МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

А. С. ЛУКАНОВ

СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С. П. Королева» в качестве учебного пособия для обучающихся по основной образовательной программе высшего образования по направлению подготовки 02.03.03 Математическое обеспечение и администрирование информационных систем

САМАРА
Издательство Самарского университета
2020

УДК 004.031.43(075)
ББК 32.973.26-018.1я7
Л840

Рецензенты: канд. техн. наук, проф. СамГУПС В. А. З а с о в,
канд. физ.-мат. наук, доц. Самарского университета
В. П. Ц в е т о в

Луканов, Александр Сергеевич
Л840 **Системы реального времени:** учебное пособие / *А. С. Луканов*. – Самара: Издательство Самарского университета, 2020. – 156 с.: ил.

ISBN 978-5-7883-1522-5

Учебное пособие содержит курс лекций по операционным системам реального времени и создаваемым на их платформе специализированным информационным системам реального времени. Пособие также содержит комплекс лабораторных работ по одноименной дисциплине.

Предназначено для студентов направления подготовки «Математическое обеспечение и администрирование информационных систем», также других специальностей, изучающих дисциплины «Операционные системы», «Операционные системы и оболочки».

Подготовлено на кафедре информатики и вычислительной математике.

УДК 004.031.43(075)
ББК 32.973.26-018.1я7

ISBN 978-5-7883-1522-5

ОГЛАВЛЕНИЕ

Глава 1. ВВЕДЕНИЕ В СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ. ОПЕРАЦИОННЫЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

§1. Функционирование информационных систем в «Реальном масштабе времени»

В настоящее время в документах и публикациях с различной тематикой встречаются слова о требовании, поддержке и т.д. «работы в режиме реального времени», «режима реального времени» или просто «реального времени». Что же такое «режим реального времени» применительно к компьютерным системам? Постараемся представить различные современные точки зрения на это понятие.

Каноническое определение системы реального времени дано Дональдом Гиллиесом и выглядит так:

«Системой реального времени является такая система, корректность функционирования которой определяется не только корректностью выполнения вычислений, но и временем, в которое получен требуемый результат. Если требования по времени не выполняются, то считается, что произошел отказ системы». *Другие добавляют:* «Поэтому необходимо, чтобы было гарантировано [аппаратными и программными средствами и алгоритмами обработки] выполнение требований по времени. Гарантия выполнения требований по времени необходима, чтобы поведение системы было предсказуемо. Также желательно, чтобы система обеспечивала высокую степень использования ресурсов, чтобы удовлетворять требованиям по времени [с минимальными затратами]».

Хорошим примером является робот, который должен брать что-либо с ленты конвейера. Объекты на конвейере движутся и робот имеет некоторый небольшой интервал времени для того,

чтобы схватить объект. Если робот опоздает, то объекта уже не будет на месте, и поэтому работа будет неверной, даже если робот переместил захват в правильное положение. Если робот поспешит, то объекта там еще не будет, более того, робот может заблокировать движение объектов.

Другой пример – цикл управления самолетом, летящим на автопилоте. Датчики самолета должны постоянно передавать измеренные данные в управляющий компьютер. Если данные измерений теряются, то качество управления самолетом падает, возможно вместе с самолетом.

Отметим следующую особенность: в примере с роботом имеем настоящий, «жесткий» режим реального времени (*hard real time*), и если робот опоздает, то это приведет к полностью ошибочной операции. Однако это мог бы быть режим «квазиреального» времени (*soft real time*), если бы опоздание робота приводило бы только к потере производительности. Многое из того, что сделано в области программирования в реальном времени, в действительности работает в режиме «квазиреального» времени. Грамотно разработанные системы, как правило, имеют уровень безопасности/коррекции поведения даже для случая, когда вычисления не закончились в необходимый момент, так что если компьютер чуть-чуть не успевает, то это может быть скомпенсировано.

Бывает, что термин «система реального времени» применяют в значении «интерактивная система» (*on-line*). Часто это просто рекламный ход. Например, системы заказа билетов или системы складского учета не являются системами «реального времени», так как человек-оператор без проблем перенесет задержку ответа на несколько сотен миллисекунд.

Также можно встретить случаи, когда термин «система реального времени» применяют просто в значении «быстродействующая система». Необходимо отметить, что определение «реального времени» не является синонимом для определения «быстродействующая». Еще раз: термин «система

реального времени» не означает, что система дает ответ на воздействие мгновенно – задержка может достигать секунд и более – но означает тот факт, что гарантируется некоторая максимально возможная величина задержки ответа, что позволяет системе решать поставленную задачу. Необходимо также отметить, что алгоритмы, обеспечивающие гарантированное время ответа, часто имеют меньшую среднюю производительность, чем алгоритмы, которые не гарантируют время ответа.

Из приведенного выше можно сделать выводы:

– термин «система реального времени» может трактоваться так:

«Системой реального времени является такая система, корректность функционирования которой определяется не только корректностью выполнения вычислений, но и временем, в которое получен требуемый результат. Если требования по времени не выполняются, то считается, что произошел отказ системы».

Для того чтобы система могла удовлетворить требованиям, предъявляемым к системам реального времени, аппаратные, программные средства и алгоритмы работы системы должны гарантировать заданные временные параметры реакции системы. Время реакции не обязательно должно быть очень маленьким, но оно должно быть гарантированным (и отвечающим поставленным требованиям);

– использование термина «система реального времени», определенного выше, для обозначения интерактивных и высокопроизводительных систем неверно;

– термин «квазиреальное время» (*soft real-time*) хотя и используется, но четко не определен. До его четкого определения вряд ли возможно его применение в документах (кроме рекламных). С уверенностью можно сказать, что смысл термина «реальное время» трактуется специалистами по-разному в зависимости от области их профессиональных интересов, от того, являются они теоретиками или практиками, и даже просто от личного опыта и круга общения;

– практически все системы промышленной автоматизации являются системами реального времени;

– принадлежность системы к классу систем реального времени никак не связана с ее быстродействием. Например, если ваша система предназначена для контроля уровня грунтовых вод, то даже выполняя измерения с периодичностью один раз за полчаса, она будет работать в реальном времени.

Исходные требования к времени реакции системы и другим временным параметрам определяются или техническим заданием на систему, или просто логикой ее функционирования. Например, шахматная программа, рассчитывающая следующий ход более часа, явно работает не в реальном времени. Однако точное определение «приемлемого времени реакции» не всегда является простой задачей, а в системах, где одним из звеньев служит человек, подвержено влиянию субъективных факторов.

Интуитивно понятно, что быстродействие системы реального времени должно быть тем больше, чем больше скорость протекания процессов на объекте контроля и управления.

Итак, системой реального времени (СРВ) будем называть аппаратно-программный комплекс, реагирующий за предсказуемое время на непредсказуемый поток внешних событий.

Это определение означает:

• СРВ должна успеть отреагировать на событие, произошедшее на объекте, в течение времени, критического для этого события. Величина критического времени для каждого события определяется объектом и самим событием и, естественно, может быть разной, но время реакции системы должно быть предсказано (вычислено) при создании системы. Отсутствие реакции в предсказанное время считается ошибкой для систем реального времени;

• СРВ должна успевать реагировать на одновременно происходящие события. Даже если два или больше внешних события происходят одновременно, система должна успеть среагировать на каждое из них в течение интервала времени, критического для этих событий.

Различают системы реального времени двух типов – системы жесткого реального времени и системы мягкого реального времени.

I. Системы жесткого реального времени не допускают никаких задержек реакции ни при каких условиях, так как:

- результаты могут оказаться бесполезны в случае опоздания;
- может произойти катастрофа в случае задержки реакции;
- стоимость опоздания может оказаться бесконечно велика.

Примеры систем жесткого реального времени – бортовые системы управления, системы аварийной защиты, регистраторы аварийных событий.

Многие теоретики ставят здесь точку, из чего следует, что время реакции в «жестких» системах может составлять и секунды, и часы, и недели. Однако большинство практиков считают, что время реакции в системах «жесткого» реального времени должно быть все-таки минимальным. Идя на поводу у практиков, так и будем считать. Разумеется, однозначного мнения о том, какое время реакции свойственно «жестким» системам, нет. Более того, с увеличением быстродействия микропроцессоров – это время имеет тенденцию к уменьшению, и если раньше в качестве границы называлось значение 1 мс, то сейчас, как правило, называется время порядка 100 мкс.

II. Системы мягкого реального времени характеризуются тем, что задержка реакции не критична, хотя и может привести к увеличению стоимости результатов и снижению производительности системы в целом.

Пример – работа компьютерной сети. Если система не успела обработать очередной принятый пакет, это приведет к тайм-ауту на передающей стороне и повторной посылке (в зависимости от протокола, конечно). Данные при этом не теряются, но производительность сети снижается.

Основное отличие между системами жесткого и мягкого реального времени можно выразить так: система жесткого реального времени никогда не опоздает с реакцией на событие, а система мягкого реального времени – не должна опаздывать с реакцией на событие.

§2. Области применения СРВ

Приведем некоторые сферы применения СРВ, чтобы продемонстрировать, насколько современная жизнь привязана к системам данного типа.

1. Военная и космическая области: бортовое и встраиваемое оборудование:

• системы измерения и управления, радары;

• цифровые видеосистемы, симуляторы;

• ракеты, системы определения положения и привязки к местности.

2. Промышленность:

• автоматические системы управления производством (АСУП), автоматические системы управления технологическим процессом (АСУТП);

• автомобилестроение: симуляторы, системы управления мотором, автоматическое сцепление, системы антиблокировки колес;

• энергетика: сбор информации, управление данными и оборудованием;

• телекоммуникации: коммуникационное оборудование, сетевые коммутаторы, телефонные станции;

• банковское оборудование.

3. Товары широкого потребления:

• мобильные телефоны;

• цифровые телевизионные декодеры;

• цифровое телевидение (мультимедиа, видеосерверы);

• компьютерное и офисное оборудование (принтеры, копиры).

§3. Понятие операционной системы реального времени

Назовем операционной системой реального времени (ОСРВ, RTOS) такую систему, которая может быть использована для построения систем жесткого реального времени.

Это определение выражает отношение к операционным системам реального времени как к объекту, содержащему необходимые инструменты, но также означает, что ими еще необходимо правильно воспользоваться.

Формальные определения ОСРВ:

• операционная система, в которой успешность работы любой программы зависит не только от её логической правильности, но и от времени, за которое она получила этот результат. Если система не может удовлетворить временным ограничениям, должен быть зафиксирован сбой в её работе;

• согласно стандарту POSIX 1003.1, «реальное время в операционных системах – это способность операционной системы обеспечить требуемый уровень сервиса в определённый промежуток времени».

Операционные системы (ОС) общего назначения, особенно многопользовательские, ориентированы на оптимальное распределение ресурсов компьютера между пользователями и задачами (системы разделения времени). В операционных системах реального времени подобная задача отходит на второй план, все отступает перед главной задачей – успеть среагировать на события, происходящие на объекте.

Другое отличие – применение операционной системы реального времени всегда связано с аппаратурой, объектом, событиями, происходящими на объекте. Операционная система реального времени ориентирована на обработку внешних событий. Именно это приводит к коренным отличиям (по сравнению с ОС общего назначения) в структуре системы, функциях ядра, построении системы ввода-вывода. Операционная система реального времени может быть похожа по пользовательскому интерфейсу на ОС общего назначения (к этому, кстати, стремятся почти все производители операционных систем реального времени), однако устроена она совершенно иначе.

Кроме того, применение ОСРВ всегда конкретно. Если ОС общего назначения обычно воспринимается пользователями (не разработчиками) как уже готовый набор приложений, то ОСРВ

служит только инструментом для создания конкретного аппаратно-программного комплекса реального времени.

Операционная система, соответствующая классу реального времени (РВ), должна отвечать следующим базовым требованиям:

• быть многозадачной и допускающей вытеснение (preemptable);

• обладать понятием приоритета для потоков;

• поддерживать предсказуемые механизмы синхронизации;

• обеспечивать механизм наследования приоритетов;

• поведение ОС должно быть известным и предсказуемым (задержки обработки прерываний, задержки переключения задач, задержки драйверов и т.д.); это значит, что во всех сценариях рабочей нагрузки системы должно быть определено максимальное время отклика.

Время реакции на события является важнейшей характеристикой ОСРВ.

Характерное время реакции на события в зависимости от области применения может быть следующим:

• математическое моделирование − несколько микросекунд;

• радиолокация − несколько миллисекунд;

• складской учет − несколько секунд;

• торговые операции − несколько минут;

• управление производством − несколько минут;

• химические реакции − несколько часов.

Как видно, времена значительно различаются, поэтому:

• для каждой задачи необходима соответствующая мощность аппаратных средств;

• события сообщаются системе посредством запросов на прерывание (IRQ), поэтому ключевым параметром является время реакции системы на прерывание (interrupt latency).

В настоящий момент, к сожалению, нет общепринятых методологий измерения этого параметра, поэтому он является «полем битвы» маркетинговых служб производителей систем реального времени.

§4. Архитектура операционных систем реального времени

Примем как очевидные следующие моменты.

1. Когда-то операционных систем совсем не было.

2. Через некоторое время после их появления возникло направление ОС РВ.

3. Все ОС РВ являются многозадачными операционными системами. Задачи делят между собой ресурсы вычислительной системы, в том числе и процессорное время.

Четкой границы между ядром (*Kernel)* и операционной системой нет. Различают их, как правило, по набору функциональных возможностей. Ядра предоставляют пользователю такие базовые функции, как планирование и синхронизация задач, межзадачная коммуникация, управление памятью и т.п. Операционные системы в дополнение к этому имеют файловую систему**,** сетевую поддержку, интерфейс с оператором и другие средства высокого уровня.

По своей внутренней архитектуре ОС РВ можно условно разделить на монолитные ОС, ОС на основе микроядра и объектно-ориентированные ОС. Графически различия в этих подходах иллюстрируются рисунками 4.1, 4.2, 4.3. Преимущества и недостатки различных архитектур достаточно очевидны, поэтому подробно мы на них останавливаться не будем.

Рис. 4.1. ОС РВ с монолитной архитектурой

Рис. 4.2. ОС РВ на основе микроядра

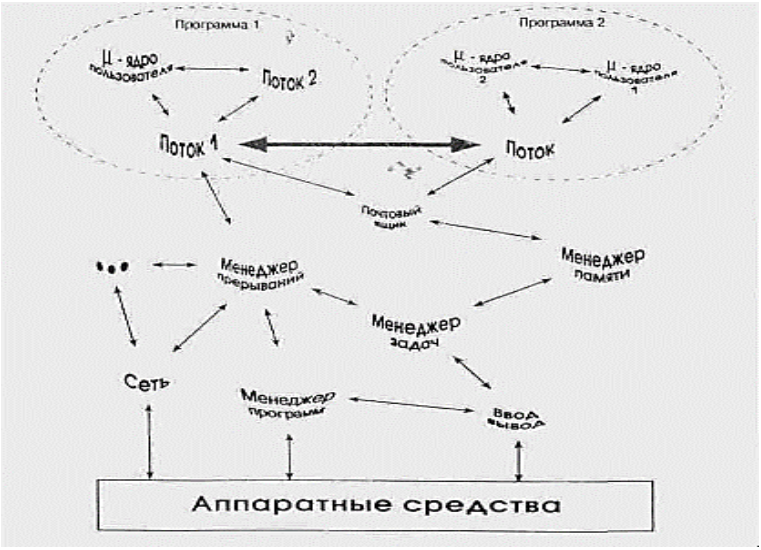


Рис. 4.3. Объектно-ориентированная ОС РВ

§5. Основные сервисы ядра ОСРВ

Как было указано ранее, основой любой среды исполнения в реальном времени является ядро. Все современные ОСРВ являются многозадачными. Таким образом, все программное обеспечение, включая также часть операционной системы, организуется в виде набора последовательных процессов. Исходя из этого, ядро может обеспечивать сервис пяти типов:

Синхронизация ресурсов. Метод синхронизации требует ограничить доступ к общим ресурсам (данным и внешним устройствам). Наиболее распространенный тип примитивной синхронизации – двоичный семафор, обеспечивающий избирательный доступ к общим ресурсам. Так, процесс, требующий защищенного семафором ресурса, вынужден ожидать до тех пор, пока семафор не станет доступным, что свидетельствует об освобождении ожидаемого ресурса, и, захватив ресурс, установить семафор. В свою очередь, другие процессы также будут ожидать доступа к ресурсу вплоть до того момента, когда семафор возвратит соответствующий ресурс системе распределения ресурсов. Системы, обладающие большей ошибкоустойчивостью, могут иметь счетный семафор. Этот вид семафора разрешает одновременный доступ к ресурсу лишь определенному количеству процессов.

Межзадачный обмен. Часто необходимо обеспечить передачу данных между программами внутри одной и той же системы Кроме того, во многих приложениях возникает необходимость взаимодействия с другими системами через сеть. Внутренняя связь может быть осуществлена через систему передачи сообщений. Внешнюю связь можно организовать либо через датаграмму (наилучший способ доставки), либо по линиям связи (гарантированная доставка). Выбор того или иного способа зависит от протокола связи.

Разделение данных. В прикладных программах, работающих в реальном времени, наиболее длительным является

сбор данных. Данные часто необходимы для работы других программ или нужны системе для выполнения каких-либо своих функций. Во многих системах предусмотрен доступ к общим разделам памяти. Широко распространена организация очереди данных. Применяется много типов очередей, каждый из которых обладает собственными достоинствами.

Обработка запросов внешних устройств. Каждая прикладная программа в реальном времени связана с внешним устройством определенного типа. Ядро должно обеспечивать службы ввода/вывода, позволяющие прикладным программам осуществлять чтение с этих устройств и запись на них. Для приложений реального времени обычным является наличие специфического для данного приложения внешнего устройства. Ядро должно предоставлять сервис, облегчающий работу с драйверами устройств. Например, давать возможность записи на языках высокого уровня – таких, как Си или Паскаль.

Обработка особых ситуаций. Особая ситуация представляет собой событие, возникающее во время выполнения программы. Она может быть синхронной, если ее возникновение предсказуемо, как, например, деление на нуль. А может быть и асинхронной, если возникает непредсказуемо, как, например, падение напряжения. Предоставление возможности обрабатывать события такого типа позволяет прикладным программам реального времени быстро и предсказуемо отвечать на внутренние и внешние события. Существуют два метода обработки особых ситуаций – использование значений состояния для обнаружения ошибочных условий и использование обработчика особых ситуаций для прерывания ошибочных условий и их корректировки.

Кроме того, важнейшей функцией ядра является **диспетчеризация** (планирование). Планировщик должен определять, какому процессу должно быть передано управление, а также должен определить время, выделяемое каждому процессу.

§6. Задачи, процессы и потоки

Существуют различные определения термина «задача» для многозадачной ОС РВ. Мы будем считать задачей набор операций (машинных инструкций), предназначенный для выполнения логически законченной функции системы. При этом задача конкурирует с другими задачами за получение контроля над ресурсами вычислительной системы.

Принято различать две разновидности задач: процессы и потоки.

Процесс представляет собой «среду обитания» запущенной программы. Т.е. при запуске программы ОС создает процесс, выделяя программе все необходимые ей (для нормального функционирования) ресурсы.

Поток – это часть кода программы, которая непосредственно исполняется на процессоре в данный момент. Поэтому потоки могут использовать общие ресурсы одного процесса.

Хорошим примером многопоточной программы является редактор текста MS Word, где в рамках одного приложения может одновременно происходить и набор текста, и проверки правописания.

Преимущества потоков.

1. Так как множество потоков способно размещаться внутри одного *EXE*-модуля, это позволяет экономить ресурсы как внешней, так и внутренней памяти.

2. Использование потоками общей области памяти позволяет эффективно организовать межзадачный обмен сообщениями (достаточно передать указатель на сообщение). Процессы не имеют общей области памяти. Поэтому ОС должна либо целиком скопировать сообщение из области памяти одной задачи в область памяти другой (что для больших сообщений весьма накладно), либо предусмотреть специальные механизмы, которые позволили бы одной задаче получить доступ к сообщению из области памяти другой задачи.

3. Как правило, контекст потоков меньше, чем контекст процессов, а значит, время переключения между задачами-потоками меньше, чем между задачами-процессами.

4. Так как все потоки, а иногда и само ядро РВ размещаются в одном *EXE*-модуле, значительно упрощается использование программ-отладчиков (*debugger*).

Недостатки потоков.

1. Как правило, потоки не могут быть подгружены динамически. Чтобы добавить новый поток, необходимо провести соответствующие изменения в исходных текстах и перекомпилировать приложение. Процессы, в отличие от потоков, подгружаемы, что позволяет динамически изменять функции системы в процессе ее работы. Кроме того, так как процессам соответствуют отдельные программные модули, они могут быть разработаны различными компаниями, чем достигается дополнительная гибкость и возможность использования ранее наработанного ПО.

2. То, что потоки имеют доступ к областям данных друг друга, может привести к ситуации, когда некорректно работающий поток способен испортить данные другого потока. В отличие от этого процессы защищены от взаимного влияния, а попытка записи в «не свою» память приводит, как правило, к возникновению специального прерывания по обработке «исключительных ситуаций».

Реализация механизмов управления процессами и потоками, возможность их взаимного сосуществования и взаимодействия определяются конкретным ПО РВ.

§7. Основные свойства задач (процессов и потоков)

Как правило, вся важная, с точки зрения операционной системы, информация о задаче хранится в унифицированной структуре данных – управляющем блоке (*Task Control Block, TCB*).

В блоке хранятся такие параметры, как имя и номер задачи, верхняя и нижняя границы стека, ссылка на очередь сообщений, статус задачи, приоритет и т. п.

Приоритет – это некое целое число, присваиваемое задаче и характеризующее ее важность по сравнению с другими задачами, выполняемыми в системе. Приоритет используется в основном планировщиком задач для определения того, какая из готовых к работе задач должна получить управление. Различают системы с динамической и статической приоритетностью. В первом случае приоритет задач может меняться в процессе исполнения, в то время как во втором приоритет задач жестко задается на этапе разработки или во время начального конфигурирования системы.

Контекст задачи – это набор данных, содержащий всю необходимую информацию для возобновления выполнения задачи с того места, где она была ранее прервана. Часто контекст хранится в *TCB* и включает в себя такие данные, как счетчик команд, указатель стека, регистры *CPU* и *FPU* и т. п. Планировщик задач в случае необходимости сохраняет контекст текущей активной задачи и восстанавливает контекст задачи, назначенной к исполнению. Такое переключение контекстов и является, по сути, основным механизмом ОС РВ при переходе от выполнения одной задачи к выполнению другой.

Состояние (статус) задачи. С точки зрения операционной системы, задача может находиться в нескольких состояниях. Число и название этих состояний различаются от одной ОС к другой. По-видимому, наибольшее число состояний задачи определено в языке *Ada*. Тем не менее практически в любой ОС РВ загруженная на выполнение задача может находиться, по крайней мере, в трех состояниях.

1. Активная задача – это задача, выполняемая системой в текущий момент времени.

2. Готовая задача – это задача, готовая к выполнению и ожидающая у планировщика своей «очереди».

