым цветом). Если отобразить промежуточные две страницы, хранящие среднюю часть данных, в адресное пространство сервера, сервер не сможет обращаться к этим данным по тем же смещениям, что и клиент.

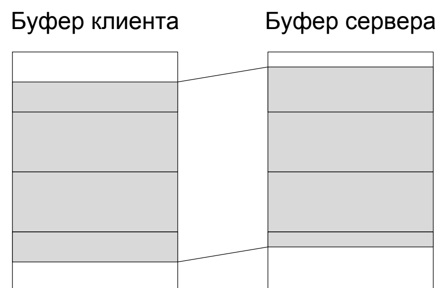

Рис. 107 Смещения данных в буфере сообщения.

8.7 Особенности реализации синхронного обмена сообщениями

Интерфейс синхронного обмена сообщениями выглядит очень просто и логично. Тем не менее, простота эта обманчива. При попытке реализовать микроядро с использованием синхронного обмена сообщениями в качестве основного механизма IPC возникает множество проблем, которые требуют расширения описанного интерфейса. Некоторые из них будут вкратце рассмотрены в этом разделе.

Представление синхронного обмена сообщениями как вызова функций не учитывает один важный момент, а именно: то, что клиент и сервер это разные потоки. В случае если у этих потоков разный приоритет, обмен сообщениями может привести к инверсии приоритета. На первый взгляд, не совсем ясно, где она может возникать, так как отсутствуют разделяемые ресурсы. На деле оказывается, что выполнение сервера является как бы продолжением выполнения запроса клиента. В том случае, если клиент имеет приоритет больший, чем сервер, то получается, что высокоприоритетный поток-клиент оказывается вынужден ждать потенциально низкоприоритетный поток-сервер, который может быть вытеснен третьим потоком и таким образом вызвать неограниченную по времени инверсию приоритета. Как и в случае с мьютексами, используются несколько стратегий, как избежать такого развития событий. Первый и самый простой случай: обязать проектировщика системы подбирать приоритеты таким образом, чтобы приоритет сервера всегда был больше, чем приоритеты клиентов, которые могут к нему обращаться. Это похоже на вариант с потолком приоритета: просто и надежно. Альтернативные подходы предусматривают реализацию некоторого механизма наследования приоритета, когда в момент получения сообщения, приоритет сервера повышается до максимального приоритета ожидающих клиентов.

Как и в случае использования мьютексов, требуется также тщательное проектирование протокола взаимодействия, с целью исключить петли в графе отношений "А обращается к Б". Существование в таком графе петли, очевидно, может привести к взаимоблокировке.

Попытки реализации поверх сообщений некоторого API, такого, например, как файловое API POSIX приводит к проблеме непрерывного буфера. Поскольку серверу необходимо

передать вместе с пользовательским буфером, содержащим данные для чтения или записи еще и некоторый заголовок, описывающий операцию, возникает проблема, как снабдить переданный буфер заголовком. Очевидно, что выделение в каждой функции чтения или записи нового буфера большего размера с копированием в него дополнительного заголовка приведет мало того что с катастрофическому падению производительности, так еще и отсутствию детерминизма из-за неопределенностей, связанных с выделением памяти. Решением этой проблемы являются так называемые векторные сообщения. Сообщение в этом случае состоит уже не из единого буфера, а из набора буферов. Пример такого API можно увидеть например в QNX:

```
int MsgSendv(int coid, const iov_t* siov, int sparts, const iov_t* riov, int rparts );
int MsgReceivev(int chid, const iov_t * riov, int rparts, struct _msg_info * info );
int MsgReplyv(int rcvid, int status, const iov_t* riov, int rparts );
```

Сообщение описывается массивом структур iov_t, каждая из которых содержит указатель и размер буфера. Здесь уже не будет проблемой прикрепить к сообщению любую дополнительную информацию, такую как заголовок.

Наконец, использование синхронных сообщений требует как правило требует наличия какого-то механизма уведомлений. Рассмотрим следующий пример: приложение, используя сообщения, отправляет запрос на чтение данных из сокета (сокеты реализованы в отдельном процессе, обслуживающем сеть). После получения запроса, сервер видит, что сокет не имеет данных в настоящий момент и клиент должен заблокироваться. Можно заблокировать серверный поток и избежать пробуждения клиента из-за отсутствия ответа на сообщение. Но очевидно, что это плохое решение: если множество клиентов обратится к сети, и количество клиентов превысит количество серверных потоков, все потоки сервера окажутся заблокированы на неопределенное время и система станет неработоспособной. Для решения таких проблем синхронный обмен сообщениями часто используется совместно с некоторым другим механизмом уведомления, который позволяет реализовать отложенные ответы на сообщения. В таком случае сервер, определив, что ответить на сообщение немедленно невозможно, сохраняет реквизиты клиента, запоминая его в каких-то внутренних структурах данных, а потом, после получения данных для ответа, использует сохраненные реквизиты для фактического ответа. Соответственно, серверные потоки не могут быть заблокированы входящими запросами даже в том случае, когда данных для ответа нет, сервер остается работоспособным.

8.8 Резюме

Хотя процессы и нужны для запуска независимых программ и изоляции их друг от друга, встроенное ПО часто работает над решением общей для всех процессов задачи, поэтому необходимы механизмы взаимодействия между процессами.

Для взаимодействия используются именованные глобальные объекты, которые позволяют процессам получить ссылку на такой объект и впоследствии использовать его так же, как если бы он был локальным объектом.

Простейший способ организации синхронизации и коммуникации между процессами это использование именованных семафоров и разделяемых областей памяти.

В более сложных случаях, а также в случаях, когда использование разделяемой памяти нежелательно, могут использоваться различные виды обмена сообщениями: синхронный или асинхронный.

Асинхронный обмен сообщениями предполагает буферизацию сообщений внутри ядра, а также усложняет протокол коммуникации. Синхронные сообщения можно рассматривать как обобщение вызова функций, поэтому во многих микроядерных ОС он используется как основной механизм IPC.

[1] G. Heiser, K. Elphinstone, "L4 Microkernels: The Lessons from 20 Years of Research and Deployment" 2016
[2] Horst Wenske; University of Karlsruhe, "Design and iplementation of Fast Local IPC for L4 microkernel" 2002
[3] Standard for Information Technology—Portable Operating System Interface Base Specifications, Issue 7. 2008 (p1898)
[4] J. Mauro, R. McDougall "Solaris Internals. Core kernel components" Sun Microsystems Press 2000 (p. 469)
[5] Jane Sales, "Symbian OS Internals Real-time Kernel Programming" John Wiley & Sons 2005 (p.161)
[6] Ю. Вахалия, " UNIX изнутри", "Питер" 2003. (6.5.2, с. 263)
[7] N. Sadeque, R. I. Mutia, "OKL4 API System Calls"