3. Блокированная задача – это задача, выполнение которой приостановлено до наступления определенных событий. Такими событиями могут быть освобождение необходимого задаче ресурса, поступление ожидаемого сообщения, завершение интервала ожидания и т. п.

Пустая задача (*Idle Task*) – это задача, запускаемая самой операционной системой в момент инициализации и выполняемая только тогда, когда в системе нет других готовых для выполнения задач. Пустая задача запускается с самым низким приоритетом и, как правило, представляет собой бесконечный цикл «ничего не делать». Наличие пустой задачи предоставляет операционной системе удобный механизм отработки ситуаций, когда нет ни одной готовой к выполнению задачи.

Многократный запуск задач. Как правило, многозадачные ОС позволяют запускать несколько копий одной и той же задачи. При этом для каждой такой копии создается свой *TCB* и выделяется своя область памяти. В целях экономии памяти может быть предусмотрено совместное использование одного и того же исполняемого кода для всех запущенных копий. В этом случае программа должна обеспечивать повторную входимость (реентерабельность). Кроме того, программа не должна использовать временные файлы с фиксированными именами и должна корректно осуществлять доступ к глобальным ресурсам.

Реентерабельность (повторная входимость) означает возможность без негативных последствий временно прервать выполнение какой-либо функции или подпрограммы, а затем вызвать эту функцию или подпрограмму снова. Частным проявлением реентерабельности является рекурсия, когда тело подпрограммы содержит вызов самой себе. Классическим примером нереентерабельной системы является *DOS*, а типичной причиной нереентерабельности служит использование глобальных переменных. Предположим, что у нас есть функция, реализующая низкоуровневую запись на диск, и пусть она использует глобальную переменную *write_sector*, которая устанавливается в

соответствии с параметром, передаваемым этой функции при вызове. Предположим теперь, что Задача *A* вызывает эту функцию с параметром 3, то есть хочет записать данные в сектор номер 3. Допустим, что когда переменная *write_sector* уже равна 3, но сама запись еще не произведена, выполнение Задачи *A* прерывается и начинает выполняться Задача *B*, которая взывает ту же функцию, но с аргументом 10. После того как запись в сектор номер 10 будет произведена, управление рано или поздно вернется к Задаче *A*, которая продолжит работу с того же места. Однако, так как переменная *write_sector* имеет теперь значение 10, данные Задачи *A*, предназначавшиеся для сектора номер 3, будут вместо этого записаны в сектор номер 10. Из приведенного примера видно, что ошибки, связанные с нереентерабельностью, трудно обнаружить, а последствия они могут вызвать самые катастрофические.

§8. Переходные состояния задач (процессов и потоков) в ОСРВ

Рассмотрим подробнее, что такое процесс. **Процесс** – это динамическая сущность программы, ее код в процессе своего выполнения. Имеет:

- собственные области памяти под код и данные, включая значения регистров и счетчика команд;
- собственный стек;
- собственное отображение виртуальной памяти (в системах с виртуальной памятью) на физическую;
- собственное состояние.

Процесс может находиться в одном из следующих типичных состояний:

- «остановлен» – процесс остановлен и не использует процессор (например, в таком состоянии процесс находится сразу после создания);
- «терминирован» – процесс терминирован и не использует процессор (например, процесс закончился, но еще не удален операционной системой);

- «ждет» – процесс ждет некоторого события (им может быть аппаратное или программное прерывание, сигнал или другая форма межпроцессного взаимодействия);
- «готов» – процесс не остановлен, не терминирован, не ожидает, не удален, но и не работает (например, процесс не может получить доступ к процессору, если в данный момент выполняется другой, более высокоприоритетный процесс);
- «выполняется» – процесс выполняется и использует процессор. В ОСРВ это обычно означает, что этот процесс является самым приоритетным среди всех процессов, находящихся в состоянии «готов».

Рассмотрим более подробно состояния процесса и переходы из одного состояния в другое (рис. 8.1).

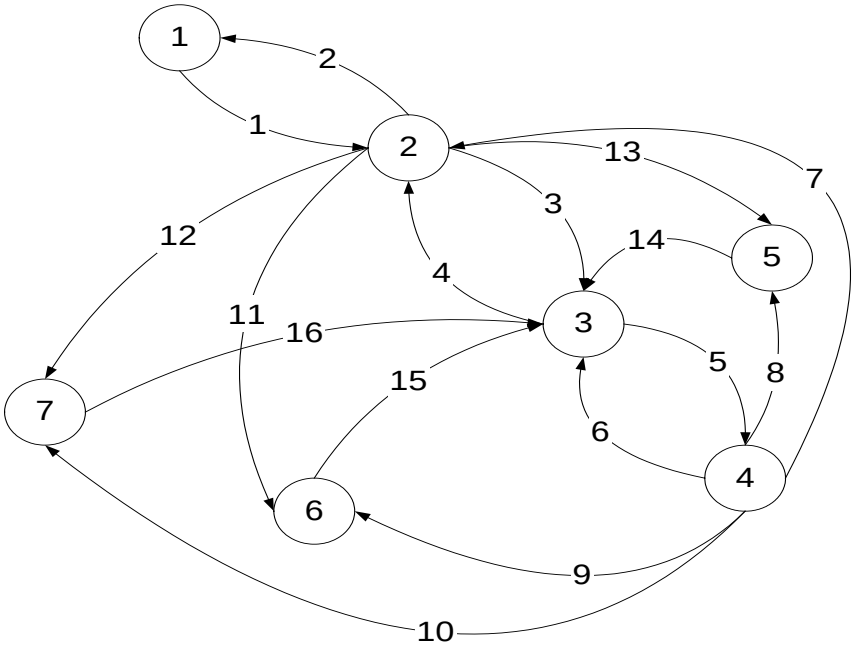


Рис. 8.1. Переходные состояния процессов ОС РВ

Состояния:

1) не существует,
2) не обслуживается,
3) готов,
4) выполняется,
5) ожидает ресурс,
6) ожидает назначенное время,
7) ожидает события.

Переходы:

1. Переход 1-2 создание процесса;
2. Переход 2-1 уничтожение процесса;
3. Переход 2-3 активизация процесса диспетчером;
4. Переход 3-2 деактивизация процесса;
5. Переход 3-4 загрузка на выполнение процесса диспетчером;
6. Переход 4-3 требование обслуживания от процессора другим процессом (preemption – приоритетное переключение);
7. Переход 4-2 завершение процесса;
8. Переход 4-5 блокировка процесса до освобождения требуемого ресурса;
9. Переход 4-6 блокировка процесса до истечения заданного времени;
10. Переход 4-7 блокировка процесса до прихода события;
11. Переход 2-6 активизация процесса приводит к ожиданию временной задержки;
12. Переход 2-7 активизация процесса приводит к ожиданию события;
13. Переход 2-5 активизация процесса приводит к ожиданию освобождения ресурса;
14. Переход 5-3 активизация процесса из-за освобождения ожидавшегося ресурса;
15. Переход 6-3 активизация процесса по истечении заданного времени;

16. Переход 7-3 активизация процесса из-за прихода ожидавшегося события.

Таким образом, каждый процесс имеет свой жизненный цикл, состоящий из 4 стадий:

1) создание,
2) загрузка,
3) выполнение,
4) завершение.

§9. Планирование задач

Важной частью любой ОС РВ является планировщик задач. Несмотря на то что в разных источниках он может называться по-разному (диспетчер задач, супервизор и т. п.), его функции остаются теми же: определить, какая из задач должна выполняться в системе в каждый конкретный момент времени. Самым простым методом планирования, не требующим никакого специального ПО и планировщика как такового, является использование *циклического алгоритма* в стиле *round robin*:

```
void main (void)
{
        for(;;){
                task0();
                task1();
                task2();
               /* и т. д. */
      }
}
```

Каждая «задача», представляющая собой отдельную подпрограмму, выполняется циклически. При этом надо придерживаться следующих правил:

1. Подпрограммы не должны содержать циклов ожидания, в стиле

```
while (TRUE) {
        if(switch_up()){
                 lamp_off();
                break;
                }
}
```

2. Подпрограммы должны выполнять свою работу как можно быстрее, чтобы дать возможность работать следующей подпрограмме.

3. При необходимости подпрограмма может сохранять свое окружение и текущие результаты, чтобы в следующем цикле возобновить работу с того же места.

Можно отметить следующие преимущества циклического алгоритма.

1. Простота использования и прозрачность для понимания.

2. Если исключить из рассмотрения прерывания, система полностью детерминирована. Задачи всегда вызываются в одной и той же последовательности, что позволяет достаточно просто произвести анализ «наихудшего случая» и вычислить максимальную задержку.

3. Минимальные размеры кода и данных. Кроме того, в отличие от алгоритмов с вытеснением, для всех задач необходим только один стек.

4. Отсутствуют ошибки, обусловленные «гонками».

К недостаткам циклического алгоритма можно отнести отсутствие приоритетности и очередей. К тому же задачи вызываются независимо от того, должны ли они в данный момент что-либо делать или нет, а на прикладного программиста ложится максимальная ответственность за работоспособность системы.

Перейдем теперь к другому широко используемому алгоритму планирования. Речь пойдет о режиме *разделения времени***.** Существуют различные реализации в рамках этого алгоритма, и некоторые западные специалисты даже различают такие, в общем-то идентичные для нас понятия, как *time-slicing (нарезание времени)* и *time-sharing (разделение времени)*. Как правило, алгоритм реализуется следующим образом: каждой задаче отводится определенное количество квантов времени (обычно кратно 1 мс), в течение которых задача может монопольно занимать процессорное время. После того как заданный интервал времени истекает, управление передается следующей готовой к выполнению задаче, имеющей наивысший приоритет. Та, в свою очередь, выполняется в течение отведенного для нее промежутка времени, после чего все повторяется в стиле *round robin*. Легко заметить, что такой алгоритм работы может привести к определенным проблемам. Представим, что в системе работают 7 задач, 3 из которых имеют высокий приоритет, а 4 – низкий. Низкоприоритетные задачи могут никогда не получить управление, так как три высокоприоритетные задачи будут делить все процессорное время между собой. Единственную возможность для низкоприоритетных задач получить управление предоставляет ситуация, когда все высокоприоритетные задачи находятся в блокированном состоянии.

Для решения этой проблемы применяется прием, получивший название *равнодоступность* (*fairness*). При этом реализуется принцип адаптивной приоритетности, когда приоритет задачи, которая выполняется слишком долго, постепенно уменьшается, позволяя менее приоритетным задачам получить свою долю процессорного времени. Равнодоступность применяется главным образом в многопользовательских системах и редко применяется в системах реального времени.

Кооперативная многозадачность – это еще один алгоритм переключения задач, с которым широкие массы компьютерной общественности знакомы по операционной системе Windows.

Задача, получившая управление, выполняется до тех пор, пока она сама по своей инициативе не передаст управление другой задаче. По сути это продолжение идеологии *round robin*, и нет нужды объяснять, почему алгоритм кооперативной многозадачности в чистом виде мало применяется в системах реального времени.

Приоритетная многозадачность с вытеснением – это, по-видимому, наиболее часто используемый в ОС РВ принцип планирования. Основная идея состоит в том, что высокоприоритетная задача, как только для нее появляется работа, немедленно прерывает (вытесняет) низкоприоритетную. Другими словами, если какая-либо задача переходит в состояние готовности, она немедленно получает управление, если текущая активная задача имеет более низкий приоритет. Такое «вытеснение» происходит, например, когда высокоприоритетная задача получила ожидаемое сообщение, освободился запрошенный ею ресурс, произошло связанное с ней внешнее событие, исчерпался заданный интервал времени и т. п.

Заканчивая рассмотрение основных принципов планирования задач, необходимо отметить, что тема эта далеко не исчерпана. Диапазон систем реального времени весьма широк, начиная от полностью статических систем, где все задачи и их приоритеты заранее определены, до динамических систем, где набор выполняемых задач, их приоритеты и даже алгоритмы планирования могут меняться в процессе функционирования. Существуют, например, системы, где каждая отдельная задача может участвовать в любом из трех алгоритмов планирования или их комбинации (вытеснение, разделение времени, кооперативность).

В общем случае алгоритмы планирования должны соответствовать критериям оптимальности функционирования системы. Однако, если для систем «жесткого» реального времени такой критерий очевиден: «ВСЕГДА и ВСЁ делать вовремя», то для систем «мягкого» реального времени это может быть, например, минимальное «максимальное запаздывание» или средневзвешенная

своевременность завершения операций. В зависимости от критериев оптимальности могут применяться алгоритмы планирования задач, отличные от рассмотренных. Например, может оказаться, что планировщик должен анализировать момент выдачи критичных по времени управляющих воздействий и запускать на выполнение ту задачу, которая отвечает за ближайшие из них (алгоритм *earliest deadline first, EDF*).

Необходимо отметить, что в одной вычислительной системе могут одновременно сосуществовать задачи и «жесткого», и «мягкого» реального времени, и что только одна из этих задач, обладающая наивысшим приоритетом, может быть по-настоящему детерминированной.

Не стоит особо увлекаться приоритетами. Если система нормально работает, когда все задачи имеют одинаковый приоритет, то и слава Богу. Если нет, то можно присвоить высокий приоритет «критической» задаче и низкий приоритет всем остальным. Если у вас больше одной «критической» задачи, при недостаточном быстродействии системы имеет смысл рассмотреть многопроцессорную конфигурацию или, отказавшись от ПО РВ, перейти к простому циклическому алгоритму.

Как правило, разработчики стараются свести свою систему реального времени к наиболее простым конфигурациям, характерным для систем «жесткого» реального времени, иногда даже в ущерб эффективности использования вычислительных ресурсов. Причина понятна: сложные динамические системы весьма трудно анализировать и отлаживать, поэтому лучше заплатить за более мощный процессор, чем иметь в будущем проблемы из-за непредвиденного поведения системы. В связи с этим большинство существующих систем реального времени представляют собой статические системы с фиксированными приоритетами. Часто в системе реализуется несколько «режимов» работы, каждый из которых имеет свой набор выполняемых задач с заранее заданными приоритетами. Значительная часть особо ответственных систем по-прежнему реализуется без применения коммерческих ОС РВ вообще.

§10. Планирование периодических процессов

Очевидно, что одной из специфических черт информационных систем реального времени является периодичность большинства процессов и потоков этих систем. Действительно, внешние события, на которые система реального времени должна реагировать, можно разделить на ***периодические*** (возникающие через регулярные промежутки времени) и ***непериодические*** (возникающие непредсказуемо). Возможно наличие нескольких потоков событий, которые система должна обрабатывать.

В зависимости от времени, затрачиваемого на обработку каждого из событий, может оказаться, что система не в состоянии своевременно обработать все события. Если в систему поступает ***m*** периодических событий, событие с номером ***i*** поступает с периодом ***Pi*** и на его обработку уходит ***Ci*** секунд работы процессора, все потоки могут быть своевременно обработаны только при выполнении условия

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \, . \qquad (10.1)$$

Система реального времени, удовлетворяющая условию (10.1), называется ***поддающейся планированию*** или ***планируемой***. Соотношение ***Pi/Ci*** является просто частью процессорного времени, используемого ***i***-м процессом, а сама сумма – это ***коэффициент использования (или коэффициент загруженности) процессора***, который, естественно, не может быть больше 1.

В качестве примера рассмотрим систему с тремя периодическими сигналами с периодами 100, 200, 500 ***мс*** соответственно. Если на обработку этих сигналов уходит 50, 30,

100 ***мс***, система является поддающейся планированию, поскольку 0,5 + 0,15 + 0,2 < 1. Даже при добавлении четвертого сигнала с периодом в 1***с*** системой все равно можно будет управлять при помощи планирования, пока время обработки сигнала не будет превышать 150 ***мс***. Эти расчеты не являются абсолютно верными, так как не учитывают время переключения контекста и не учитывают возникновение непериодических событий.

Алгоритмы планирования заданий могут быть разделены на статические и динамические. ***Статические*** алгоритмы определяют приемлемый план выполнения заданий по их априорным характеристикам, динамический алгоритм модифицирует план во время исполнения заданий. Издержки на статическое планирование низки, но оно крайне нечувствительно и требует полной предсказуемости той системы реального времени, на которой оно установлено. ***Динамическое*** планирование связано с большими издержками, но способно адаптироваться к меняющемуся окружению.

Алгоритмы планирования будем рассматривать на примере 3-х периодических процессов ***A, B, C***. Предположим, что процесс ***A*** запускается с периодом 30 ***мс*** и временем обработки 10 ***мс***. Процесс ***B*** имеет период 40 ***мс*** и время обработки 15 ***мс***. Процесс ***C*** запускается каждые 50 ***мс*** и обрабатывается за 5 ***мс***. Суммарно эти процессы потребляют 0,808 процессорного времени, что меньше единицы. Соответственно, система в данном примере поддается планированию.

На рис. 10.1 представлена временная диаграмма работы процессов. Видно, что необходимо применить некоторый алгоритм планирования, так как в определенные моменты времени имеется сразу несколько готовых к выполнению процессов.

На рисунке представлена *временная диаграмма* работы процессов.

Видно, что необходимо применить некоторый алгоритм планирования, так как в определенные моменты времени имеется сразу несколько готовых к выполнению процессов.

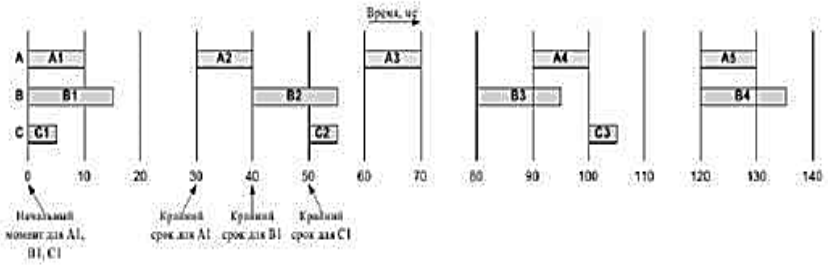


Рис. 10.1. Три периодических процесса с разным периодом и временем обработки

Статический алгоритм планирования *RMS*

Классическим примером статического алгоритма планирования реального времени для прерываемых периодических процессов является алгоритм ***RMS*** (Rate Monotonic Scheduling – планирование с приоритетом, пропорциональным частоте). Этот алгоритм может использоваться для процессов, удовлетворяющих следующим условиям:

1. Каждый периодический процесс должен быть завершен за время его периода.

2. Ни один процесс не должен зависеть от любого другого процесса.

3. Каждому процессу требуется одинаковое процессорное время на каждом интервале.

4. У непериодических процессов нет жестких сроков.

5. Прерывание процесса происходит мгновенно, без накладных расходов.

Алгоритм RMS работает, назначая каждому процессу фиксированный приоритет, обратно пропорциональный периоду и, соответственно, прямо пропорциональный частоте возникновения событий процесса. Например, в примере рис. 10.1 процесс ***A*** запускается каждые 30 мс (33 раза в секунду) и получает приоритет 33. Процесс ***B*** запускается каждые 40 мс (25 раз в секунду) и получает приоритет 25. Процесс ***C*** запускается каждые 50 мс (20 раз в секунду) и получает приоритет 20. Отметим, что реализация алгоритма требует, чтобы у всех процессов были разные приоритеты.

Во время работы планировщик всегда запускает готовый к работе процесс с наивысшим приоритетом, прерывая при необходимости работающий процесс с меньшим приоритетом. Таким образом, в нашем примере процесс ***A*** может прервать процессы ***B*** и ***C***, процесс ***B*** может прервать ***C***. Процесс ***C*** всегда вынужден ждать, пока процессор не освободится.

На рис. 10.2 показана работа алгоритма планирования для процессов ***A, B, C***.

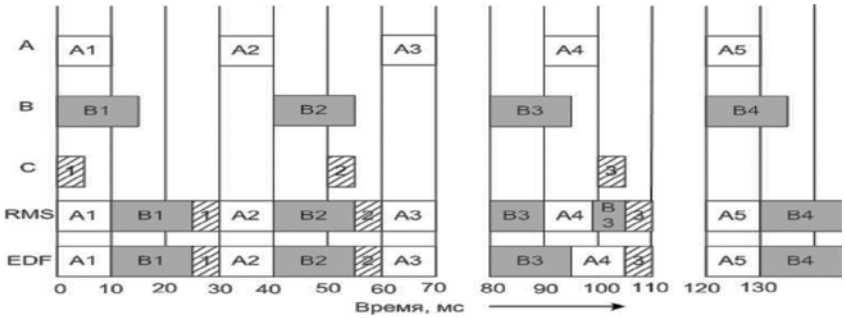


Рис. 10.2. Пример алгоритмов планирования RMS и EDF

Изначально все три процесса готовы к работе. Выбирается процесс с максимальным приоритетом – *A*. Ему разрешается работать в течение 10 мс, требующихся процессу до завершения.

Когда процесс ***A*** освобождает процессор, начинает работать процесс ***B***, а затем процесс ***C***. Вместе эти процессы потребляют 30 мс, поэтому, когда процесс ***C*** заканчивает работу, снова запускается процесс ***A***. Этот цикл повторяется до тех пор, пока в момент времени 70 мс у системы начинается период простоя. В момент времени 80 мс процесс ***B*** переходит в состояние готовности и запускается. Однако в момент времени 90 мс процесс ***A***, обладающий более высоким приоритетом, также переходит в состояние готовности. Поэтому он прерывает выполнение процесса ***B*** и работает до момента времени 100 мс, пока не закончит свою работу. В этот момент времени система должна выбрать между процессом ***B***, который не закончил обработку, и ***C***, который находится в состоянии готовности. Согласно алгоритму ***RMS***, выбирается процесс ***B***, имеющий больший приоритет.

Алгоритм ***RMS*** может быть использован только при не слишком высокой загруженности процессора. Предположим, что процесс ***A*** имеет продолжительность работы не 10 мс, а 15 мс. Коэффициент использования процессора в таком случае равен 0,975. Теоретически для данного случая должен иметься метод планирования. Работа алгоритма показана на рис. 10.3. На этот раз процесс ***B*** завершает обработку к моменту времени 30 мс. В этот же момент процесс ***A*** снова приходит в состояние готовности. К тому времени, когда он заканчивает работу, снова готов процесс ***B*** и поскольку у него приоритет больше, чем у ***C***, то запускается процесс ***B***. Процесс ***C*** пропускает свой критический срок, алгоритм ***RMS*** терпит неудачу. Теоретически было показано, что данный алгоритм гарантированно работает в любой системе периодических процессов при условии

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq m\left(2^{\frac{1}{m}} - 1\right). \qquad (10.2)$$

Таким образом, для 3-х процессов максимальная загруженность процессора равна 0,780. Хотя в нашем примере (рис.

10.2) загруженность процессора составила 0,808, алгоритм ***RMS*** еще работал, хотя с другими периодами и временем обработки при том же коэффициенте загруженности процессора мог потерпеть неудачу. При увеличении коэффициента загруженности на алгоритм RMS не было надежды.

Динамический алгоритм планирования *EDF*

Другим популярным алгоритмом планирования является алгоритм ***EDF*** (Earliest Deadline First – процесс с ближайшим сроком завершения в первую очередь). Алгоритм ***EDF*** представляет собой динамический алгоритм, не требующий от процессов периодичности. Он также не требует и постоянства временных интервалов использования процессора. Каждый раз, когда процессу требуется процессорное время, он объявляет о своем присутствии и о своем сроке выполнения задания. Планировщик хранит список процессов, сортированный по срокам выполнения заданий. Алгоритм запускает первый процесс в списке, то есть тот, у которого самый близкий по времени срок выполнения. Когда новый процесс переходит в состояние готовности, система сравнивает его срок выполнения со сроком выполнения текущего процесса. Если у нового процесса график более жесткий, он прерывает работу текущего процесса. Пример работы алгоритма EDF показан на рис. 10.2. Вначале все процессы находятся в состоянии готовности. Они запускаются в порядке своих крайних сроков. Процесс A должен быть выполнен к моменту времени 30, процесс B должен закончить работу к моменту времени 40, процесс C – 50. Таким образом, процесс A запускается первым. Вплоть до момента времени 90 выбор алгоритма EDF не отличается от RMS. В момент времени 90 процесс A переходит в состояние готовности с тем же крайним сроком выполнения 120, что и у процесса B. Планировщик имеет право выбрать любой из процессов, но поскольку с прерыванием процесса B не связано никаких накладных расходов, лучше предоставить возможность продолжать работу этому процессу.

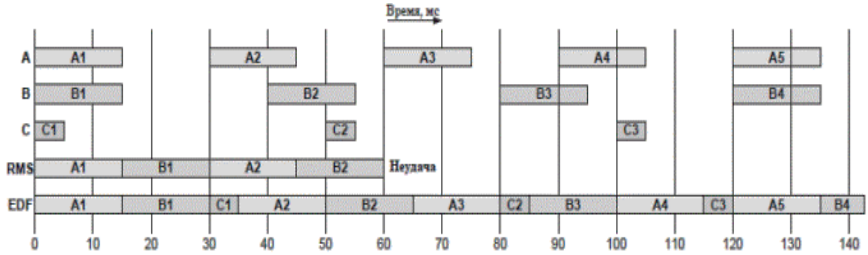


Рис. 10.3. Пример, в котором алгоритм RMS не может быть использован

Рассмотрим рис. 10.3. В момент времени 30 между процессами ***A*** и ***C*** возникает спор. Поскольку срок выполнения процесса ***C*** равен 50 мс, а процесса ***A*** – 60 мс, планировщик выбирает процесс ***C***. Этим данный алгоритм отличается от алгоритма ***RMS***, в котором побеждает процесс ***A***, как обладающий большим приоритетом. В момент времени 90 процесс ***A*** переходит в состояние готовности в 4-й раз. Предельный срок процесса ***A*** такой же, что и у текущего процесса, поэтому у планировщика появляется выбор – прервать работу процесса ***B*** или нет. Поскольку необходимости прерывать процесс ***B*** нет, то он продолжает работу. Алгоритм ***EDF*** работает с любым набором процессов, для которых возможно планирование. Платой за это является использование более сложного алгоритма.

§11. Синхронизация задач

Хотя каждая задача в системе, как правило, выполняет какую-либо отдельную функцию, часто возникает необходимость в согласованности (синхронизации) действий, выполняемых различными задачами. Такая синхронизация необходима в основном в следующих случаях.

1. Функции, выполняемые различными задачами, связаны друг с другом. Например, если одна задача подготавливает исходные данные для другой, то последняя не выполняется до тех

пор, пока не получит от первой задачи соответствующего сообщения. Одна из вариаций в этом случае – это когда задача при определенных условиях порождает одну или несколько новых задач.

2. Необходимо упорядочить доступ нескольких задач к разделяемому ресурсу.

3. Необходима синхронизация задачи с внешними событиями. Как правило, для этого используется механизм прерываний.

4. Необходима синхронизация задачи по времени. Диапазон различных вариантов в этом случае достаточно широк, от привязки момента выдачи какого-либо воздействия к точному астрономическому времени до простой задержки выполнения задачи на определенный интервал времени. Для решения этих вопросов в конечном счете используются специальные аппаратные средства, называемые таймером.

Давайте рассмотрим все четыре случая более подробно.

Связанные задачи. Взаимное согласование задач с помощью сообщений является одним из важнейших принципов операционных систем реального времени. Способы реализации межзадачного обмена отличаются большим разнообразием, что не в последнюю очередь приводит к обилию терминов в этой области. Можно встретить такие понятия, как сообщение (*message*), почтовый ящик (*mail box*), сигнал (*signal*), событие (*event*), прокси (*proxy*) и т. п. Если, читая описание какой-либо ОС РВ, вы встретите уже знакомое название, не спешите делать выводы. Даже один и тот же термин может для разных ОС РВ обозначать разные вещи. Чтобы не запутаться, мы будем в дальнейшем называть сообщениями любой механизм явной передачи информации от одной задачи к другой (такие объекты, как семафоры, можно отнести к механизму неявной передачи сообщений).

Объем информации, передаваемой в сообщениях, может меняться от 1 бита до всей свободной емкости памяти вашей системы. Во многих ОС РВ компоненты операционной системы, так же как и пользовательские задачи, способны принимать и

передавать сообщения. Сообщения могут быть асинхронными и синхронными. В первом случае доставка сообщений задаче производится после того, как она в плановом порядке получит управление, а во втором случае циркуляция сообщений оказывает непосредственное влияние на планирование задач. Например, задача, пославшая какое-либо сообщение, немедленно блокируется, если для продолжения работы ей необходимо дождаться ответа, или если низкоприоритетная задача шлет высокоприоритетной задаче сообщение, которого последняя ожидает, то высокоприоритетная задача, если, конечно, используется приоритетная многозадачность с вытеснением, немедленно получит управление.

Иногда сообщения передаются через отведенный для этого буфер определенного размера («почтовый ящик»). При этом, как правило, новое сообщение затирает старое, даже если последнее не было обработано.

Однако наиболее часто используется принцип, когда каждая задача имеет свою очередь сообщений, в конец которой ставится всякое вновь полученное сообщение. Стандартный принцип обработки очереди сообщений по принципу «первым вошел, первым вышел» (*FIFO*) не всегда оптимально соответствует поставленной задаче. В некоторых ОС РВ предусматривается такая возможность, когда сообщение от высокоприоритетной задачи обрабатывается в первую очередь (в этом случае говорят, что сообщение наследует приоритет пославшей его задачи).

Иногда полезным оказывается непосредственное управление приоритетом сообщений. Представим, что задача послала серверу (драйверу) принтера несколько сообщений, содержащих данные для печати. Если теперь задача хочет отменить всю печать, ей надо послать соответствующее сообщение с более высоким приоритетом, чтобы оно встало в очередь впереди всех посланных ранее заданий на печать.

Сообщение может содержать как сами данные, предназначенные для передачи, так и указатель на такие данные. В последнем случае обмен может производиться с помощью разделяемых областей памяти, разделяемых файлов и т. п.

Общие ресурсы. Трудно переоценить важность правильной организации взаимодействия различных задач при доступе к общим ресурсам. Хорошей аналогией может служить обед в многодетной крестьянской семье позапрошлого века. Едокам (задачам) не разрешалось одновременно лезть ложками в общую миску (ресурс). Нарушители порядка могли получить от отца семейства (супервизора) ложкой по лбу.

Ресурс – это общий термин, описывающий физическое устройство или область памяти, которые могут одновременно использоваться только одной задачей. Процессорное время тоже представляет собой своеобразный конкурентно используемый ресурс вычислительной системы. Примерам физических устройств могут служить клавиатура, дисплей, дисковый накопитель, принтер и т. п. Представим, например, что несколько задач пытаются одновременно выводить данные на принтер. На распечатке в результате ничего, кроме странной мешанины символов, мы не увидим. В качестве другого примера рассмотрим ситуацию, когда в бортовом компьютере мирно летящего самолета МИГ-29 среди прочих работают две задачи. Одна из них, взаимодействуя с радиолокационной системой, выдаст удаление и направление до цели, а другая задача использует эти данные для пуска ракет класса «воздух-воздух». Не исключено, что первая задача, записав в глобальную структуру данных удаление до цели, будет прервана второй задачей, не успев записать туда направление до цели. В результате вторая задача считает из этой структуры ошибочные данные, что может привести к неудачному пуску со всеми вытекающими отсюда неприятными последствиями. Прервись первая задача чуть позже, и все было бы нормально. Упомянутые здесь проблемы обусловлены *времязависимыми ошибками* (*time dependent error*), или «гонками» и характерны для многозадачных ОС, применяющих алгоритмы планирования с вытеснением (кстати, системы с разделением времени также относятся к категории «вытесняющих»).

Приведенный пример показывает, что ошибки, обусловленные «гонками», а) характерны для работы с любыми ресурсами, доступ к которым имеют несколько задач, и б) происходят только в результате совпадения определенных условий, а потому с трудом обнаруживаются на этапе отладки.

Вот возможные пути решения проблемы.

1. Не использовать алгоритмы планирования задач с вытеснением. Это решение, правда, не всегда приемлемо.

2. Использовать специальный сервер ресурса, то есть задачу, ответственную за упорядочивание доступа к ресурсу. В этом случае запрос на изменение значения глобальных данных посылается этому серверу в виде сообщения. Аналогичный подход применим и для физических устройств. Так, например, задача может послать данные на печать в виде сообщения, направленного к серверу принтера.

3. Запретить прерывания на время доступа к разделяемым данным. Кардинальное решение, которое, впрочем, не приветствуется в системах реального времени.

4. Использовать для упорядочивания доступа к глобальным данным семафоры. Наиболее часто применяемое решение, которое, впрочем, может привести в некоторых случаях к «инверсии приоритетов».

Последний пункт стоит прокомментировать подробнее, поскольку понятие «семафор» встречается первый раз.

Семафор – это как раз то средство, которое часто используется для синхронизации доступа к ресурсам. В простейшем случае семафор представляет собой байтовую переменную, принимающую значение 0 или 1. Задача, перед тем как использовать ресурс, захватывает семафор, после чего остальные задачи, желающие использовать тот же ресурс, должны ждать, пока семафор (ресурс) освободится. Существуют также так называемые счетные семафоры, где семафор представляет собой счетчик. Пусть к системе подключено три принтера. Семафор, отвечающий за

доступ к функциям печати, инициализируется со значением 5, а затем каждый раз, когда какая-либо задача запрашивает семафор для осуществления печати, его значение уменьшается на 1. После завершения печати задача освобождает семафор, в результате чего значение последнего увеличивается на 1. Если текущее значение семафора равно 0, то ресурс считается недоступным, и задачи, запрашивающие печать, должны ждать, пока не освободится хотя бы один принтер. Таким образом может производиться синхронизация доступа множества задач к группе из 3 принтеров. Так как по своей сути семафор также представляет собой глобальную переменную, все неприятности, которые упоминались ранее в связи с самолетом МИГ-29, по идее, должны поджидать нас и здесь. Однако, так как работа с семафорами происходит на уровне системных вызовов, программист может быть уверен, что разработчики операционной системы обо всем заранее позаботились.

Проникнувшись сознанием того, насколько опасно изменять глобальные переменные в условиях, когда все вокруг так и норовят друг друга вытеснить, наверно, не удивительно, что участки кода программ, где происходит обращение к разделяемым ресурсам, называются *критическими секциями***.**

Так как процессы обычно не имеют доступа к данным друг друга, а ресурсы физических устройств, как правило, управляются специальными задачами-серверами (драйверами), наиболее типична ситуация, когда «гонки» за доступ к глобальным переменным устраивают различные потоки, исполняемые в рамках одного программного модуля. Для того чтобы гарантировать, что критическая секция кода исполняется в каждый момент времени только одним потоком, используют механизм взаимоисключающего доступа, или попросту *мутексов (Mutual Exclusion Locks, Mutex*). Практически мутекс представляет собой разновидность семафора, который сигнализирует другим потокам, что критическая секция кода кем-то уже выполняется.

Критическая секция, использующая мутекс, должна иметь определенные суффиксную и префиксную части. Например:

```
int global_counter;
void main (void)
{
mutex_t mutex;
( /* И все это лишь для того, чтобы увеличить глобальную переменную на единицу.*/
mutexjnit (& mutex, USYNC, NULL);
mutex_lock (& mutex);
global_counter++;
mutex_unlock (& mutex);
}
```

Если мутекс захвачен, то поток, пытающийся войти в критическую секцию, блокируется. После того как мутекс освобождается, один из стоящих в очереди потоков (если таковые накопились) разблокируется и получает возможность доступа к глобальному ресурсу.

На этом рассмотрение средств синхронизации доступа к общим ресурсам можно закончить, хотя, разумеется, множество тем осталось за скобками. Например, в *WINDOWS* используется, в числе прочего, специальная разновидность мутексов под названием *Critical Section Object*. Необходимо также помнить, что, кроме ОС, имеющих *WINDOWS* или *POSIX API*. существует большое число ни с чем не совместимых ОС, поэтому наличие средств синхронизации и особенности их реализации должны рассматриваться отдельно для каждой конкретной ОС РВ.

А вот возможные неприятности в борьбе за ресурсы.

Смертельный захват (*Deadlock*). В народе побочные проявления этой ситуации называются более прозаично: «зацикливание» или «зависание». А причина этого может быть достаточно проста – задачи не поделили ресурсы. Пусть, например,

Задача *A* захватила ресурс клавиатуры и ждет, когда освободится ресурс дисплея, а в это время Задача *B* также хочет пообщаться с пользователем и, успев захватить ресурс дисплея, ждет теперь, когда освободится клавиатура. Так и будут задачи ждать друг друга до второго потопа. В таких случаях рекомендуется придерживаться тактики «или все, или и ничего». Другими словами, если задача не смогла получить все необходимые для дальнейшей работы ресурсы, она должна освободить всё, что уже захвачено, и, как говорится, «зайти через полчаса». Другим решением, которое уже упоминалось, является использование серверов ресурсов.

Инверсия приоритетов (*Priority inversion*). Как уже отмечалось, алгоритмы планирования задач (управление доступом к процессорному времени) должны находиться в соответствии с методами управления доступом к другим ресурсам, а все вместе – соответствовать критериям оптимального функционирования системы. Эффект инверсии приоритетов является следствием нарушения гармонии в этой области. Ситуация здесь похожа на «смертельный захват», однако сюжет закручен еще более лихо. Представим, что у нас есть высокоприоритетная Задача *A*, среднеприоритетная Задача *B* и низкоприоритетная Задача *C*. Пусть в начальный момент времени Задачи *A* и *B* блокированы в ожидании какого-либо внешнего события. Допустим, получившая в результате этого управления Задача *C* захватила Семафор *A*, но не успела она его отдать, как была прервана Задачей *A*. В свою очередь, Задача *A* при попытке захватить Семафор *A* будет блокирована, так как этот семафор уже захвачен Задачей *C*. Если к этому времени Задача *B* находится в состоянии готовности, то управление после этого получит именно она, как имеющая более высокий, чем у Задачи *C*, приоритет. Теперь Задача *B* может занимать процессорное время, пока ей не надоест, а мы получаем ситуацию, когда высокоприоритетная Задача *A* не может функционировать из-за того, что необходимый ей ресурс занят низкоприоритетной Задачей *C*.

Синхронизация с внешними событиями. Известно, что применение аппаратных прерываний является более эффективным методом взаимодействия с внешним миром, чем метод опроса. Разработчики систем реального времени стараются использовать этот факт в полной мере. При этом можно проследить следующие тенденции:

1. Стремление обеспечить максимально быструю и детерминированную реакцию системы на внешнее событие.

2. Старание добиться минимально возможных периодов времени, когда в системе запрещены прерывания.

3. Подпрограммы обработки прерываний выполняют минимальный объем функций за максимально короткое время. Это обусловлено несколькими причинами. Во-первых, не все ОС РВ обеспечивают возможность «вытеснения» во время обработки подпрограмм прерывания. Во-вторых, приоритеты аппаратных прерываний не всегда соответствуют приоритетам задач, с которыми они связаны. В-третьих, задержки с обработкой прерываний могут привести к потере данных.

Как правило, закончив элементарно необходимые действия, подпрограмма обработки прерываний генерирует в той или иной форме сообщение для задачи, с которой это прерывание связано, и немедленно возвращает управление. Если это сообщение перевело задачу в разряд готовых к исполнению, планировщик в зависимости от используемого алгоритма и приоритета задачи принимает решение о том, необходимо или нет немедленно передать управление получившей сообщение задаче. Разумеется, это всего лишь один из возможных сценариев, так как каждая ОС РВ имеет свои особенности при обработке прерываний. Кроме того, свою специфику может накладывать используемая аппаратная платформа.

Синхронизация по времени. Совсем не так давно (лет 20 назад) аппаратные средства, отвечающие в вычислительных системах за службу времени, были совершенно не развиты. В те

приснопамятные времена считалось достаточным, если в системе генерировалось прерывание с частотой сети переменного тока. Те же, кто не знал, что частота сети в США 60 Гц, а не 50, как у нас, постоянно удивлялись тому, что системное время в *RSX-11 M* никогда не бывает правильным. Программисты для получения задержек по времени часто использовали программные циклы ожидания и, разумеется, пользователи таких программ получали массу сюрпризов при попытке их переноса на следующее поколение компьютеров с более высокими тактовыми частотами. Благодаря научно-техническому прогрессу, сейчас любой мало-мальски приличный компьютер имеет часы/календарь с батарейной поддержкой и многофункциональный таймер (а то и несколько) с разрешением до единиц микросекунд.

Как правило, в ОС РВ задается эталонный интервал (квант) времени, который иногда называют тиком (*Tick*) и который используется в качестве базовой единицы измерения времени. Размерность этой единицы для разных ОС РВ может быть разной, как, впрочем, разными могут быть набор функций и механизмы взаимодействия с таймером. Функции по работе с таймером используют для приостановки выполнения задачи на какое-то время, для запуска задачи в определенное время, для относительной синхронизации нескольких задач по времени и т.п. Если в программе для ОС РВ вы увидите операнд *delay (50)*, то, скорее всего, это обозначает, что в этом месте задача должна прерваться (блокироваться), а через 50 мс возобновить свое выполнение, а точнее, перейти в состояние готовности. Все это время процессор не простаивает, а решает другие задачи, если таковые имеются. Множество задач одновременно могут запросить сервис таймера, поэтому если для каждого такого запроса используется элемент в таблице временных интервалов, то накладные расходы системы по обработке прерываний от аппаратного таймера растут пропорционально размерности этой таблицы и могут стать недопустимыми. Для решения этой проблемы можно вместо

таблицы использовать связный список и алгоритм так называемого дифференциального таймера, когда во время каждого тика уменьшается только один счетчик интервала времени.

Для точной синхронизации таймера вычислительной системы с астрономическим временем могут применяться специальные часы с подстройкой по радиосигналам точного времени или навигационные приемники *GPS*, которые позволяют воспользоваться атомными часами на борту орбитальных космических аппаратов, запущенных по программе *Navstar*.

Глава 2. ОБЗОР ОПЕРАЦИОННЫХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

До недавнего времени для решения задач автоматизации в основном использовались операционные системы (ОС) реального времени (РВ). Это компактные системы, основанные на микроядерной архитектуре, – такие как *VxWorks, OS*-9, *PSOS, QNX, LynxOS*. Эти системы обладают быстрым временем реакции на события и прерывания, компактностью кода, хорошей встраиваемостью и другими преимуществами, характерными для операционных систем с микроядерной архитектурой.

Кроме того, перечисленные системы уже много лет на рынке, имеют массу применений в ответственных проектах, что является дополнительным фактором доверия к ним. И, наконец, разработчик ответственной системы с повышенными требованиями к надежности, купив коммерческую ОСРВ, не остается наедине с возможными проблемами, так как может пользоваться услугами технической поддержки поставщика.

Конечно, без ОСРВ пока нельзя обойтись во многих применениях, и они оправдывают себя в высокотехнологичных проектах (тренажеры, уникальное оборудование) или при массовом количестве поставок (сотовые телефоны, анализаторы). В то же время существуют задачи, для решения которых требуется поддержка реального времени, но при этом большие затраты на разработку не окупятся.

Наверное, классические операционные системы реального времени остаются самым надежным решением при построении систем жесткого реального времени повышенной надежности. Однако систем с такими требованиями не так много в мире, который нас окружает. Возможны ли другие решения?

В настоящее время происходит активный процесс слияния универсальных ОС и ОС реального времени. На программном рынке появляются различные инструменты поддержки режима реального времени, встраиваемые в привычные операционные

системы. Этот класс продуктов обладает значительными преимуществами со стороны пользователей и программистов, сочетая в себе привычный пользовательский интерфейс, средства разработки программ и *API*-интерфейс реального времени. Правда, пока еще нет оснований утверждать, что подобные решения полностью заменят собой обычные системы реального времени. У традиционных ОС реального времени есть пока большое преимущество – встраиваемость и компактность. Однако и на этом рынке происходят изменения: корпорация *Microsoft* выпустила *Windows NT Embedded* (*NTE*), использование которой позволяет "ужать" *NT* до 8-10 Мбайт. Уже появились и продукты реального времени, рассчитанные на эту операционную систему, например, *RTX*.

А что же *Unix*? Традиционно во многих ОС реального времени используются подмножества *API*-интерфейсов *Unix*, что сближает эти операционные системы с *Unix*. Существуют также полнофункциональные *Unix*-системы, поддерживающие режим реального времени.

§1. *Linux* реального времени

Бурный рост популярности *Linux* побуждает разработчиков внимательнее присмотреться к этой операционной системе [5,6]. У *Linux* много достоинств: открытость кода; большое количество сопутствующего программного обеспечения, пока в основном ориентированного на серверные применения; наличие неплохой документации на *API*-интерфейс и ядро операционной системы; работа на процессорах различных классов. В данный момент эта ОС готова к стабильной работе, а открытость ее исходных текстов и архитектуры наряду с растущей популярностью заставляет программистов переносить свои наработки на многие аппаратные платформы: *SGI*, *IBM*, *Intel*, *Motorol*а и т.д. В частности, *Motorola* активно работает в своей традиционной сфере встраиваемых систем и продвигает на рынок продукт *LinuxEmbedded*. Вообще говоря,

Linux прекрасно подходит для компактных встроенных применений; на рынке уже появились поставщики, предлагающие усеченные варианты этой операционной системы, которые занимают 1-2 Мбайт на жестком диске. В качестве примера можно привести проект *Linux Router Project*.

Для задач реального времени сообщество разработчиков *Linux* активно применяет специальные расширения – *RTLinux*, *KURT* и *UTIME*, позволяющие получить устойчивую среду реального времени. *RTLinux* представляет собой систему "жесткого" реального времени, а *KURT* (*KU Real Time Linux*) относится к системам "мягкого" реального времени. *Linux*-расширение *UTIME*, входящее в состав *KURT*, позволяет добиться увеличения частоты системных часов, что приводит к более быстрому переключению контекста задач.

Проект *KURT* характеризуется минимальными изменениями ядра *Linux* и предусматривает два режима работы – нормальный (*normal mode*) и режим реального времени (*real-time mode*). Процесс, использующий библиотеку *API*-интерфейсов *KURT*, в любые моменты времени может переключаться между этими двумя режимами. Программный пакет *KURT* оформлен в виде отдельного системного модуля *Linux – RTMod*, который становится дополнительным планировщиком реального времени. Данный планировщик доступен в нескольких вариантах и может тактироваться от любого системного таймера или от прерываний стандартного параллельного порта. Так как все процессы работают в общем пространстве процессов *Linux*, программист использует в своих программах стандартные *API*-интерфейсы *Linux* и может переключаться из одного режима в другой по событиям либо в определенных местах программы. При переключении в режим реального времени все процессы в системе "засыпают" до момента освобождения ветви процесса реального времени. Это довольно удобно при реализации задач с большим объемом вычислений, по своей сути требующих механизмов реального времени, например в задачах обработки аудио- и видеоинформации.

Стандартно такты планировщика *RTMod* задаются от системного таймера – время переключения контекста задач реального времени (*time slice*) равно 10 мс. Используя же *KURT* совместно с *UTIME*, можно довести время переключения контекста задач до 1 мс. Прерывания обрабатываются стандартным для ОС *Linux* образом – через механизм драйверов.

API-интерфейс *KURT* разделяется на две части – прикладную и системную. Прикладная часть позволяет программисту управлять поведением процессов, а системный *API*-интерфейс предназначен для манипулирования пользовательскими процессами и программирования собственных планировщиков. Прикладная часть *API*-интерфейса *KURT* состоит всего из четырех функций:

set_rtparams позволяет добавить процесс в ядро с маской *SCHED_KURT*. Только процессы, чья политика в планировщике установлена как *SCHED_KURT*, смогут работать в режиме реального времени;

get_num_rtprocs получает идентификатор "*rt_id*" процесса из планировщика реального времени *RTMod*;

rt_suspend позволяет приостановить планировщик реального времени;

get_rt_stats получает статус процесса из таблицы планировщика процессов реального времени.

Простота использования *KURT* позволяет с максимальным комфортом программировать задачи, требующие как функций реального времени, так и всего многообразия *API*-интерфейса *Unix*. Использование "мягкого" реального времени обычно подходит для реализации мультимедийных задач, а также при обработке разного рода потоков информации, где критично время получения результата. Совершенно другой подход применен при реализации в *Linux* "жесткого" реального времени.

RTLinux – это дополнение к ядру *Linux*, реализующее режим "жесткого" реального времени, которое позволяет управлять различными чувствительными ко времени реакции системы

процессами. По сути дела, *RTLinux* – это операционная система, в которой маленькое ядро реального времени сосуществует со стандартным *POSIX*-ядром *Linux*.

Разработчики *RTLinux* пошли по пути запуска из ядра реального времени *Linux*-ядра как задачи с наименьшим приоритетом. В *RTLinux* все прерывания обрабатываются ядром реального времени, а в случае отсутствия обработчика реального времени – передаются *Linux*-ядру. Фактически *Linux*-ядро является простаивающей (*idle*) задачей операционной системы реального времени, запускаемой только в том случае, если никакая задача реального времени не исполняется. При этом на *Linux*-задачу накладываются определенные ограничения, которые, впрочем, для программиста прозрачны.

Нельзя выполнять следующие операции: блокировать аппаратные прерывания и предохранять себя от вытеснения другой задачей. Ключ к реализации данной системы – драйвер, эмулирующий систему управления прерываниями, к которому обращается *Linux* при попытке блокировать прерывания. В этом случае драйвер перехватывает запрос, сохраняет его и возвращает управление *Linux*-ядру.

Все аппаратные прерывания перехватываются ядром операционной системы реального времени. Когда происходит прерывание, ядро *RTLinux* решает, что делать. Если это прерывание имеет отношение к задаче реального времени, ядро вызывает соответствующий обработчик. В противном случае, либо если обработчик реального времени говорит, что хочет разделять это прерывание с *Linux,* обработчику присваивается состояние ожидания (*pending*). Если *Linux* потребовала разрешить прерывания, то прерывания, которые находятся в состоянии "*pending*", эмулируются. Ядро *RTLinux* спроектировано таким образом, что ядру реального времени никогда не приходится ожидать освобождения ресурса, занятого *Linux*-процессом (рис. 1.1).

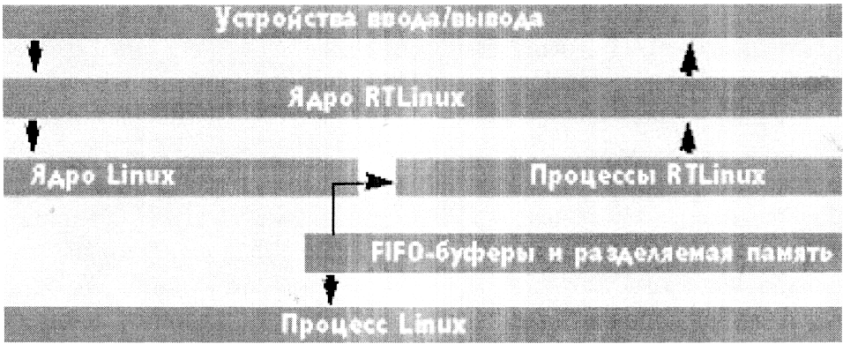


Рис. 1.1. Механизм работы *RTLinux* – *Linux*-расширения жесткого реального времени

Для обмена данными между операционными системами реального времени и *Linux* могут использоваться следующие механизмы:

– разделяемые области памяти;

– псевдоустройства, предоставляющие возможность обмена данными с приложениями реального времени.

Ключевой принцип построения *RTLinux* – как можно больше использовать *Linux* и как можно меньше собственно *RTLinux*. Действительно, именно *Linux* заботится об инициализации системы и устройств, а также о динамическом выделении ресурсов. "На плечи" *RTLinux* ложится только планирование задач реального времени и обработка прерываний. Для простоты запуска в контексте ядра, сохранения модульности и расширяемости системы процессы реального времени реализованы в виде загружаемых модулей *Linux*. Как правило, приложение реального времени с *RTLinux* состоит из двух независимых частей: процесса, исполняемого ядром *RTLinux*, и обыкновенного *Linux*-приложения.

Для иллюстрации работы приложения реального времени рассмотрим прикладной модуль, который использует ядро реального времени *RTLinux* в виде загружаемого модуля *Linux*:

```
 #define MODULE
#include <linux/module.h>
/* необходимо для задач
реального времени */
#include <linux/rt_sched.h>
#inlcude <linux/rt_fifo.h>
RT_TASK mytask;
```

В заголовке модуля определяется структура *RT_TASK*, которая содержит указатели на структуры данных модуля, а также управляющую информацию. В нашем случае для связи между процессами используются очереди сообщений *FIFO* (рис. 1.2).

Модуль реального времени читает данные из устройства и кладет их в очередь сообщений, откуда их потом забирает обыкновенное приложение *Linux.*

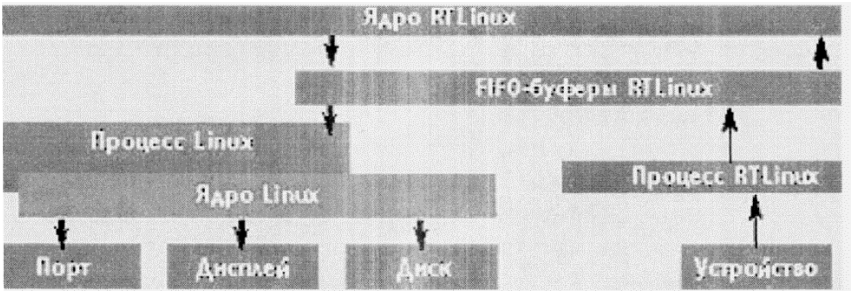


Рис. 1.2. Механизм межпроцессной связи через очереди сообщений *FIFO*

Как и каждый модуль *Linux*-ядра, процесс реального времени должен выполнить процедуры инициализации и завершения, аналогичные модулям *Linux*:

```
int init_module(void)
{
/* определить номер
очереди сообщений */
    #define RTFIFODESC 1
```

```
/* взять локальное время */
RTIME now = rt_get_time()
rt_task_init(&mytask, mainloop,
RTFIFODESC,3000, 4);
rt_task_make_periodic(&mytask,
now+1000, 25000);
return 0;
}
```

Подобный модульный подход к написанию приложений присущ многим расширениям реального времени для универсальных систем, когда задача реального времени работает независимо от ОС. Разработчики уже приняли схему, по которой задачи, критичные к времени реакции, программируются с помощью *API*-интерфейсов, предусмотренных расширением реального времени, а все служебные и интерфейсные функции возлагаются на традиционную операционную систему. Логично предположить, что при использовании данного подхода разработчику потребуется изучить только *API*-интерфейс реального времени.

Оба подхода в реализации механизмов реального времени, заложенные в *KURT* и *RTLinux*, применимы для большинства задач реального времени. Многие разработчики уже приняли *Linux* и внедряют ее в своих коммерческих проектах как основную операционную систему для серверов приложений. Однако до сих пор при реализации вертикальных проектов на нижнем уровне применялись специализированные ОС реального времени. Ситуация может существенно измениться благодаря использованию расширений реального времени для *Linux*.

§2. Операционные системы реального времени и *Windows*

Сегодня становится широко распространенным желание потребителей использовать *Windows NT* в системах реального

времени [9]. Для этого имеется ряд весомых, на первый взгляд, причин: *Win*32 *API* считается стандартом, а на его базе разработано огромное количество программ; графический интерфейс стал сегодня очень популярным; для *NT* имеется немало готовых решений для коммерческих применений; в среду *NT* включены многие виды средств разработки. Тем не менее возможно ли использование *Windows* для разработки системы реального времени?

Перечислим необходимые требования к ОС для обеспечения предсказуемости.

1. ОС РВ должна быть многонитевой и допускать вытеснение (*preemtible*).

Предсказуемость достигается, если в ОС допускается много параллельных потоков управления (нитей), а диспетчер ОС может прервать выполнение любой нити (вытеснить ее) в системе и предоставить ресурсы той нити, которой они требуются в первую очередь. ОС и аппаратная архитектура также должны предоставлять множество уровней прерываний, чтобы вытеснение было возможно и на уровне прерываний.

2. Диспетчеризация должна осуществляться на базе приоритетов.

Основная сложность диспетчеризации заключается в том, чтобы обнаружить, какая именно нить нуждается в ресурсах в первую очередь. В идеале ОС РВ предоставляет ресурсы той нити или драйверу, которым осталось меньше всего времени до установленного срока. Чтобы сделать это, ОС должна знать, когда нить обязана завершить свою работу и сколько времени ей понадобится. Поскольку это очень трудно реализовать, таких ОС пока еще не существует. Поэтому механизм диспетчеризации потоков управления в современных ОС базируется на понятии приоритета: ресурсы предоставляются нити с наивысшим приоритетом.

3. Механизм синхронизации нитей должен быть предсказуемым.

Механизм захвата ресурсов и межнитевых связей необходим, поскольку нити разделяют общие ресурсы.

4. Должна существовать система наследования приоритетов.

Разделение нитями с разными приоритетами общих ресурсов может привести к классической проблеме инверсии приоритетов. Чтобы избежать этого, ОС РВ должна допускать "наследование" приоритета, подталкивая нить с низшим приоритетом. Наследование приоритета означает, что блокирующая нить наследует приоритет нити, которую она блокирует (конечно, только если последняя обладает более высоким приоритетом).

5. Временные характеристики ОС должны быть предсказуемы и известны.

Разработчик СРВ должен знать, сколько времени затрачивается на ту или иную системную работу. Кроме того, должны быть известны уровни системных прерываний и уровни *IRQ* (линий запросов прерываний) драйверов устройств, максимальное время, которое они затрачивают, и т.п.

Несмотря на то что сегодня *Windows NT* не отвечает в полной мере требованиям, предъявляемым к операционной системе реального времени, давление рынка привело к появлению коммерческих решений, расширяющих *NT* возможностями обработки в реальном времени.

Удовлетворяет ли *Windows NT* требованиям, предъявляемым к ОС РВ?

Очевидно, что *NT* – многонитевая ОС, она позволяет вытеснение и тем самым удовлетворяет требованию 1.

В *Windows NT* имеются два класса приоритетов: класс реального времени и динамический класс. Процессы в классе реального времени имеют фиксированный приоритет, менять который может лишь само приложение, тогда как приоритет процессов динамического класса может меняться диспетчером. Процесс имеет базовый уровень приоритета. Нить в процессе может иметь приоритет в диапазоне плюс/минус 2 около базового уровня или один из двух крайних уровней класса (16 или 31 для реального

времени). Например, нить в процессе с базовым уровнем 24 может иметь приоритет 16, 22 – 26, 31. Очевидно, что гарантировать предсказуемость системы можно только при использовании процессов первого класса.

Казалось бы, второе требование также удовлетворено. Но малое число возможных уровней препятствует реализации СРВ на базе *NT*. Большинство современных ОС РВ позволяет иметь, по крайней мере, 256 различных уровней. Чем больше имеется уровней, тем более предсказуемо поведение системы. В *Windows NT* имеется только 7 различных уровней для нити в данном процессе. В результате многие нити будут выполняться с одинаковыми приоритетами и, как следствие, предсказуемость поведения системы будет посредственной. Более того, общее число уровней для всех процессов класса только 16 и положение не спасает даже замена нитей процессами, не говоря уже о том, что переключение контекста для процессов снижает производительность.

В ОС РВ вызовы системы синхронизации (семафоры или критические секции) должны уметь управлять наследованием приоритетов. В *Windows NT* это невозможно, поэтому требование 4 не удовлетворяется.

Еще одна проблема обусловлена реализацией некоторых вызовов системы *GUI*. Они обрабатываются синхронно (вызывающая нить приостанавливается, пока не завершится системный вызов) процессом, выполняемым с более низким приоритетом (динамического класса). В результате нить может помешать нити с более высоким приоритетом – возникает инверсия приоритета.

Таким образом, малое число приоритетов и невозможность решить проблему инверсии делают *Windows NT* пригодной только для очень простых СРВ.

Предсказуемость системных вызовов Win32 API.

Для тестирования системных вызовов был написан процесс (принадлежащий классу реального времени), содержащий вызовы системы синхронизации нитей, и измерялось время, затраченное на

каждый вызов. Максимальное значение на вызове *mutex* достигает 670 мкс при среднем времени 35 мкс. Это произошло потому, что во время работы теста происходили обращения к диску и по сети. Если компьютер искусственно загрузить обращениями к диску и сетевой обработкой, то задержки возрастают до нескольких миллисекунд. *Win*32 *API* очень богат, но не предназначен для реального времени. Например, запросы *mutex* обрабатываются в порядке поступления, а не в порядке приоритетов, что снижает предсказуемость. Для синхронизации нитей в одном процессе критические секции следует предпочесть всем другим способам (этот вызов затрачивает всего несколько мкс по сравнению с 35 мкс на вызов *mutex*).

Несмотря на все достоинства *API,* ядро и менеджер ввода/вывода *Windows NT* недостаточно приспособлены к обработке событий реального времени на уровне приложений.

Управление прерываниями в NT.

В *Windows NT* единственный способ управления аппаратурой – через драйвер устройства. Поскольку приложение реального времени имеет дело с внешними событиями, разработчик должен написать и включить в ядро драйвер устройства, дающий доступ к аппаратуре. Этот драйвер реагирует на прерывания, генерируемые соответствующим устройством.

Прерывания обрабатываются в два этапа. Сначала выполняется очень короткая программа обслуживания прерываний (*ISR*). Впоследствии работа завершается выполнением *DPC* – процедуры отложенного вызова. Возникает следующий поток событий:

– Происходит прерывание.

– Процессор сохраняет *PC, SP* и вызывает диспетчер.

– ОС сохраняет контекст и вызывает *ISR*.

– В *ISR* выполняется критическая работа (чтение/запись аппаратных регистров).

– *DPC* ставится в очередь.

– ОС восстанавливает контекст.

– Процессор восстанавливает *PC, SP*.

– Ожидающие в очереди *DPC* выполняются на уровне приоритета *DISPATCH_LEVEL*.

– После завершения всех *DPC* ОС переходит к выполнению приложений.

ISR должно быть как можно короче, поэтому большинство драйверов выполняют значительную часть работы в *DPC* (которая может быть вытеснена другим *ISR*), ожидая окончания других *DPC*, ранее попавших в очередь (все *DPC* имеют одинаковый приоритет).

Из документации по *NT* следует, что *ISR* может быть вытеснена другим *ISR* с более высоким приоритетом и что *DPC* имеет более высокий приоритет, чем пользовательские и системные нити. Но поскольку все *DPC* имеют одинаковый уровень приоритета и *ISR* должна быть сведена к минимуму, ваш *DPC* будет вынужден ждать других и ваше приложение будет зависеть от остальных драйверов устройств непредсказуемым образом. Задержки системных вызовов, описанные в предыдущем разделе, обусловлены именно тем, что *DPC* от драйверов жесткого диска и сети блокируют все другие.

В настоящих ОС РВ разработчик знает, на каком уровне выполняются драйверы всех других устройств. Обычно имеется свободное пространство для прерываний выше стандартных драйверов. Вся критическая работа выполняется в *ISR*, что позволяет настроить конфигурацию драйвера в зависимости от временных ограничений приложения.

Управление памятью в NT

Другим важным моментом при проектировании СРВ является политика управления памятью в ОС РВ. В *Windows NT* процессы выполняются в своем собственном пространстве памяти. Добиться этого позволяют механизмы виртуальной памяти и подкачки. Для бизнес-приложений это хорошо, но для СРВ, которая должна реагировать на внешние события с заранее определенным лимитом времени, это порождает непредсказуемость, особенно если система отправит страницу из памяти на диск. *Windows NT* позволяет

захватить страницу в памяти посредством вызова функции *VirtualLock*. Тем не менее *NT* может разблокировать страницу и выгрузить ее на диск, если весь процесс неактивен.

Может ли *Windows NT* использоваться в качестве ОС РВ?

Итак, можно сделать вывод, что *Windows NT*, предназначенная в основном для классических приложений, не является хорошей платформой для поддержания обработки в реальном времени. Тем не менее на ее базе можно все-таки построить простую мягкую СРВ, время от времени допускающую опоздания. Следующие обстоятельства могут облегчить построение СРВ на базе *NT*:

– загрузка *CPU* низка (*DPC* имеют достаточно времени);

– критическая работа (или даже вся) делается на уровне *DPC* или (еще лучше) на уровне *ISR*. В таком случае непонятно, зачем вообще нужна ОС.

Но для жесткой СРВ использование *Windows NT* невозможно – система реального времени никогда не будет предсказуемой. Что следует изменить в *Windows NT*, чтобы ее можно было использовать в жестких СРВ?

а) Класс процессов реального времени должен иметь больше уровней.

б) Необходимо решить проблему инверсии приоритетов.

в) Время, затрачиваемое каждым системным вызовом, должно быть предсказуемо и известно.

г) Система прерываний должна быть заменена целиком:

– *DPC* должны иметь много уровней приоритетов;

– *DPC* должны быть вытесняемыми более приоритетными *DPC*;

– драйверы от третьих фирм и системные драйверы должны быть настраиваемыми (уровни прерываний *ISR*, уровни прерываний *DPC*);

– драйверы от третьих фирм должны быть открыты для разработчиков, должно быть известно, по крайней мере, максимальное время, затрачиваемое на работу *ISR* и *DPC*;

– должно быть известно время маскирования прерываний.

Коммерческие решения, расширяющие *NT* возможностями обработки в реальном времени.

Существуют разные варианты использования технологии *NT* для разработки систем реального времени:

– использование *NT* как она есть для построения мягкой системы реального времени;

– реализация *Win*32 *API* над другой ОС РВ;

– совместная работа на одном процессоре *NT* и другой ОС РВ (или ее части);

– использование мультипроцессорной архитектуры, когда *NT* выполняется на одном процессоре (или более), а часть реального времени – на остальных.

Во многих решениях производители модифицируют *HAL* или ядро *NT*. Политика *Microsoft* заключается в том, чтобы не допускать никаких модификаций ядра *NT*, кроме драйверов устройств. Это единственно возможный способ связи с ядром. Политика компании относительно *HAL* другая. *HAL* (*Hardware Abstraction Layer*) – уровень аппаратных абстракций – уровень, лежащий ниже программного обеспечения, который виртуализирует интерфейс *NT* с аппаратурой, допуская переносимость *NT* с одной аппаратной платформы на другую. Такие модификации *HAL*, как манипуляции с часами или замена методов обработки прерываний, представляются беспримерно незаконным использованием *HAL*. Они создают нестандартную среду и могут привести к проблемам сопровождения, если, например, *Microsoft* изменит *HAL* в следующих версиях. Поэтому различие в решениях, предлагаемых поставщиками, заключается в попытках сделать модификации *HAL* минимальными.

Также возможен перехват *HAL* посредством трюков с процессором *Intel*. Однако это можно реализовать только на платформе *Intel*. Механизмы перехвата посредством обработки исключительных ситуаций на уровне устройства поглощают определенную вычислительную мощность.

Использование NT как таковой

Подобное применение предполагает включение структур, обрабатывающих прерывания. Однако учитывая, что *NT* не предназначена для обработки в реальном времени, это надо делать очень аккуратно.

Иногда вводят усовершенствования в механизм обработки прерываний. Единственный способ сделать это – перехватить прерывание, для чего необходимо добавить специальное аппаратное расширение. *LP-Elektronik,* например, перехватывает прерывание и использует затем *NMI* (немаскируемое прерывание, не используемое на уровне *NT*) для обработки событий реального времени. Этот подход применим только тогда, когда процессор имеет отдельный стек прерываний. Программа *NMI* должна быть написана аккуратно: в ней нельзя использовать вызовы ОС и она должна быть как можно короче, чтобы не потерять другие прерывания. Такое решение дает минимальную задержку прерывания, но требует дополнительной аппаратуры. Как и в других решениях, здесь необходим дополнительный механизм связи между *NT* и частью реального времени. Разница в том, что при этом требуется большая аккуратность в использовании *NMI.*

Реализация Win32 API над другой ОС РВ.

Добавление *Win*32 *API* к ОС, предназначенной для обработки в реальном времени, избавляет от необходимости модифицировать *HAL* или использовать другие трюки с *NT*.

Преимущества такого подхода:

– имеется переносимость;

– небольшой след;

– поведение ОС РВ известно.

Недостатки:

– нельзя использовать стандартные приложения *NT*;

– невозможно смешивание с драйверами устройств *NT*, поэтому весь мир коммуникаций *NT* недоступен;

– другие драйверы графических устройств и приложения *NT* невозможно или трудно использовать;

– ограничена возможность расширения в будущем;

– необходимо использовать специальные средства разработки и компиляторы.

Среди коммерческих реализаций этого подхода – *QNX* и *VxWorks*, использующие библиотеку *Windows*.

Совместная работа на одном процессоре NT и ОС РВ

Мощность современных процессоров достаточна для одновременного функционирования *NT* и программ реального времени. Возможны две разновидности такого подхода:

– модификация *HAL* с перехватом прерываний и включением небольшого диспетчера или ОС РВ;

– выполнение *NT* как одной из задач над (супервизором) ОС РВ.

В любом случае *HAL* должен быть модифицирован или, по крайней мере, перехвачен. В основном все такие решения похожи. В качестве иллюстрации можно привести следующее возможное решение с модификацией HAL.

Программу голубого экрана *NT* можно рассматривать как упрощенную программу супервизора. Модифицируя ее внутри *HAL*, можно сделать простой мультизадачный механизм выхода из нее, запустить *NT* как одну из задач с самым низким приоритетом и запустить нечто другое как другую задачу. Это нечто другое может быть набором задач реального времени или полной ОС РВ.

В обоих случаях необходим механизм связи между частями реального времени и *NT*. Он может быть выполнен одним из двух способов. Первый заключается во введении альтернативного механизма межпроцессорных связей (*IPC*). Проще всего реализовать *IPC* с помощью разделяемой памяти. Недостатком такого протокола *IPC* является уровень приоритета, на котором выполняются пользовательские приложения *NT*. При втором способе интерфейс реализуется с помощью драйвера устройства, в результате чего у *NT* создается впечатление, что подсистема реального времени – это устройство.

Задачи реального времени используют собственный интерфейс с системой, в большинстве случаев отличный от *Win*32 *API*. Среда разработки может быть обычной для используемой ОС РВ средой и может взаимодействовать со средой *NT*. Задачи реального времени будут выполняться, не получая преимуществ от механизма защиты памяти *NT*. Особо аккуратно следует продолжать выполнение частей реального времени, когда *NT* рухнет и сгенерирует голубой экран. След памяти – это комбинация следа *NT* (8 Мбайт в стандартной конфигурации) плюс минимальные требования для части ОС РВ.

Простота части реального времени может привести к высокой производительности, которая зависит от используемого ядра реального времени. Преимущества здесь таковы:

– сохранение (почти) всей среды *NT* нетронутой означает, что все программное обеспечение, устройства и драйверы устройств *NT* можно использовать (чтобы выполнять части приложения, не связанные с реальным временем). Поддерживается совместимость с *NT*;

– можно включить защиту для задач реального времени, зависящую от используемой ОС РВ.

Недостатки:

– отсутствует переносимость между реальным временем и средой *NT*, за исключением случая, когда *RT-API* разработан на базе *Win*32;

– драйверы устройств *NT* нельзя использовать в части реального времени;

– среда разработки усложняется, если для задач реального времени требуется отдельная среда;

– может быть много уровней задач и поэтому много уровней определений контекста. Переключение этих контекстов требует времени.

Известны следующие коммерческие реализации подхода, не требующего модификации аппаратуры: *IMAGINATION* с *HyperKERNEL*; *RADISYS* с комбинацией *iRMX/NT*; *VenturCom* с *RTX, KPX* и *RTAPI*.

В реализации фирмы *RadiSys* ОС РВ *iRMX* работает, как первичная ОС, загружающая *Windows NT* в качестве низкоприоритетной задачи. Пользователь работает только с *NT*, не видя и не чувствуя *iRMX*. Все управляющие функции выполняются как высокоприоритетные задачи *iRMX,* изолированные в памяти от приложений и драйверов *NT*. Используя функции защиты памяти внутри процессора *Intel, Windows NT* защищена от задач реального времени.

Комбинация *iRMX/NT* преодолела трудности, которые возникают при модификации *HAL* и оставляют пользователя уязвимым при сбоях жесткого диска, сбоях драйверов и системных сбоях *NT* ("голубой экран смерти"). В этом решении управляющая программа в случае краха *NT* может либо продолжить нормальное выполнение, либо произвести правильное закрытие системы (*shutdown*).

Реализация фирмы *VenturCom* состоит из двух этапов. На первом – мягкая реализация *RTX* 3.1 содержит интерфейс прикладной программы реального времени *RTAPI*, который дает время реакции 1-5 мкс. *RTAPI* 1.0 работает со стандартным *NT*. Единственное изменение, обеспечивающее лучшую синхронизацию событий реального времени, внесено в часы. Так как в *Windows NT* имеются некоторые плохо предсказуемые процессы, то *RTAPI* позволит построить только мягкую СРВ с небольшим временем отклика, но недостаточно предсказуемую. Впрочем, большую часть непредсказуемости *NT* можно устранить, ограничив доступ к системному диску и сети.

Чтобы сделать *NT* более предсказуемой, необходимо прерывать ее внутренние задачи. В основе второй жесткой реализации *RTX* 4.1 лежит модификация *HAL*. В обеспечении детерминизма важную роль играют программируемые часы. В каждый тик часов – аппаратное прерывание с регулярными интервалами времени – предпочтение отдается задаче реального времени. Оставшееся время предоставляется процессам *NT*, в том числе и процессам мягкого реального времени. Чем чаще тикают

часы, тем больше возможностей у процессора для выполнения задач реального времени. Необходимо добиться баланса между многими факторами: частота тиков, время, выделенное для обработки в реальном времени, время, выделенное для выполнения задач *NT*.

Использование многопроцессорной архитектуры.

Простое решение здесь состоит в том, что *NT* выполняется на одной группе процессоров, а часть реального времени – на другой. Возможно применение архитектур параллельной шины с *VMEbus*, *PCI*, *PMC* или архитектур последовательной шины с *Ethernet*. Если необходимо, подсистемы могут быть связаны механизмом *IPC* и процедурами удаленного вызова. Преимущества такого решения:

– нет модификаций каждой ОС;

– можно применять в больших сложных системах;

– для каждой подсистемы можно выбрать свою, наиболее подходящую ОС РВ;

– можно использовать имеющиеся среды разработки.

Недостатки:

– высокая стоимость;

– поведение в реальном времени зависит от поведения канала межпроцессорной связи. Поведение прерываний на шине в таких структурах предсказуемо лишь при хорошей организации;

– среды разработки относятся к разным мирам.

§3. Операционная система *QNX*

В последнее время система *QNX* быстро развивалась, превращаясь из узкоспециализированной ОС для систем реального времени в более-менее универсальную систему [1,2]. Однако ее разработчики отказались от попыток "добавить еще одну функцию" в *QNX*, поскольку этот путь ведет в тупик.

QNX относится к категории *Unix*-подобных систем, несмотря на имеющиеся фундаментальные различия. Поэтому лучший способ понять особенности каждой системы – сравнить их.

С точки зрения пользовательского интерфейса и *API* эта ОС очень похожа на *Unix*. Если вы знакомы с *Unix* на уровне пользователя, то, вероятно, сможете работать с *QNX* без проблем – в ней присутствует практически весь набор стандартных утилит и сохраняется большая часть семантики. Конечно, есть и *X Window*, равно как и *TCP/IP*. Если вы программист, знающий *Unix*, то для вас не составит большого труда перенести собственные или *GNU/Free*-приложения в *QNX*. Например, *Apache* и *Mosaic* хорошо демонстрируют степень совместимости *API*.

Однако *QNX* – это не версия *Unix*. Она была разработана с нуля и построена на совершенно других архитектурных принципах, но с учетом группы стандартов *POSIX*, которые возникли в результате обобщения существующей практики в различных версиях системы *Unix*. Разработка ведется канадской фирмой *QNX Software Systems Limited* (далее – *QSSL*).

QNX – это первая коммерческая ОС, построенная на принципах микроядра и обмена сообщениями. Система реализована в виде совокупности независимых (но взаимодействующих через обмен сообщениями) процессов различного уровня (менеджеры и драйверы), каждый из которых реализует определенный вид сервиса. Эти идеи обеспечили ряд важнейших преимуществ.

Предсказуемость, означающую ее применимость к задачам жесткого реального времени. Ни одна версия *Unix* не может достичь подобного качества, поскольку нереентрабельный код ядра слишком велик. Любой системный вызов в *Unix* может привести к непредсказуемой задержке. То же самое относится к *Windows NT*, где реальное время заканчивается между *ISR* (первичный обработчик прерывания) и *DPC* (вызов отложенной процедуры).

Масштабируемость и эффективность, достигаемую оптимальным использованием ресурсов и означающую ее применимость для встроенных (*embedded*) систем. В каталоге */dev* нет огромной кучи файлов, соответствующих ненужным драйверам. Драйверы и менеджеры можно запускать просто из

командной строки. И удалять (кроме журналируемых файловых систем) динамически. Можно иметь только тот сервис или те модули, которые реально нужны, причем это не требует серьезных усилий и не порождает проблемы.

Расширяемость и надежность одновременно, поскольку написанный вами драйвер не нужно компилировать в ядро, рискуя вызвать нестабильность системы. Менеджеры ресурсов (сервис логического уровня) работают в кольце 3, при этом можно добавлять свои, не опасаясь за систему. Драйверы работают в кольце 1 и могут вызвать проблемы, но не фатального характера. Кроме того, их достаточно просто писать и отлаживать. Например, постоянная головная боль разработчиков драйверов для *Linux* – получение физически непрерывного блока памяти для *DMA*-устройств – устраняется просто и элегантно, через функцию *mmap*().

Быстрый сетевой протокол FLEET, прозрачный для обмена сообщениями, автоматически обеспечивающий отказоустойчивость, балансирование нагрузки и маршрутизацию между альтернативными путями доступа. Он не так универсален, как *TCP/IP*, но гораздо проще в использовании и эффективнее.

Богатый выбор графических подсистем, включающий *QNX Windows, X*11*R*5 и *Photon*, что позволяет разработчикам выбирать ту, которая лучше подходит для их целей. Для типичных областей применения *QNX* традиционная система *X*11 слишком тяжеловесна, но система *Photon*, построенная на тех же принципах модульности, что и сама *QNX*, позволяет получить полнофункциональный *GUI* (типа *Motif* 2.0), работающий вместе с *POSIX*-совместимой ОС всего в 4 Мбайт памяти.

Конечно, любая медаль имеет оборотную сторону. Поскольку *QNX* не базируется на ядре *Unix*, не следует ожидать бинарной совместимости. Существуют также некоторые ограничения, связанные с ориентацией системы на рынок встроенных систем реального времени. Вот важнейшие из них:

– нет поддержки *SMP*;

– нет своппинга виртуальной памяти на диск;
– неэффективная и нестандартная поддержка нитей (*threads*);
– нет поддержки *Java* (как следствие предыдущего пункта);
– нет поддержки отображения файлов в память;
– многочисленные ограничения файловой системы;
– нет поддержки *Unix-domain sockets*;
– слабые средства разграничения и контроля доступа пользователей;
– отсутствие средств безопасности в рамках собственного сетевого протокола.

Хотя этот список содержит достаточно важные пункты, не все они являются критичными для рынка *QNX*, поскольку она не проектировалась для конкуренции с *Unix*. Но, что гораздо более важно, некоторые пункты этого списка, а также другие ограничения *QNX* являются серьезными недостатками с точки зрения теории систем реального времени. И это притом, что *QNX* является лидером этого рынка!

Что же это за недостатки? Чтобы понять, нужно определить требования к ОС, предназначенной для реализации систем реального времени. Именно этот смысл мы вкладываем в общеупотребительный термин "ОС реального времени" (*Real Time OS*). Использование какой-либо ОС еще не гарантирует получения результата. Можно взять любую ОС такого типа и создать на ее базе некую систему, предназначенную для работы в реальном времени (далее – "система реального времени"), но не способную на это фактически. Чем отличается система реального времени? Есть два основных требования.

– Система должна реагировать на события, успевая обработать их за фиксированное время или к фиксированному моменту времени (далее - "временные рамки").

Для выполнения этого требования система должна обладать предсказуемостью, которую не следует путать с производительностью. Никакой процессор не сделает *Windows* предсказуемой.

– Система должна обладать способностью к параллельной обработке нескольких событий. Если эти события наступают одновременно, система должна успеть обработать всех их в соответствующих каждому из них временных рамках, независимо от количества, порядка поступления и соотношения их временных рамок.

Для выполнения этого требования система должна обладать естественным параллелизмом. Практически это означает, что она должна поддерживать вытесняющую многозадачность, основанную на приоритетах, а также быть способной использовать несколько процессоров одновременно.

Однако можно взять ту или иную ОС и на ее основе создать систему реального времени при условии, что такая ОС отвечает, по крайней мере, следующим требованиям.

1. ОС должна поддерживать вытесняющую многопоточность (*preemptive multi-threading*) и мультипроцессорные архитектуры.

2. Аппаратная архитектура должна поддерживать несколько уровней прерываний (*interrupt levels*), а ОС должна обеспечивать вытеснение (*preemption*) обработчиков прерываний.

3. Каждая нить управления (*thread*) должна иметь способ выражения собственной важности. В идеале планировщик должен предоставлять процессор той нити, у которой осталось меньше всего времени до исчерпания своих временных рамок (алгоритм, известный как *EDF - Earliest Deadline First*). Но, учитывая сложность реализации такой схемы, можно ограничиться наличием приоритетов у нитей при условии поддержки достаточно большого количества уровней приоритетов.

4. ОС должна обеспечивать предсказуемые механизмы для синхронизации между нитями и взаимодействия процессов, разрешающие проблему "инверсии приоритетов". Это означает, что как при передаче данных, так и при синхронизации нитей должно обеспечиваться "наследование приоритетов" (или эквивалентный механизм).

5. Поведение самой ОС после системных вызовов и наступления событий должно быть предсказуемо и известно заранее. Это означает, что разработчики ОС должны специфицировать такие временные характеристики, как "задержка обработки прерывания" (*interrupt latency*), максимальное время маскировки прерываний, а также максимальное время исполнения всех системных вызовов.

6. ОС должна быть способна работать в ограниченных ресурсах, особенно это касается оперативной памяти.

7. Стоимость системы при массовых тиражах должна быть достаточно низкой.

8. ОС должна обеспечивать *API* и "нижележащий" сервис, соответствующий по структуре и реализации требованиям систем реального времени.

Немногие ОС способны выполнить хотя бы часть этих требований, хотя не все из них одинаково важны. Например, *Windows NT* абсолютно не соответствует критическим требованиям 2 и 4 и весьма слабо – условиям, изложенным в п. 3, 5, 6, 7 и 8. В свою очередь заметим, что *QNX* также не вполне отвечает всем требованиям. В частности, она имеет следующие серьезные недостатки:

– неадекватная поддержка нитей и отсутствие поддержки симметричных мультипроцессорных архитектур (*SMP*);

– ограниченное количество уровней прерываний;

– отсутствие поддержки "наследования приоритетов" для механизмов синхронизации (семафоров);

– неспособность работать на системах с объемом памяти менее 512 Кбайт, а с графическими возможностями потребности еще больше;

– относительно высокие цены, даже при больших объемах закупок.

Некоторые из указанных недостатков в определенной степени компенсируются достоинствами *QNX*. Например, практическое отсутствие нитей и *SMP* отчасти уравновешивается

очень малым временем переключения контекста между процессами, а высокие цены – возможностью модульного лицензирования. Семафоры же вообще являются "пятым колесом в телеге" для *QNX*.

§4. Проект *Neutrino*

Еще до появления *Windows* 95 стартовал проект создания совершенно новой ОС, которая, не наследуя устаревшую кодовую базу, могла бы воплотить в себе лучшие идеи, разработанные в теории операционных систем. Этот проект получил кодовое название "*Neutrino*", довольно удачно отражающее его суть – очень маленькая и неуловимо быстрая ОС.

Все проблемы *QNX* можно коротко выразить тремя пунктами:

– недостаточная согласованность с требованиями *POSIX* к системам реального времени;

– невозможность применения на встроенных системах с ресурсами 64 Кбайт – 512 Кбайт;

– невозможность применения на системах высшего уровня (*SMP*-серверах).

Отсюда видно, что глобальная цель проекта *Neutrino* – создание *POSIX*-совместимой масштабируемой ОС, пригодной для построения систем реального времени на самом широком спектре оборудования.

При этом была еще одна цель: добиться независимости кода приложений от характера целевой системы, т.е. код для "тостера" должен быть бинарно совместим с кодом для *SMP*-сервера. Такого рода система должна быть гибкой, эффективной и универсальной.

Однако общая формулировка цели нуждается в уточнениях. Во-первых – почему ОС должна быть *POSIX*-совместимой? На это есть множество причин, приведем лишь некоторые из них.

1. Переносимость кода приложений и возможность использования широкой существующей кодовой базы. *POSIX*

представляет собой идеальный стандарт для этой цели, поскольку он очень строго определяет интерфейсы, не накладывая ограничений на реализацию и предоставляя исчерпывающий набор тестов на совместимость.

2. Независимость приложений от используемого процессора и операционной системы. Уже сейчас перенос приложений, например с платформы *SPARC/Solaris* на *x*86/*QNX*, не представляет значительного труда. *Neutrino* должна сделать этот процесс практически безболезненным.

3. Переносимость средств разработки и наличие достаточного количества квалифицированных разработчиков для *POSIX API*.

4. Близость *POSIX* и *Unix* дает возможность совмещения системы разработки и *runtime*-системы, что позволяет разрабатывать и тестировать приложения еще до появления прототипа устройства, для которого оно предназначено.

5. Правительственные органы некоторых стран (например США) считают совместимость с *POSIX* очень важной. Даже более важной, чем сертификацию по классу *C*2, поскольку *POSIX*-сертифицированная система неявно обладает достаточными средствами защиты.

Впрочем, *POSIX* – это большая группа стандартов, а термин "*Neutrino*", если говорить конкретно, применяется на данном этапе не ко всей ОС, а лишь к ее микроядру. Это микроядро будет совместимо, в частности, со следующими стандартами *POSIX*:

– 1003.1, 1003.1*a*,
– 1003.1*b Realtime*,
– 1003.1*c Threads*,
– 1003.1*d Realtime Extensions*,
– 1003.13 *Realtime Profile Support*.

Кроме того, микроядро *Neutrino* разрабатывалось с учетом некоторых других требований, таких, например, как поддержка твердотельных дисков и возможности исполнения кода непосредственно из *ROM*.

Архитектура микроядра Neutrino. Указанные цели продекларировать гораздо легче, чем их достичь. Например, идея реализации ОС для систем реального времени с интерфейсом *POSIX* существует давно, но никому этого пока не удавалось сделать. *POSIX*-системы имеют репутацию "раздутых", поскольку они ассоциируются в первую очередь с *Unix*. В некотором смысле *Neutrino* является доказательством возможности существования компактной *POSIX*-системы.

Для достижения этих целей недостаточно было просто косметической модернизации микроядра *QNX*. *Neutrino* представляет собой значительно более совершенную модель, выполняющую гораздо больше функций, чем микроядро *QNX*, имея при этом лучшую производительность и временные характеристики. Улучшенное микроядро позволило также вчетверо уменьшить размер менеджера процессов (с 80 до 20 Кбайт), уменьшив таким образом их суммарный размер почти вдвое.

Микроядро и наноядро. Очевидно, расширение функций привело к увеличению размера микроядра с 10 до 28 Кбайт. Однако его содержимое теперь лучше структурировано. Фактически в нем выделилось "наноядро", обеспечивающее поддержку фундаментальных объектов микроядра, которое в свою очередь поддерживает базовый сервис для пользовательских нитей и дополнительных системных модулей.

Функции, включенные в микроядро, были выбраны по принципу минимального количества операций, необходимых для их исполнения. Все относительно сложные функции были вынесены во внешние модули.

По этой причине *Neutrino* остается достаточно маленьким и простым и по-прежнему оправдывает название "микроядра". Например, *Neutrino* поддерживает понятие нити, работающей в контексте процесса, но не может создавать процессы. В микроядре вообще нет кода, предназначенного для управления виртуальной памятью, необходимой для реализации защиты процессов. Такое решение объясняется тем, что некоторые системы реального

времени предъявляют более жесткие требования к размеру и скорости исполнения кода, чем к защите, поскольку имеют контролируемую среду исполнения. Кроме того, некоторые процессоры вообще не поддерживают механизм виртуальной памяти. А на процессорах с архитектурой, отличной от *x*86, все вообще выглядит иначе. Так что такой подход повышает модульность и эффективность системы, а также упрощает ее перенос на другие платформы.

Все системные вызовы *Neutrino* могут вытесняться при необходимости, чтобы обработать вызов от нити с более высоким приоритетом, причем даже в процессе передачи сообщений. Это качество микроядра, а также его сравнительная простота и малый размер позволяют минимизировать невытесняемые последовательности кода в системе. В свою очередь, за счет этого улучшаются временные характеристики системы. Скромные требования к памяти упрощают разработку встроенных систем низшего уровня. Кроме того, это уменьшает количество блокировок в коде (*spin-locks*), необходимых для поддержки мультипроцессорных архитектур, что упрощает реализацию *SMP* и повышает эффективность использования дополнительных процессоров. Улучшаются также характеристики ОС, необходимые для построения систем высокого уровня.

По заявлениям *QSSL,* бета-тестирование *SMP Neutrino* продемонстрировало близкий к линейному рост производительности при добавлении дополнительных процессоров (до 8), при автоматической балансировке нагрузки. Многое в *QNX* и *Neutrino* "недостижимо". Кроме того, реализация *SMP Neutrino* допускает ручное распределение процессов между процессорами, что позволит добиться еще большей эффективности в контролируемой среде исполнения. Эта система уже вызвала большой интерес со стороны телекоммуникационных компаний, нуждающихся в сверхпроизводительных системах для реализации мощных коммутационных систем.

Микроядро и дополнительные модули. Главное отличие микроядра *Neutrino* от микроядра *QNX* – это его соотношение с внешними модулями. В *QNX* микроядро физически существовало в коде менеджера процессов, что означало необходимость использования последнего даже там, где не нужны его функции. Поскольку *Neutrino* должна быть применима для систем самого низкого уровня, типа "интеллектуального тостера", это ограничение необходимо было ликвидировать с целью снижения требований к памяти. Микроядро *Neutrino* может существовать вне менеджера процессов, что позволяет связать его с пользовательским кодом, получив таким образом сущность, называемую "системный процесс" (*system process*). Такой процесс не требует для работы ни ОС, ни даже *BIOS*, поскольку в комплект системы входит набор модулей *IPL* (начальной загрузки), способных заменить *BIOS* в этом качестве.

Системный процесс способен к самостоятельному исполнению. Тостеру не нужна ОС в полном смысле этого слова, ему нужна управляющая система реального времени. *Neutrino* дает возможность разрабатывать такие системы, пользуясь стандартным *API*, определенным в *POSIX*, обеспечивая при этом большинство необходимых для подобной системы функций.

Если же для системы требуется полноценная ОС, то нужно связать микроядро с менеджером процессов *Neutrino* (*ProcNto*), затем сформировать шаблон ядра и получить из него загружаемый двоичный образ – так, как это делается в *QNX*.

А что, если для реализации системы *ProcNto* не подходит? *Neutrino* предоставляет разработчикам целый спектр решений на этот случай.

Альтернативная реализация дополнительного сервиса. Менеджер процессов *ProcNto* представляет собой набор нитей, исполняющихся в адресном пространстве микроядра, и отвечает за управление памятью, поддержку пространства имен и создание новых процессов. Не всегда эти функции бывают нужны одновременно. Например, встроенной системе, использующей

фиксированный набор процессов, "зашитых" в ядро, вряд ли понадобятся функции создания новых процессов, с поддержкой различных форматов. Поэтому, несмотря на свой малый размер, *ProcNto* может оказаться нецелесообразно велик. В таких случаях разработчик системы может реализовать самостоятельно альтернативный вариант, в котором вместо *ProcNto* используется его собственный код, связанный с определенной библиотекой, содержащей упрощенные заменители для минимально необходимого набора функций. Так, например, можно переопределить реализацию функций *open*(), *read*() и *write*() – если это все, что необходимо для системы.

Расширения ядра и добавление новых системных вызовов. Еще одно новшество микроядра *Neutrino* заключается в поддержке расширений (*extensions*). Код *Neutrino* содержит различные таблицы переходов, которые могут быть переопределены в момент исполнения любой нитью, работающей в адресном пространстве микроядра. Эти таблицы могут указывать на адреса функций в любой другой нити. Например, *ProcNto* использует этот механизм для замены примитивных функций управления памятью, содержащихся в *Neutrino*, на более сложные, позволяющие выполнять операции с множественными виртуальными адресными пространствами, соответствующими различным процессам. Само микроядро не содержит кода, способного работать с виртуальной памятью, чтобы не обременять системы, которые этот механизм не поддерживают или не используют.

ProcNto также добавляет в микроядро вызов для создания новых нитей в контексте другого процесса, чего *Neutrino* самостоятельно делать не может, поскольку оно было сознательно лишено возможности манипулировать чужим адресным пространством. Аналогичные расширения *ProcNto* делает для обеспечения передачи сообщений между процессами, работающими в защищенных виртуальных адресных пространствах.

Neutrino также содержит специальную точку входа, через которую нити, исполняющиеся в адресном пространстве микроядра, могут передавать ему адреса функций. Затем микроядро вызывает эти функции со своим контекстом. Этот механизм используется менеджерами сети для того, чтобы выполнять манипуляции объектами микроядер на различных узлах от их собственного имени. Это и позволяет добиться полной прозрачности сетевого взаимодействия в *QNX/Neutrino*, поскольку исчезает разница между локальным и удаленным исполнением программы. Сеть *Neutrino* превращается в "виртуальный компьютер", позволяя создавать высокопроизводительные кластерные *SMP*-системы.

Наконец, привилегированные нити могут определять и регистрировать в микроядре новые системные вызовы, расширяя его функциональные возможности.

Управление процессами и памятью. Как уже отмечалось, управление процессами и памятью не является, строго говоря, функцией *Neutrino* – это функция менеджера процессов *ProcNto*, который, кроме этого, занимается поддержкой пространства имен ввода/вывода и еще рядом "мелочей". Однако обзор был бы неполным без рассмотрения данного вопроса.

Прежде чем управлять процессами, необходимо иметь возможность загружать их с какого-либо носителя. С этой целью в состав *ProcNto* также входят "нить загрузчика" и "нить терминатора". Нить загрузчика обеспечивает загрузку исполняемых модулей в формате *ELF*, *QNX*4 и *shell*-скриптов. Формат *ELF* (известный также как *Evil Linkage Format J*) является для *Neutrino* стандартным, поскольку он обеспечивает ряд преимуществ, таких как поддержка динамического связывания, и, что очень важно для встроенных систем, совместим со спецификацией *XIP* (*eXecute-In-Place*), предусматривающей исполнение кода прямо из *ROM*, без загрузки в *RAM*. Нить терминатора обеспечивает "уборку мусора" после завершения процессов на тот случай, если они не смогли сделать это сами.

Еще одна функция *ProcNto* – поддержка виртуальной файловой системы, встраиваемой в двоичный загружаемый образ системы. Для встраиваемых систем, не требующих хранения данных, этот вариант очень привлекателен, поскольку он упрощает структуру системы, снижает требования к памяти и стоимость лицензии.

И все же самое интересное – это управление памятью. Данный вопрос является достаточно болезненным, поскольку от его решения зависит очень многое. Решение, примененное в *QNX*, не было достаточно гибким. *QNX* всегда использует виртуальную память, что не позволяет делать этого на некоторых типах *Intel*-совместимых процессоров, довольно широко применяемых во встроенных системах, например производства *National Semiconductor* или *AMD*, поскольку они не содержат *Paged-MMU* (устройство управления виртуальной памятью). Кроме того, зависимость микроядра от специфичной для *x*86 аппаратуры *MMU* затрудняет перенос системы на другие платформы.

В результате *Neutrino* принимает соломоново решение – предоставить выбор модели защиты памяти разработчикам. Код *Neutrino* не использует *MMU* или виртуальную память в явном виде. Это достигается за счет выноса функции инициализации *MMU* во внешний модуль (*mmuon*) и выноса функций управления виртуальной памятью в расширения микроядра, обеспечиваемые *ProcNto*. Для поддержки *MMU* модуль *mmuon* нужно включить в ядро, после чего менеджер процессов сможет поддерживать виртуальную память. Этот модуль не является "сервером", он выполняет инициализацию процессора и немедленно завершает свою работу. Сам менеджер процессов также существует в нескольких вариантах, соответствующих типу защиты памяти. Таким образом, *Neutrino/ProcNto* поддерживает 4 варианта управления памятью, от полного отсутствия защиты до предоставления каждому процессу собственного виртуального адресного пространства в 4 Гбайт. В будущем появится также

версия *ProcNto*, поддерживающая своппинг виртуальной памяти на диск, что может оказаться желательным для некоторых систем верхнего уровня.

Вариант 1. *Физическая память.* Все нити перемещаются при построении системы в адреса, расположенные в адресном пространстве *Neutrino*. Менеджер процессов обычно отсутствует. Это типичная конфигурация, которую предоставляют различные *realtime executive*, но отличие *Neutrino* в том, что она пытается даже в этой модели памяти выполнять (насколько это возможно) функцию *mmap*(), что позволяет обходиться без изменения исходного кода системы при смене модели памяти.

Вариант 2. *Защита "системных" нитей от пользовательских.* В этой модели нити, работающие в адресном пространстве *Neutrino* (это обычно *ProcNto*, менеджер сети или определенная разработчиком нить), защищены от остальных нитей, но последние не защищены друг от друга. Защита обеспечивается аппаратно *MMU*, путем маркировки страниц как "системных" и "пользовательских".

Вариант 3. *Защита всех нитей друг от друга.* Этот вариант отличается от предыдущего тем, что пользовательские нити также защищены друг от друга. Защита обеспечивается путем динамической маркировки страниц. Изначально страницы всех нитей маркируются как системные. Затем при передаче управления любой из пользовательских нитей ее страницы помечаются как пользовательские (тем самым разрешая нити доступ к своим данным), до тех пор, пока она не потеряет управление, после чего они снова маркируются как системные.

Вариант 4. *Виртуальная память.* В этой модели каждый процесс имеет собственное адресное пространство, начинающееся с адреса 0 и защищенное от остальных процессов. Нити процесса делят с ним одно адресное пространство. Системное адресное пространство *Neutrino* также защищено от остальных процессов. Защита поддерживается аппаратурой *Paged-MMU* и реализуется соответствующей версией *ProcNto*.

Объекты и сервис микроядра. *Neutrino* поддерживает 48 системных вызовов (*QNX* – 14), обеспечивающих нити, передачу сообщений, сигналы, системные часы и таймеры, обработку прерываний и механизмы синхронизации нитей.

Процессы и нити: диспетчеризация и синхронизация. *Neutrino* поддерживает модель нитей *POSIX* 1003.1*c*, в соответствии с которой процесс может динамически создавать и уничтожать одну или более нитей. Разработчики могут по своему выбору использовать для работы с нитями либо *API Neutrino*, либо стандартную библиотеку *pthreads*.

Этот же стандарт определяет, что нити должны иметь собственные уровни приоритетов. *Neutrino* к моменту выхода окончательной версии будет поддерживать 256 уровней, причем каждая нить может также иметь собственный алгоритм диспетчеризации, список которых традиционен для *QNX* (и определен *POSIX*) – *round-robin, FIFO* и адаптивный.

Разумеется, поддержка нитей подразумевает также, что они могут делить общие данные. Для обеспечения синхронизации *Neutrino* поддерживает два механизма: условные переменные (*condvars*) и блоки взаимного исключения (*mutexes*). Для этих объектов микроядро поддерживает механизм наследования приоритетов.

Реализация *mutexes* отличается высокой эффективностью. Получение или освобождение несвязанного объекта типа *mutex* требует выполнения всего одного кода. Для сравнения, в *Windows NT* эта операция может занимать время до 700 мс.

Модель событий и средства обмена сообщениями. Модель событий *Neutrino* представляет собой еще одно значительное достижение этой системы. Учитывая сложность и многообразие форм событий и способов уведомления о них, реализация такой системы в каждой паре "клиент-сервер" может занять значительный объем кода и затруднить разработку надежной модели взаимодействия. Поэтому *Neutrino* использует другой подход,

называемый *event steering*, при котором сервер может передать микроядру форму уведомления клиента, которую он у него запросил.

Практически нити получают уведомления от одного из трех типов источников: сообщение от другой нити, прерывание или таймер. События существуют в форме синхронных сообщений, асинхронных пульсов, *Unix* или *POSIX*-сигналов, прерываний, а также специального события *ForceReady*, вызывающего безусловный переход нити в состояние *Ready* без доставки какого-либо события. Механизм *event steering* работает следующим образом:

• клиент посылает серверу сообщение, содержащее структуру с описанием желаемого механизма уведомления;

• сервер регистрирует эту форму в микроядре и отвечает клиенту, выводя его из блокировки;

• когда возникает необходимость в уведомлении клиента, сервер посылает ему сообщение, а микроядро транслирует его в заказанную форму.

Сигналы в *Neutrino* поддерживаются в двух разновидностях: классические сигналы *Unix* и *real-time* сигналы *POSIX*, с которыми можно передавать короткую порцию данных (4 байт). Еще одно различие между ними заключается в том, что сигналы *POSIX* при поступлении к процессу буферизуются, пока какая-либо из нитей процесса не проявит к ним интерес. *Neutrino* расширяет семантику *POSIX* тем, что такое поведение можно заказать выборочно для любого сигнала, в том числе для стандартных сигналов *Unix*. Кроме того, *Neutrino* позволяет адресовать сигнал к конкретной нити внутри процесса, а не просто к процессу.

Сообщения являются классическим механизмом *QNX*, который сохранился и в *Neutrino,* но несколько расширился и усложнился. Помимо синхронных сообщений *Neutrino* поддерживает короткие асинхронные сообщения, называемые "пульсами" (*Pulses*), позволяющие передать 4 байт данных и

реализованные на той же основе, что и сигналы *POSIX*. Пульсы представляют собой заменитель недостаточно гибкого механизма *Proxy*, применяемого в *QNX*.

Введение полноценного понятия нитей потребовало также усложнения механизма передачи сообщений. Если в *QNX* процессы устанавливали виртуальный канал непосредственно друг с другом, то в *Neutrino* они должны использовать для этой цели Соединения (*connections*) и Каналы (*channels*).

Каналы используются нитями на стороне сервера, поскольку они необходимы для приема сообщений (но не для получения ответов на посланные сообщения). Соединения используются нитями клиентов для посылки сообщений или пульсов в каналы. Несколько нитей могут посылать сообщения в один канал, разделяя между собой одно соединение. При этом несколько нитей могут быть одновременно блокированы на канале. И соединения и каналы идентифицируются в программах числовыми идентификаторами, аналогично файловым дескрипторам и гнездам.

Системные часы и таймеры. Системные часы используются для отслеживания времени суток, которое, в свою очередь, используется таймерами. *Neutrino* обеспечивает системный вызов для плавной синхронизации часов между несколькими системами.

Общая проблема всех операционных систем с вытесняющей многозадачностью заключается в потенциальной возможности вытеснения нити, установившей таймер, до того, как она успела выполнить запрос на сервис, для которого этот таймер предназначался. В результате, когда запрос все же начнет выполняться, таймер может быть уже исчерпан, что приводит к ненадежному функционированию системы.

В *Neutrino* эта проблема решена за счет изменения процедуры и механизма использования таймеров. Они также имеют весьма эффективную реализацию. Практически, когда приложение хочет запросить сервис, нуждающийся в тайм-ауте, оно запрашивает у микроядра автоматический запуск таймера в случае перехода

приложения в определенное состояние. Запуск таймера и запрос сервиса становятся атомарной операцией, исключая тем самым неоднозначности.

Обработка прерываний. Этот пункт является одним из самых сложных при разработке ОС для системы реального времени, поскольку необходимо выполнить множество плохо согласующихся требований. *API* обработки прерываний в достаточной степени соответствует предварительному стандарту *POSIX* 1003.1*d*. Модель обработки выглядит следующим образом.

Нить, имеющая соответствующие привилегии, может динамически устанавливать и удалять обработчики прерываний путем передачи микроядру адреса функции-обработчика (*ISR*). При возникновении прерывания обработка передается сначала в микроядро, которое содержит код для его переадресации. Перед вызовом *ISR* микроядро сохраняет контекст исполняемой нити и переустанавливает регистры процессора таким образом, что *ISR* имеет доступ к адресному пространству нити, в которой он содержится. Это позволяет *ISR* выполнить обработку, пользуясь данными и буферами нити, в которой она содержится, или буферизовать принятые данные для последующей обработки этой нитью. Это также дает возможность *ISR* осуществлять доступ ко всем устройствам, находящимся в домене ответственности (области префиксов) данной нити, и непосредственно выполнять операции ввода-вывода. В результате драйверы аппаратуры оказываются не связанными с ядром и могут работать в кольце 1.

К одному прерыванию можно присоединить несколько *ISR*, при этом они все будут вызваны. *ISR* должна вернуть управление по возможности быстро, отложив длительные операции для выполнения соответствующей нитью драйвера и информировав его об этом, например, с помощью пульса. Если возвращенное любой *ISR* значение указывает на то, что возникло некоторое событие, оно будет буферизовано. После вызова всех *ISR* микроядро завершает работу с программируемым контроллером прерываний и

возвращает управление из прерывания. Однако возврат не обязательно происходит в то место, где оно произошло. Если одно из буферизованных событий вызвало переход более приоритетной нити в состояние *READY*, то управление будет возвращено в контекст этой нити. Основное отличие этой схемы от механизма *ISR/DPC*, используемого в *Windows NT*, заключается в том, что все *DPC* диспетчеризуются с одним и тем же уровнем приоритета, а это означает невозможность вытеснения одного *DPC* другим и приводит к непредсказуемой задержке обработки более приоритетных прерываний.

Описанная модель обеспечивает очень хорошие временные характеристики (*interrupt latency* и *sheduling latency*), поскольку *Neutrino* запрещает прерывания лишь на очень короткие промежутки времени, не зависящие от данных. Максимальное время задержки обработки прерывания можно вычислить на основании задержки, вносимой микроядром, и суммы времен исполнения всех *ISR*, назначенных для прерываний с более высоким аппаратным приоритетом.

Эту сумму также можно контролировать, поскольку *Neutrino* предоставляет *API* для переназначения приоритетов уровней прерываний в контроллере. Можно также свести ее к нулю, разработав систему таким образом, чтобы на уровне *ISR* ничего не делалось, а вся работа происходила бы на уровне нитей с приоритетами, назначенными разработчиком, а не в соответствии с приоритетами аппаратных прерываний.

Наконец, помимо обычных прерываний, нить может перехватить некоторые внутренние события *Neutrino*, и даже немаскируемые прерывания процессора (*NMI*). Заметим, что поскольку механизм системных вызовов основан на программных прерываниях, работающих так же, как и аппаратные, обработка системных вызовов происходит идентично обработке прерываний – вытеснение процессов при обработке прерываний и системных вызовов происходит одинаково быстро.

Практические аспекты применения системы. В соответствии с целями проекта система *Neutrino* должна быть в конечном итоге пригодна для решения таких разных задач, как создание системы управления беспилотным летательным аппаратом или создание корпоративного *Internet*-сервера. Пригодность системы может быть теоретической или практической, что определяется такими факторами, как наличие средств разработки, графических средств, дополнительного программного обеспечения, поставляемого независимыми разработчиками, и многими другими объективными и субъективными факторами, в том числе общественным мнением.

Пригодность также не означает целесообразность, которая всегда зависит от конкретных факторов, играющих роль в том или ином проекте. Например, для реализации *Internet*-сервера на базе *QNX/Neutrino*.

Графическая подсистема Photon. Существует несколько довольно известных операционных систем, пригодных для создания систем реального времени. Однако большинство из них неспособны решить проблему реализации графического интерфейса пользователя (*GUI*) для встраиваемых систем, поскольку представляют собой очень ограниченные по возможностям *realtime executives*. Те, что способны поддержать полноценный *GUI*, например *RtLinux*, не позволяют извлечь из этого практическую пользу, поскольку реализация традиционных *GUI*, типа *X*11, связана с очень высокими затратами ресурсов, особенно памяти.

Приступая к проекту *Neutrino*, его разработчики продумали и это. Для реализации *GUI*, пригодного к использованию во встраиваемых системах реального времени, был начат еще один параллельный проект – *Photon*. В результате появилась графическая подсистема, по внешнему виду и структуре пользовательского интерфейса очень похожая на *X*11/*Motif*, но весьма скромная по затратам ресурсов. Это стало возможным благодаря применению

при ее разработке принципа модульности и ряда новых фундаментальных идей. Вот лишь наиболее существенные из них:

– расширенный набор оптимизированных драйверов, которые имеют теперь новую архитектуру, повышенное быстродействие (не используется *int*10) и обеспечивают поддержку режимов *High Color* и *True Color* для всех адаптеров. Список поддерживаемых адаптеров пополнился такими моделями, как *Matrox Millenium, ATI Rage* 3*D*/3*D*II, *IBM XGA, Trident* 9470;

– поддержка мобильных масштабируемых шрифтов формата *Bitstream TrueDOC* через фронт-сервер, обеспечивающий единую систему именования и отображения имен в шрифтовые файлы с учетом кодировки *Unicode* (*UTF*-8), а также низкоуровневый доступ к шрифтам из приложений;

– документация и средства разработки для файлов отображения клавиатуры, обеспечивающих поддержку клавиатуры любого национального языка, в том числе русского;

– новые виджеты, например *PtHTML, PtTree, PtDivider, PtMenuBar, PtGrid, PtFontSel, RtProgress* и ряд других, которые значительно перекрывают набор виджетов *Motif* 2.0;

– примеры исходного кода и документация для создания собственных виджетов и включения их в генератор приложений *PhAB*, который таким образом стал наконец расширяемым;

– набор новых приложений, включающий в себя *File Manager, CD Player, Audio Player, Calculator, Personal Information Manager*;

– графическая программа конфигурирования видеорежима;

– система *XinPh*, которая обеспечивает запуск системы *X Window* в окне системы *Photon*;

– фронт-процессор клавиатуры для поддержки азиатских языков (японский, китайский, корейский).

Бета-версия *Photon* 1.12 содержит ряд новых средств, включая:

– поддержку печати на принтерах *PostScript, Epson и Hewlett-Packard, Canon*;

– поддержку протокола *Drag'n'Drop* на уровне виджетов;

– новые виджеты (*PtNumber, PtPrintSel, PtFileSelector* и другие);

– универсальный драйвер видеоадаптеров класса *VESA* 2.0 (любые современные адаптеры, которые можно перевести в режим *flat-memory*);

– графический редактор текстов, поддерживающий кодировку *Unicode*;

– расширения *API* для поддержки новых возможностей *Neutrino*.

Сетевой сервис и файловая система. Сетевой сервис в *Neutrino* представлен только протоколом *TCP/IP*. Разработчики *Neutrino* хотели предоставить пользователям полный набор функциональных возможностей классического стека *TCP/IP*, но были вынуждены учитывать потребности рынка встроенных систем, для которых классическая реализация слишком велика и содержит много ненужных элементов. В результате они создали специальную версию стека для встроенных систем – *Micro TCP/IP*, который занимает всего около 40 Кбайт кода за счет ряда ограничений. Для тех же, кому нужны все возможности *TCP/IP*, например динамическая маршрутизация, будет предоставлен другой вариант, совместимый на 100% с *BSD-sockets*.

Neutrino также поддерживает сетевой протокол *FLEET*, используемый сейчас в *QNX*, с некоторыми усовершенствованиями, касающимися автоконфигурирования.

Файловая система в *Neutrino* реализована иначе, чем в *QNX*. Главное функциональное отличие – улучшенная приспособленность к сменным носителям. Для этого была изменена модель взаимодействия менеджеров ресурсов и драйверов устройств, примененная в *QNX*. Если там менеджер файловой системы обращался к драйверу устройства для получения сервиса физического уровня, то в *Neutrino* все наоборот. Теперь приложение обращается к драйверу, который определяет тип файловой системы

(по сигнатурам) и динамически загружает соответствующую файловую систему, реализованную в виде разделяемой библиотеки.

Собственно говоря, файловых систем в *Neutrino* много. Поддерживаются все файловые системы, имеющиеся в *QNX*, а также виртуальная файловая система *Proc*. Для обеспечения обмена данными с другими операционными системами *Neutrino* также поддерживает файловую систему *CIFS* (*Common Internet File System*), которая представляет собой обобщенный вариант *SMB*, способный использовать любой сервис имен (например *DNS*) вместо *Netbios NS*. Разумеется, все файловые системы реализованы с учетом возможности работы в ограниченных ресурсах, т.е. очень компактно. Например, код для поддержки файловой системы *Tiny QNX* (*POSIX*) занимает всего 12 Кбайт, конечно, за счет некоторых ограничений. Эта система способна читать разделы, созданные *QNX*4, но не может создавать жесткие ссылки и файлы с именами длиннее 16 символов (иначе говоря, не может писать в файл *.inodes*).

Средства разработки и совместимость. Для успеха любой операционной системы необходимо наличие высококачественных средств разработки приложений. В отличие от большинства систем *Neutrino* имеет свои собственные средства разработки вместо обычного компилятора *GCC* и связанных с ним программ. Однако эти средства ничуть не хуже и они вполне стандартны – это компилятор *Watcom C/C++* 10.6. Среда разработки включает все стандартные средства *Watcom*. Существует версия системы кросс-разработки приложений *QNX/Neutrino* под *Windows NT*, с использованием системы разработки *Watcom*. Таким образом, разработчики, привыкшие использовать *Windows*, могут больше не пересаживаться за *QNX* и пользоваться "кровавыми" командными строками.

Более того, система *Willows* предоставит возможность компиляции приложений, написанных с использованием *API Win*32 под *QNX/Neutrino/Photon*. При этом обеспечивается поддержка бинарных объектов (*DLL* от третьих фирм) и непосредственное

исполнение приложений *Windows* через эмуляцию. Однако перекомпилированные приложения будут иметь преимущество в скорости (вероятно, они будут работать быстрее, чем в *Windows*) и смогут использовать одновременно *API* системы *QNX/Neutrino* для выполнения задач реального времени и обмена сообщениями.

Между тем компилятор *GCC* 2.7.2 был перенесен в *QNX* и в *Neutrino*. А также перенесена стандартная библиотека *C* из *Unix* (*libc*). Эти программы предоставляются бесплатно и являются неплохим дополнением к системе разработки *Watcom*. Данный факт может сыграть ключевую роль в ускорении переноса приложений из *Unix* и включении *QNX/Neutrino* в список платформ, поддерживаемых разработчиками приложений для *Unix*.

Таким образом, *QNX/Neutrino* становится платформой, на которую можно будет без проблем перенести приложения и из *Unix,* и из *Windows,* что должно существенно расширить круг готовых приложений для этой платформы.

Средства работы с Internet и разработка Internet-приложений. Ни одна современная ОС не может сегодня игнорировать "фактор *Internet*". В поддержке *Internet* нет ничего необычного, за исключением того, что и здесь нужно было учитывать основное требование встроенных систем – низкие затраты ресурсов. Сочетание *QNX/Neutrino* и графической системы *Photon* открывает совершенно новые возможности для рынка встроенных клиентских систем для *Internet*-устройств "карманного" размера (*handheld devices*). Фирма *QSSL* лицензировала *WEB-browser* фирмы *Spyglass* (на котором также основан *MS Internet Explorer*) и разработала комплект клиентских приложений для работы с *Internet* (*Voyager Pro*), включающий в себя *Web*-браузер, *mail/news*-клиент и графическую программу установления соединения с *ISP*.

Этот комплект доступен почти полностью с исходным кодом под названием *Internet Applliance Toolkit* (*IAT*). Разработчики могут использовать этот код для создания модифицированных версий клиентских программ (*browser, mail, news*), оптимизированных под

конкретные нужды. В результате разработчики получают уникальную возможность создавать встроенные системы с комплектом *Internet*-приложений за очень короткое время, поскольку все, что им потребуется, – это модифицировать пользовательский интерфейс с помощью визуального средства разработки (*Photon Application Builder*).

Заключение. Neutrino – не единственная новая разработка в области операционных систем. Существует ряд других интересных проектов, некоторые из них построены на принципах, сходных с *QNX* (микроядро и обмен сообщениями), и пригодны для применения в системах реального времени. Такие системы, как *L*3/*L*4 и *MkLinux*, имеют также некоторые преимущества перед существующей версией *Neutrino*, например, поддержку алгоритма диспетчеризации *EDF* и возможность исполнять приложения *Linux* (которых достаточно много). Тем не менее ни одна из этих систем не пригодна для применения во встраиваемых системах с ограниченными ресурсами, представляющими наибольший интерес для рынка систем реального времени.

Чем *Neutrino* отличается от *QNX*? Те, кто хорошо знаком с *QNX*, могут сделать вывод, что *Neutrino* имеет множество преимуществ, как-то:

– большую степень масштабируемости, как вниз, так и вверх;

– более высокую производительность, за счет улучшения архитектуры (нити);

– большее количество уровней приоритетов (256);

– новые средства синхронизации (*condvars* и *mutexes*) с поддержкой наследования приоритетов;

– отсутствие необходимости в *BIOS*;

– улучшенные средства асинхронного обмена (*pulses*);

– поддержку *SMP*;

– поддержку файлов, отображаемых в память;

– поддержку *Unix-domain sockets* и 100% совместимость с *BSD-sockets*;

– современный формат исполняемых модулей (*ELF*, с расширениями для сжатия);

– поддержку динамически связываемых библиотек (*DLL*);

– повышенный уровень безопасности (шифрование сообщений);

– усовершенствованную файловую систему;

– поддержку виртуальной файловой системы *Proc*;

– мультиплатформенность (потенциальную);

– поддержку *Java*;

– расширяемость системы за счет подстановки системных вызовов;

– более гибкую цену, за счет более модульной структуры.

Обратная сторона медали? Перечисленные нововведения настолько глобальны, что они неизбежно должны привести к некоторой несовместимости с *QNX* 4.*x*.

QNX или *Neutrino*? Такой вопрос возникает у тех, кто уже давно использует *QNX* для своих разработок, равно как и у тех, кто только начинает работу в этой области. *QNX*4 является наиболее полной системой, с точки зрения функциональных возможностей. Она проверена временем, следовательно, ее можно считать также более надежной. Но ее архитектура отличается меньшей гибкостью, чем у *Neutrino*, что неизбежно означает некоторое замедление или прекращение ее дальнейшего развития в пользу *Neutrino*.

На данный момент *API* двух платформ имеют значительные непересекающиеся части, однако представители *QSSL* утверждают, что эта ситуация будет улажена. Вероятно, тогда проблема несколько упростится. Уже сейчас в системе *Photon*, которая поддерживается для обеих платформ, появились функции, маскирующие различия между *QNX* и *Neutrino* (например *proxy/pulses*). Впрочем, это не решит всех проблем. *Neutrino* предлагает совершенно иную парадигму программирования, с сильным акцентом на использование многопоточности, поэтому для ее эффективного использования особую актуальность приобретет переобучение специалистов, привыкших к *QNX*4.

Следует отметить, что богатый набор функций, реализованный в соответствии со стандартами *POSIX*, уже сейчас делает *Neutrino* весьма привлекательной альтернативой существующим решениям. Исследования рынка систем реального времени показали, что для систем с жесткими ограничениями ресурсов все еще широко используются различные исполняющие системы (*realtime executive*) с нестандартизированным *API* и часто довольно бедными функциональными возможностями. Именно поэтому разработчики *Neutrino* приняли решение выпустить предварительную версию системы, не имеющую пока возможностей для масштабирования вверх, но уже пригодную для применения во встроенных системах. Не случайно фирма *Intel* с некоторых пор поддерживает очень хорошие отношения с *QSSL* и покупает лицензии на *Neutrino* в огромных количествах. Несмотря на наличие собственной ОС реального времени, *Intel* также официально объявила о том, что *QNX/Neutrino* является для нее "*Realtime OS of preference*".

§5. Программное обеспечение промышленных систем

Как показала практика, стоимость создания систем промышленной автоматизации определяется в основном затратами на разработку ПО, доля которого может доходить до 60%.

Чем располагает разработчик промышленной системы? В первую очередь это операционные системы, поддерживающие функционирование разрабатываемого приложения. В сфере промышленной автоматизации свой мир операционных систем – ОС реального времени (ОС РВ). Для VME-процессоров, например, существует 17 ОС (OS-9/OS-9000, VRTX/Spectra, VxWorks, PDOS, pSOS+, LynxOS, VMEexec, iRMX, C-Exec, QNX и т. д.). Поскольку крупные компании-производители обеспечивают для своих процессоров и устройств ввода/вывода полную поддержку сразу нескольких ОС РВ, можно найти версии систем из приведенного списка для многих платформ: 68000, PowerPC, ix86, Pentium.

ОС РВ характеризуется прежде всего малым временем реакции на внешние события. Так, гарантированное время реакции системы на процессоре MC68040/25 МГц под управлением операционной системы OS-9 v3.0/Atomic составляет 3 мкс. Как правило, это многопользовательские многозадачные ОС, выполненные по технологии микроядра. Последние версии ОС имеют прерываемое микроядро, что гарантирует быструю реакцию на внешнее событие при любом состоянии системы. Особенность большинства ОС – возможность стопроцентного размещения в памяти ядра, сетевого и графического обеспечения, драйверов и прикладных программ. Для встроенных бездисковых систем это чрезвычайно важно.

Традиционно ОС РВ делятся на "жесткие" и "мягкие". Система "жесткого" РВ должна без сбоев отвечать на внешние события в рамках заранее определенного интервала времени. Время ответа должно быть предсказуемым и не зависеть от текущего состояния системы. "Мягкая" ОС РВ тоже должна отвечать очень быстро, но гарантированное время ответа в ней не обеспечивается. Здесь нужно отметить, что временные характеристики последних версий промышленных ОС практически стерли ранее существовавшую грань между двумя этими разновидностями. Сейчас OS-9, ранее считавшаяся "мягкой" ОС, практически не уступает классическим "жестким" ОС - pSOS+ и VxWorks.

Все большее распространение в сфере промышленной автоматизации получают ОС общего назначения: Unix в разных реализациях, NT, OS/2, VMS. Для этого есть несколько причин. В различные реализации Unix стали включаться ядра реального времени, и это движение поддержано открытыми стандартами. В POSIX стандартизованы, например, диспетчеризация и синхронизация процессов, нити, тайм-ауты, управление прерываниями. Во-вторых, трудно устоять перед мощной экспансией ПК, особенно после выхода Windows NT, и рабочих станций. Третья причина состоит в наличии на платформах общего назначения широкого разнообразия инструментальных средств.

Действительно, кроме базовой поддержки, предоставляемой ОС РВ, для создания приложений разработчику требуется развитый инструментарий, которого, конечно, больше в общецелевых системах. Нельзя сказать, что ОС РВ в этом отношении бедны – в них обычно поддерживаются сетевые протоколы, X Window, Motif, но конкурировать на равных им все же тяжело, особенно с новейшими инструментальными технологиями. С другой стороны, все ОС РВ имеют собственные среды разработки (FasTrack, Microtec, MasterWorks), чьи возможности в определенных аспектах превосходят аналогичные общецелевые средства, которые, например, не могут помочь разработчику систем РВ на финальных стадиях отладки, когда нужно измерять время выполнения программ и время реакции системы. Поэтому можно выбрать подход, при котором ОС общего назначения используются в качестве среды для разработки ПО целевого приложения реального времени, а в качестве среды исполнения применяется та или иная ОС РВ. В этом варианте загрузка и отладка целевого ПО производится с инструментального компьютера по сети.

Конечно, можно создать ПО системы автоматизации, опираясь только на общецелевые средства, однако цена такой разработки будет очень высока. Системы автоматизации – это такая же прикладная область, как, например, графика или САПР, и кроме универсального, здесь нужен специализированный инструментарий.

Специализированные средства разработки. Реакцией на неблагополучное состояние дел в этой сфере явилось то, что в мае 1995 года VITA инициировала проектирование стандарта на унифицированную программную среду встраиваемых систем (ESSE). Необходимость в таком стандарте назрела давно, поскольку его отсутствие привело к кризису в разработке встраиваемых систем: и пользователи и производители много времени тратят на инсталляцию драйверов, переписывание программ и разработку новых инструментальных интерфейсов для каждой встраиваемой системы. Первоначально исследования намечались по трем

направлениям: инструментальные средства, прикладные пользовательские интерфейсы ОС (OSAPI) и драйверы В/В. В апреле 1996 года была образована еще и новая группа по двоичному интерфейсу встраиваемых приложений (EABI). Что же имеется на сегодняшний день, кроме проектов?

Стандарт программирования контроллеров. Использование даже простых, но полномасштабных компьютерных конфигураций на нижних уровнях промышленных систем для непосредственного управления оборудованием невыгодно как по экономическим соображениям, так и по причине избыточной сложности. В системах автоматизации для этого традиционно применяются более простые и дешевые устройства – программируемые логические контроллеры. В первом поколении ПЛК представляли собой релейные схемы, вырабатывающие сигналы для оборудования по правилам булевой логики. Сегодня они стали более интеллектуальными: например, устройства типа Smart I/O имеют конфигурацию, в которой центральный процессор на базе дешевого микропроцессора MC68302, последовательные порты и DC/DC-преобразователь собраны в одном компактном промышленном кожухе. Модульная структура Smart I/O позволяет гибко изменять конфигурацию, сокращать и наращивать число каналов ввода/вывода за счет широкой номенклатуры производимых модулей.

Программирование ПЛК может осуществляться двумя способами. В качестве инструментальной системы можно использовать мощные системы разработки типа Dev Pak для OS-9 или кросс-системы, имеющиеся практически на любых современных компьютерных платформах: Unibridge (Unix), PCbridge (PC), FasTrack (Unix, DOS, Windows). В составе этих систем поставляется большое количество драйверов ввода/вывода, и прикладное ПО для ПЛК становится мобильным. Все вроде бы хорошо, однако, во-первых, мобильность ограничена рамками одной инструментальной системы и, во-вторых, программирование

ПЛК таким способом нетрадиционно и в общем неадекватно: требуется знание ОС и языков программирования.

Радикально изменить ситуацию мог стандарт на языки программирования ПЛК, процесс разработки которого начался в 1979 году, и только к 1992 году он был утвержден как IEC 1131-3. При его разработке было обнаружено так много вариаций языков контроллеров, что оказалось невозможно выбрать какой-то один как базовый. Поэтому был предложен совершенно новый язык с применением современных принципов структурного программирования, абстрактных типов данных, выделения данных и процедур в блок. Однако был сохранен и графический стиль классических языков для программируемых контроллеров. В обоих вариантах стандарта введена абстракция управления, и это можно считать главным достижением. Разработчик имеет дело с переменными состояния, не зависящими от типа контроллера способами их обработки, а реальный В/В вынесен на уровень драйверов.

Стандарт IEC 1131-3 [8] описывает два графических языка, "Диаграмма цепей" (LD) и "Диаграмма функциональных блоков" (FBD). В этих языках стандартные символы обеспечивают прямое соответствие между графическим представлением задачи и способом ее решения. В LD используется стандартизированный набор символов для ступенчатого программирования. По существу, здесь разработчик просто составляет релейные схемы. FBD – это тоже графический язык, но его элементами являются функциональные блоки, соединяемые проводами в электрическую цепь, а сами функциональные блоки – это программные объекты, реализующие функции управления.

В дополнение к графическим языкам LD и FBD стандарт IEC 1131-3 определяет язык "Схем последовательных функций" (SFC). Этот язык уже ближе к традиционному программированию и предназначен для записи алгоритмов последовательного управления. Элементы этого языка – шаги, переходы и блоки – используются для определения порядка операций, написанных на любом языке стандарта.

В IEC 1131-3 определяются также два текстовых языка: "Список команд" (IL) и "Структурированный текст" (ST). IL – это язык низкого уровня, в то время как ST поддерживает структурное программирование.

В стандарте специфицированы механизмы, посредством которых производители и пользователи могут определять новые типы данных, функции и функциональные блоки. Таким образом, данный стандарт является саморасширяющимся, и можно надеяться, что он будет в состоянии обслуживать много поколений новых технологий управления. В IEC 1131-3 тщательно описываются механизмы инкапсуляции данных и операций. Например, если пользователь хочет многократно применять одну и ту же последовательность функций управления, он может выделить ее в функциональный блок, поместить его в библиотеку, а затем устанавливать копии этого блока там, где он требуется.

Все языки, включенные в стандарт IEC 1131-3, можно комбинировать, а также включать в программу фрагменты на традиционных языках. Полная реализация IEC 1131-3 доступна как коммерческий продукт: например, это ISaGRAF для Windows производства фирмы CJ International. При использовании в сочетании с OS-9 ядро ISaGRAF выполняется как пользовательская задача и принимает управление загруженным в ПЛК приложением. Высокую оценку специалистов по промышленной автоматизации получил также пакет IOWorks (VMIC).

Будущее стандарта IEC 1131-3 связывается со стандартизацией форматов программных файлов, что обеспечит возможность обмена пакетами функциональных блоков между различными платформами и позволит реально перейти к модульному программированию – созданию больших систем из готовых пакетов. Второе направление развития – использование в распределенных управляющих системах функциональных блоков за пределами программируемых контроллеров. С этой целью был создан стандарт IEC TC65/WG6, в котором концепции IEC 1131-3

применены к стандарту Fieldbus, который определяет, каким образом средства связи могут объединяться с прикладным ПО.

Инструментарий ПО для мониторинга и управления. На уровне ПО мониторинга и управления (SCADA) существенное место занимает интерфейс человек-компьютер (MMI). Процесс разработки приложений SCADA обычно делится на две части. Первая включает проектирование и реализацию аппаратуры и ее логики, обычно генерируемой с помощью языка программирования ПЛК. Вторая часть – создание пользовательского интерфейса. Очень часто эти два этапа выполняются последовательно разными специалистами, имеющими соответствующую подготовку. Поскольку решаемые ими задачи перекрываются, трудоемкость разработки возрастает многократно.

Между тем абстракции управления, которые вводит стандарт IEC 1131, позволяют скомбинировать обе части в единый процесс, если переменные из программ логического управления сделать доступными из ПО SCADA, и наоборот. Примерами интеграции редакторов программ ПЛК в стандарте IEC 1131 с ПО разработки интерфейса пользователя могут служить система WizPLC (PC Soft Int. + Smart Software Solutions) и надстройка компании Ci Technologies над пакетом логического программирования IEC 1131-3 ISaGRAF.

Рассмотрим более подробно пакет InTouch (Wonderware, США), который был признан лучшим инструментальным средством для разработки SCADA-систем на выставке-ярмарке в Ганновере. InTouch реализован для среды Windows NT: управление окнами, работа со шрифтами, механизм межзадачного интерфейса (DDE) – все это базируется на штатных средствах API Windows, что создает привычную для пользователей ПК обстановку.

Основная задача, которую решает InTouch, – разработка графического интерфейса к переменным (текстовым, дискретным, действительным и целым). Переменные определяет разработчик, и они могут быть двух категорий: DDE (для связи с внешними объектами) либо Memory (внутренние). Кроме значения,

переменная имеет набор атрибутов: наличие предупредительного сообщения, вызванного выходом значения за границы установок, признак квитированности этого сообщения, групповая принадлежность переменной, комментарий и т. д. Переменную типа DDE можно связать с данными, поступающими от внешних устройств, либо с объектами других пакетов, например ячейкой Excel.

При создании интерфейса переменной ставится в соответствие графический образ, который размещается в рабочем окне и визуализирует значение переменной, ее атрибуты, например предупредительное сообщение. Графический образ может быть составным, и в него могут входить элементы для изменения значений переменной. В этом плане графический образ соответствует диалоговым панелям Windows. Между переменной и графическим образом устанавливаются анимационные связи, изменяющие внешний вид образа в зависимости от значения переменной. Может происходить изменение размера, цвета, положения на экране, мигание, вращение. В составе InTouch поставляется постоянно расширяемая библиотека графических образов: панели, лампочки, тренды, измерительные линейки, часы, переключатели, клавиши. Благодаря разнообразию типов графических образов визуализация данных возможна либо в числовой форме, либо в виде графика (тренда), изменяющегося в реальном времени.

Поскольку InTouch – инструментальная система, все вводимые разработчиком переменные заносятся в базу данных, которая в целевой системе начинает работать как БД РВ. Установив связи через DDE-интерфейс между переменной InTouch и переменной любого программного пакета, можно сохранять и обрабатывать данные InTouch в стандартной БД или электронной таблице.

Фирмой Wonderware разработан специальный протокол (NetDDE) для сетевого расширения DDE, который позволяет взаимодействовать любым прикладным программам (не

обязательно InTouch) в разных узлах сети. Использовать этот механизм в InTouch очень просто: к собственному имени DDE-переменной добавляется имя узла, в котором она определена.

Контроль за состоянием внешней среды формализуется в InTouch понятием предупредительного сообщения. Они могут генерироваться различными способами: поступать от внешних источников (например контроллеров), возникать при выходе значений за установки или при изменении значения дискретной переменной. InTouch поддерживает многоуровневую структуру приоритетов предупредительных сообщений. Для их обработки можно воспользоваться стандартной функцией квитирования либо написать собственную программу.

Кроме того, InTouch содержит много полезных вспомогательных функций: систему контроля доступа пользователей, генератор отчетов, драйверы для ПЛК и промышленных сетей, статистическую обработку информации, поддержку SQL-доступа, загрузку в устройства нижнего уровня.

Глава 3. КОМПЛЕКС ЛАБОРАТОРНЫХ РАБОТ В СРЕДЕ ОС QNX

§1. Лабораторная работа № 1 «Инсталляция QNX Momentics»

Системные требования.
Рекомендуемый минимум аппаратных мощностей

Элементы архитектуры	Минимум	Рекомендовано
Процессор	2 GHz or more Intel Pentium 4	2 GHz or more Intel Pentium 4
ОП	512 MB	1 GB
Место на ЖД	2.2 GB	2.2 GB
Монитор	1024×768	1280×1024

Для установки QNX 6.4.1 необходимо 1200 MB свободного места на жестком диске, включая место под временные файлы и файлы восстановления при сбоях [3,4,7].

ОС QNX инсталлируется в собственный раздел диска.

Шаг 1. Скачивание установочного образа

Зайти на сайт www.qnx.com

В меню под названием Downloads выбираем «QNX Software Development Platform 6.4.x».

На новой странице находим iso образ под названием «QNX® Software Development Platform 6.4.1 — QNX Neutrino RTOS Installation and Boot CD» размером 627.78MB. Я выбрал тот образ, так как он поддерживает множество архитектур (ARM, MIPS, PPC, SH4, x86). И нажимаем по кнопке «Download Now» (рис. 1).

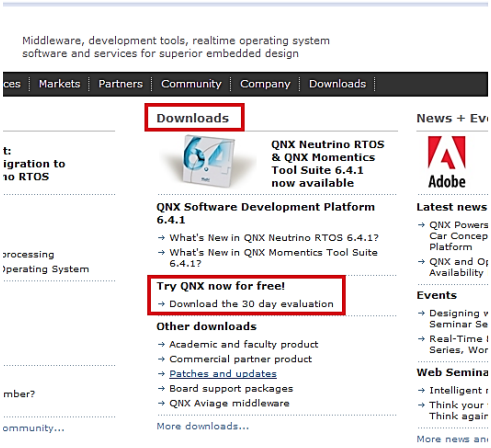
Шаг 2. Получение лицензионного ключа

Вернемся на главную страницу сайта www.qnx.com. В среднем столбце под названием «Downloads» находим строку «Try QNX now for free!» (рис. 2), нажимаем на «Download the 30 day evaluation». На открытой странице нажимаем на изображение Get your license key» (рис. 3).

Браузер перейдет на часть страницы, где находится регистрация. Здесь необходимо заполнить все поля. В поле «If you are a commercial organization and would like some technical assistance, please indicate here.» галочку ставить не нужно, так как мы не коммерческая организация и используем QNX Neutrino исключительно в ознакомительных и учебных целях.

После необходимо ознакомиться с лицензией, и если мы согласны с условиями, поставить галочку в графе «**I have read, understand and agree to the terms of the Non-Commercial End User License Agreement above.**» (*Я прочитал, понял и согласен с условиями некоммерческой пользовательской лицензии, находящейся выше*). Нажимаем на кнопку Submit и ждем письмо в электронном почтовом ящике, который указали при регистрации.

В полученном письме находится лицензионный ключ, который будет необходимо ввести при установке QNX Neutrino.



Шаг 3. Запись образа на диск

После того как файл загрузится, необходимо записать образ с помощью программ для записи дисков (Например Nero). Рекомендуется записывать на минимальной скорости для избегания дальнейших проблем с чтением.

Шаг 4. Сохранение важных данных и разделение диска.

Чтобы не потерять важную информацию, рекомендуется сохранить ее копию в безопасное место. После того как данные сохранены, можно приступить к созданию нового раздела. Это необходимо сделать с помощью специальных программ (Acronis, Partition Magic) либо с помощью стандартных средств Windows (Пуск-Панель управления- Администрирование- Управление компьютером- Управление дисками).

Необходимо создать новый раздел размером 4Гб, этого хватит с большим запасом.

Шаг 5. Установка QNX Neutrino

После записи образа на диск приступим к установке.

Необходимо вставить диск в устройство чтения (CD-DVD ROM). И перезагрузить компьютер. Во время запуска компьютера, если в БИОСе CD привод не стоит первым на загрузке, необходимо нажать клавишу выбора загрузки с устройства (F8, F11, F9, везде по-разному) и выбрать загрузку с устройства чтения оптических дисков. Так как в лаборатории загрузка с привода оптических дисков стоит по умолчанию, то нажимать ничего не пришлось. Начнется загрузка с диска.

Если во время загрузки будет написано, что QNX не смог найти ни одного устройства, на которое можно установиться, то необходимо все Sata диски переключить через БИОС в режим AHCI.(Такая проблема возникла у меня дома, в лаборатории все было хорошо).

После недолгого ожидания на экране появится вопрос:

`Please select a boot option. Option F2 is great for testing`

QNX compatibility on new hardware without writing anything to
the hard disk. It can also be used for system recovery.
F2 – Run from CD (Hard Disk filesystems mounted under /fs)
F3 – Install QNX to a new disk partition
Select?

(Выберите, пожалуйста, вариант загрузки. Выбор клавиши <F2> удобен для проверки совместимости QNX с новой аппаратурой без записи какой-либо информации на диск. Этот вариант можно также использовать для восстановления системы после сбоя.

<F2> – загрузить систему непосредственно с компакт-диска (файловые системы монтируются в /fs);

<F3> – установить QNX в новый раздел диска.

Ваш выбор?)

Если нажать клавишу <F2>, то QNX загрузится с компакт-диска. При этом будет предложено изменить настройки видеорежима. Инсталляция QNX выполняться не будет.

Нажмите клавишу <F3>. На экране появится сообщение:

This installation will create a QNX partition on your hard disk and create a bootable QNX Neutrino image. You may abort this installation at any prompt by pressing the F12 key.
Press F1 to continue.
Press F2 to set verbose (debug) mode.
Choice (F1, F2)?

(Во время инсталляции будут созданы раздел на жестком диске и загрузочный образ QNX. Процесс инсталляции можно прервать в любой момент нажатием клавиши <F12>.

Нажмите клавишу <F1> для продолжения инсталляции.

Нажмите клавишу <F2> для перехода в режим подробного вывода информации (отладочный режим).

Ваш выбор (<F1>, <F2>)?)

В дальнейшем мы будем рассматривать инсталляцию в обычном режиме. Чаще всего этого вполне достаточно. Итак, нажмите клавишу <F1>.

В любом случае на экране появится приглашение ввести лицензионный ключ (license key), который указан в лицензионном сертификате, поставляемом с компакт-диском:

Please enter your license key:

Аккуратно напечатайте требуемый номер, который получили по электронной почте, и нажмите клавишу <Enter>.

Если вы ошибетесь, система вежливо сообщит об этом печальном обстоятельстве:

**** Invalid or expired license key entered. ****

Если же введен верный номер, то на экране появится текст лицензии и вам нужно будет снова ознакомиться с лицензией – принять или отвергнуть ее условия. Итак:

F1 - accept F2 - reject

(<F1> – принять условия; <F2> – отвергнуть условия).

Если нажать клавишу <F2>, то инсталляция прекратится, программа- инсталлятор предложит извлечь компакт-диск QNX из дисковода и перезагрузить ЭВМ. Если вы согласны с условиями лицензии, то нажмите клавишу <F1>. На экране появится сообщение:

Please enter the disk your would like to install QNX Neutrino on. The disk must be bootable from your BIOS.

(Выберите, пожалуйста, диск, на который вы бы хотели установить QNX Neutrino. Этот диск должен загружаться с помощью BIOS вашей ЭВМ.) За этой фразой следует список обнаруженных дисков. Каждая строка списка начинается с названия функциональной клавиши, которую нужно нажать для установки ОС на соответствующий диск. В лаборатории только один жесткий, поэтому выбора у меня нет – жму клавишу <F1>. В ответ получаю сообщение:

*** WARNING***

You have a disk which is greater than 8.4 Gigabytes. The original BIOS calls to access the disk are unable to read data above 8.4G. If your BIOS is older then 1998 you may be forced to choose option 2. Newer BIOSes support on extended disk read calls which can access the entice drive. F1 Allow the QNX partition to be anywhere on the disk. F2 Keep the QNX partition below 8.4G. Choice (F1, F2)?

(Объем диска на вашем компьютере превышает 8,4 гигабайта. Обычные функции BIOS, предназначенные для доступа к дискам, не могут читать данные, расположенные за пределами 8,4 Гбайт. Если ваша BIOS произведена до 1998 года, то вам следует нажать клавишу <F2>. Более новые BIOS поддерживают расширенные вызовы чтения, позволяющие работать с любыми дисками.

<F1> – раздел QNX может находиться в любом месте диска;

<F2>– раздел QNX должен находиться в пределах 8,4 Гбайт.

Ваш выбор (<F1>, <F2>)?)

Так как компьютер выпущен после 1998 года, жмем клавишу <F1>.

Снова вопрос:

The disk does not have room for a QNX partition. There is no free partition entry…

F1-…

F2-…

Нам говорят, что не найдено QNX раздела. И все остальные разделы тоже не свободны. Два варианта:

F1- Показать таблицу разделов и выбрать, какой раздел удалить и создать новый, при этом все данные, хранящиеся на разделе, будут удалены.

F2- Прервать установку и создать новый пустой раздел с помощью специальных программ (Например Partition Magic)

Так как мы о специальном разделе заранее позаботились в 4-м шаге, нажимаем F1.

Показаны Клавиши и напротив них имена и размеры разделов, выбираем наш раздел размером 4Гб, нажав соответствующую клавишу (например F2);

Новое сообщение:

Choose Partition Type

Please choose a type for your partition. The default is type 179. Choose this if your are unsure.

You would typically choose an alternative partition if there was data on an existing Type 179 partition which you do not wish to overwrite, or if you wish to use the QNX 4 file system (used in QNX Neutrino 6.2.x and 6.3.x).

Power-Safe file system:

F1 Type 179 (not currently used)

F2 Type 178 (not currently used)

F3 Type 177 (not currently used)

QNX 4 file system:

F4 Type 79 (not currently used)

F5 Type 78 (not currently used)

F6 Type 77 (not currently used)

(Выберите Тип Разделения

Пожалуйста, выберите тип для своего разделения. Стандартный - тип 179. Выберите его, если Вы неуверенны.

Выберите альтернативное разделение, если бы были данные по существующему Типу 179 разделения, которое Вы не желаете переписывать, или если Вы желаете использовать QNX 4 файловых системы (используемый в Нейтрино QNX 6.2.x и 6.3.x)).

Выбираем F1.

Новый вопрос:

Your disk has room for a XXXX (у меня 4096) megabyte QNX partition?

Please select the size of the partition you would like to create for.

F1- Use all available space

F2-Enter a size

F3-Delete a partition

Choice (F1,F2,F3)?

Нажимаем F1, если хотим использовать все свободное место раздела, F2-Ввести размер. Я выбрал все место (F1) и на экране появилось сообщение:

You have more then one partition on your hard disk. You need a special partition boot loader to let you choose which partition to load when you boot. Your choices are:

F1 Install the QNX partition boot loader. When you boot, it will prompt you to select which partition (OS) to boot. If your partition may start above 8.4G, we recommend this choice.

F2 Install the QNX partition boot loader for machines with old BIOS (before 1996/1997). It should not be used with drives greater then 8.4 G.

F3 Use your existing boot loader, which may already provide this capability. Examples include System Commander or LILO. If it does not provide this capability you will be able to boot only the currently set active partition. If your partition starts above 8.4G, this existing loader will need to use the new extended BIOS disk calls.
Choice (F1, F2, F3)?

(На вашем жестком диске больше одного раздела. Чтобы можно было выбирать, какой из них использовать при загрузке, необходим специальный загрузчик. У вас есть варианты: <F1> – установить "родной" загрузчик QNX. Он будет выдавать приглашение выбрать раздел (т. е. ОС) для загрузки. Если раздел QNX находится за пределами 8,4 Гбайт, то мы рекомендуем этот вариант;

<F2> – установить "родной" загрузчик QNX для компьютеров со старой BIOS (до 1996–1997 гг. выпуска). Его нельзя использовать с дисками объемом свыше 8,4 Гбайт.

<F3> – оставить прежний загрузчик. QNX могут загружать "не родные" загрузчики, например System Commander или LILO. Однако если такой загрузчик не установлен, то вы сможете загружать только ОС, установленную в активном на данный момент разделе диска. Если раздел QNX находится за пределами 8,4 Гбайт, то прежний загрузчик должен поддерживать новые вызовы BIOS для дисков. Ваш выбор (<F1>, <F2>, <F3>)?)

Нас вполне устраивает загрузчик QNX, однако некоторых пользователей он не удовлетворяет, т. к. не имеет графического интерфейса. Что ж, такие пользователи могут по экспериментировать с мультизагрузчиками третьих производителей. Вот и все. Программа инсталляции получила информацию, достаточную для установки как самой операционной системы, так и инструментов разработчика. На самом деле, процесс инсталляции может несколько отличаться от описанного выше в зависимости от используемого компьютера (например, на вашей ЭВМ есть несколько жестких дисков, или программа-инсталлятор обнаружила уже готовый раздел QNX). Но в главном последовательность действий сохраняется. Итак, нажимаем клавишу <F1> (установить загрузчик QNX) – на экране идет установка…

`mounting filesystems ...` (монтируются файловые системы ...)

`Copying files to the new QNX partition. Please wait` ... (Файлы копируются в новый раздел QNX. Пожалуйста, подождите …)

В процессе копирования файлов на экране в реальном времени

отображается, сколько процентов файла уже скопировано:

`100% Copying core files to hard disk`

```
    100%  COPY  /cd/boot/fs/qnxbase.ifs  to
/hdisk/boot/fs
    100%  COPY  /cd/boot/fs/qnxbase.ifs  to
/hdisk/.altboot
    100%  COPY  /cd/boot/fs/qnxbasedma.ifs  to
/hdisk/boot/fs
    100%  COPY  /cd/boot/fs/qnxbasedma.ifs  to
/hdisk/.boot
    100%  COPY  /cd/boot/fs/qnxbasesmp.ifs  to
/hdisk/boot/fs
    Recording licenses … complete
```

По окончании копирования файлов на экране появляется сообщение:

QNX Momentics 6.4.1 Development Seat Installation

Throughout installation, you will be prompted for necessary information. The default answers are displayed in brackets after the question. If you press the <Enter> key, the default answer will be used. After typing your answer, press the <Enter> key to continue the installation.

(Инсталляция инструментальной среды QNX Momentics 6.4.1. В ходе инсталляции у вас будет запрошена необходимая информация. Ответы по умолчанию показаны в квадратных скобках после вопроса. Если вы нажмете клавишу <Enter>, то будет использован ответ по умолчанию. Напечатав ответ, нажмите клавишу <Enter> для продолжения инсталляции.) Затем система начнет задавать эти самые вопросы:

```
    Do you wish to install to the default
location of `/usr/qnx641` (y/n)? [y]
```

(Хотите ли вы выполнить установку в назначенный по умолчанию каталог /usr/qnx641 (да/нет)? [да])

Соглашаюсь (нажимаю клавишу <Enter>).

Затем инсталлятор сообщает, что у нас есть возможность инсталлировать для целевых систем пакет программ,

распространяемых под условием лицензии GNU GPL (General Public License, генеральной общедоступной лицензии GNU1:

Do you wish to install the GNU Public License Utility package (У/n)? [y]

(Хотите ли вы установить пакет утилит, распространяемых на условиях лицензии GNU GPL (да/нет)? [да])

Можем согласиться (нажать клавишу <Enter>).

Система сообщит, что она начала инсталляцию выбранных составных частей и предложит немного подождать:

```
Installing qnx-host ... please wait.
Installing qnx-target ... please wait.
Installing qnx-qde ... please wait.
Installing qnx-target-gpl ... please wait.
```

Далее сообщается, в каких файлах содержится информация о том, что же мы установили, и выводится сообщение о завершении инсталляции с просьбой извлечь инсталляционный CD-ROM и нажать любую клавишу для перезагрузки ЭВМ:

** Installation complete **

Please remove the installation media then press 'Enter' to reboot

Вынимаем CD-ROM и нажимаем какую-нибудь клавишу. Начинается первая загрузка QNX Neutrino после установки.

Шаг 6. Запуск QNX

После включения питания ЭВМ, на которой инсталлирована ОСРВ QNX Neutrino, мультизагрузчик предлагает выбрать ОС для загрузки. (Вернее, он предлагает выбрать раздел диска, с которого следует выполнять загрузку.) После инсталляции QNX Neutrino раздел QNX будет активным и загрузится по умолчанию, если ничего не нажимать:

```
Press F1-F4 for select drive or select partition 1,2,3,4? 1
```

(Нажмите клавишу <F1>, <F2>, <F3> или <F4> для выбора дисковода или выберите раздел 1, 2, 3 или 4? 1)

Единица после вопросительного знака – это вариант, предлагаемый по умолчанию.

Следует заметить, что в QNX обычно используется два загрузочных образа: основной образ помещается в файл /.boot, а резервный – в файл /.altboot. Загрузчик предлагает нажать клавишу <Esc> для загрузки резервного образа операционной системы:

Hit Esc for .altboot

Если ничего не нажимать, то загружаться будет основной образ. Устанавливаемая с компакт-диска QNX Neutrino сконфигурирована так, что после загрузки образа и инициализации микроядра запускается утилита diskboot, выполняющая значительную часть работы по загрузке системы. Эта утилита предложит следующее:

Press the space bar to input boot options or D to disable DMA...

(Нажмите клавишу <пробел> для ввода опций загрузки или клавишу <D>, чтобы запретить DMA)

Если ничего не нажимать, загрузка продолжится в автоматическом режиме. Если нажать клавишу <пробел>, то на экране появится сообщение:

```
    F1 Safe modes
    F5 Start a debug shell after mounting filesystems
    F6 Be Verbose
    F7 Mount read-only partitions read/write if possible
    F8 Enable a previous package configuration
    F9 Target output to debug device defined in startup code
    F10 Force a partition install
    F11 Enumerator disables
    F12 Driver disables
    Enter Continue boot process
    Please select one or more options via functional keys.
    Selection?
```

(<F1> – загружаться в "безопасном режиме";

<F5> – запустить командный интерпретатор после подключения файловых систем;

<F6> – установить режим подробного вывода диагностических сообщений;

<F7> – попытаться подключить с возможностью записи разделы, доступные только для чтения;

<F8> – вернуться к предыдущей конфигурации пакетов;

<F9> – выводить диагностическую информацию на устройство, указанное в модуле startup;

<F10> – сразу приступить к инсталляции системы;

<F11> – отключить автоматическое распознавание устройств;

<F12> – отключить драйверы;

<Enter> – продолжить загрузку.

Выберите, пожалуйста, одну или более опций, нажав соответствующие функциональные клавиши.

Ваш выбор?)

Нажатие клавиши <F1> позволяет использовать несколько вариантов загрузки системы с ограниченной функциональностью. Опция <F10> имеет смысл только при загрузке с инсталляционного компакт-диска – без этой опции утилита **diskboot**, кроме инсталляции, предложит еще загрузить систему с одного из обнаруженных разделов QNX (на компакт-диске или на жестком диске). Опция <F11> позволяет отключить автоопределение некоторых типов устройств, а <F12> – отключить некоторые типы драйверов. Выяснив, как загружать систему, **diskboot** ищет раздел QNX на жестком диске. Таких разделов, разумеется, может быть несколько. Утилита **diskboot** должна выбрать, какой из этих разделов сделать основным (т. е. "корневым"). Возможность использования раздела в качестве корневого **diskboot** определяет по наличию в этом разделе файла /.diskroot. Если найдено больше одного раздела с файлом /.diskroot, то будет предложено выбрать, какой из этих разделов назначить корневым:

```
    You have more then one .diskroot file which
wants to mount at /
    F1 /dev/hd0t78
    F2 /dev/hd0t79
    Which one do you wish to mount?
```

После этого (речь все еще о первом запуске) система переключается в графический режим и выводит окно для первоначального конфигурирования графического драйвера. Вам необходимо задать четыре параметра:

- видеодрайвер;
- разрешение экрана;
- глубину цвета;
- частоту обновления экрана.

Кроме того, конфигуратор позволяет выбрать – запускать автоматически графическую оболочку Photon microGUI при каждой загрузке системы или нет. Впрочем, в процессе эксплуатации системы эту настройку всегда можно изменить. После выбора нужных значений параметров нажимаем кнопку **Test** (тестовый режим) – это позволит изменить неудачные настройки. Как только получим приемлемое изображение, сразу нажимаем кнопку **Continue** (продолжить). Если в тестовом режиме не нажать эту кнопку, то система автоматически переключится на прежние настройки через 15 с. Заданные значения параметров можно будет снова изменить в любое время после загрузки системы. Затем QNX предложит ввести имя пользователя и пароль. В свежеустановленной системе уже есть несколько зарегистрированных системных псевдопользователей и один вполне реальный – суперпользователь, имеющий неограниченные полномочия в операционной системе. Это вы ☺. Этот пользователь имеет имя root и пока не имеет пароля. Вводим имя, игнорируем запрос на ввод пароля – и (поздравляю!) мы в QNX Neutrino. Инсталляция и первая загрузка операционной системы успешно завершены.

Задание по лабораторной работе № 1 «Инсталляция QNX Momentics»

Используя ресурсы Интернет и доступную литературу, составить реферат на указанную тему.

В реферате необходимо раскрыть следующие вопросы:

• Существующие версии RTOS QNX и среды разработки «под» QNX;

• Алгоритмы установок;

• Различные версии установок

◦ В собственный раздел

◦ В раздел Windows

◦ Создание виртуальной QNX машины

◦ Загрузка с CD – диска

• Подробно (каждый пункт главного меню) описать интерфейс графической оболочки Photon microGUI.

В реферате обязателен список ресурсов (не менее пяти различных сайтов (не документов с одного сайта))

Реферат сдавать только в электронном виде.

Обязательно наличие «скриншотов».

§2. Лабораторная работа №2 «Простейший пример»

2.1. Теория

Минимальный набор действий, необходимый для демонстрации примера программы для QNX:

1. Набрать текст программы, выбрав вариант арифметического выражения согласно списку в журнале преподавателя.

2. Откомпилировать программу.

3. Запустить программу на исполнение.

Текст программы можно набрать во встроенном редакторе, или взять готовый текстовый файл.

Для дальнейших действий желательно сделать текущим каталог, где находится текст программы. Для этого можно воспользоваться командами **# cd <имя директории>** (- сменить текущую директорию на указанную) или **# cd ..** (- подняться на уровень выше). Чтобы просмотреть содержимое директории, можно воспользоваться командой **# ls** .

Чтобы откомпилировать программу, можно воспользоваться встроенным компилятором – GCC. Для этого в командной строке необходимо написать **# gcc <имя_файла>**. Если в тексте программы есть ошибки, то они будут выведены на экран. Если ошибок нет, буден создан файл **a.out** – это и есть исполняемый файл программы. Чтобы его запустить на исполнение, в командной строке необходимо написать **# `pwd`/a.out .**

2.2. Текст программы

```
#include <stdio.h>
// Подключение библиотеки математических функций

int main(void)
{printf("Студент Иванов И.И. \n");

//  Ввод операндов арифметического выражения

// Вычисление арифметического выражения
// Вывод результата
return(1);
}
```

2.3. Последовательность действий

Создаём текстовый файл программы.
Компилируем его и запускаем на исполнение.

2.4. Результаты исполнения программы

```
# cd ..
# ls
.   .lastlogin   .ph    a.out    lab2    lab4
..  .profile     lab1   lab3     lab5
# cd lab1
# ls
.          ..         myfirst.c
# gcc myfirst.c
# ls
.          ..         a.out      myfirst.c
# `pwd`/a.out
Иванов И.И.

-0,00456
#
```

2.5. Варианты арифметических выражений

1) $y = \frac{\sqrt[3]{(x^2-\sin a)}\times(a^2-b^3)}{\tan(a-b^2)\times(x-b^2)}$

2) $y = \frac{\sqrt[3]{(a^2-\sin x)}\times(x-b^2)}{\tan(x-a^2)\times(a-b^3)}$

3) $y = \frac{\ln(c-d^2)\times\sqrt{(x+c)}}{\sin(x-c)\times(c+d)^3}$

4) $y = \frac{\sqrt[3]{(x^2+\cos(b-\operatorname{ctg} x)}\times(a^2-b^3)}{\operatorname{tg}(a+b^2)\times(x-b^2)}$

5) $y = \frac{\sqrt[3]{(x^2+\sin a)}\times(a+b^3)}{\tan(a-b^2)\times(x-b^2)}$

6) $y = \frac{\sqrt[3]{(x^2-\sin c)}\times(c^2-d)}{\tan(c-d^2)\times(x-c^2)}$

7) $y = \frac{\sqrt[3]{(x^2-\cos(b-\operatorname{ctg} x)}\times(a^2-b^3)}{\operatorname{tg}(a-b^2)\times(x-b^2)}$

8) $y = \frac{\ln(a-b^2)\times\sqrt{(x+b)}}{\sin(x-a)\times(a-b)^3}$

9) $y = \frac{\sqrt[3]{(x^2 - \sin a)} \times (a^2 - b^3)}{\tan(a - b^2) \times (x - b^2)}$

10) $y = \frac{\sqrt[3]{(x^2 - \cos(b - \operatorname{ctg} x)} \times (a - b^2)}{\operatorname{tg}(a + b^2) \times (x + b^2)}$

11) $y = \frac{\sqrt[3]{(x^2 - \cos(c + \operatorname{tg} x)} \times (d^2 - c^3)}{\operatorname{ctg}(c - d^2) \times (x - c^2)}$

12) $y = \frac{\ln(c - d^2) \times \sqrt{(x - d)}}{\sin(x - d) \times (c + d)^3}$

§3. Лабораторная работа №3 «Процессы и потоки»

3.1. Теория

Процессы и потоки

На самом высоком уровне абстракции система состоит из множества процессов. Каждый процесс ответственен за обеспечение служебных функций определенного характера, независимо от того, является ли он элементом файловой системы, драйвером дисплея, модулем сбора данных, модулем управления или чем-либо еще.

В пределах каждого процесса может быть множество потоков. Число потоков варьируется. Один разработчик ПО, используя только единственный поток, может реализовать те же самые функциональные возможности, что и другой, использующий пять потоков. Некоторые задачи сами по себе приводят к многопоточности и дают относительно простые решения, другие, в силу своей природы, являются однопоточными, и свести их к многопоточной реализации достаточно трудно.

Почему процессы?

Почему же не взять просто один процесс с множеством потоков? В то время как некоторые операционные системы вынуждают вас программировать только в таком варианте, возникает ряд преимуществ при разделении объектов на множество процессов:

• возможность декомпозиции задачи и модульной организации решения;

• удобство сопровождения;

• надежность.

Концепция разделения задачи на части, т. е. на несколько независимых задач, является очень мощной. И именно такая концепция лежит в основе QNX. Операционная система QNX состоит из множества независимых модулей, каждый из которых наделен некоторой зоной ответственности. Эти модули независимы и реализованы в отдельных процессах. Единственная возможность установить зависимость этих модулей друг от друга — наладить между ними информационную связь с помощью небольшого количества строго определенных интерфейсов.

Это, естественно, ведет к упрощению сопровождения программных продуктов, благодаря незначительному числу взаимосвязей. Поскольку каждый модуль четко определен, устранять неисправности в одном таком модуле будет гораздо проще - тем более, что он не связан с другими.

Запуск процесса

Теперь обратим внимание на функции, предназначенные для работы с потоками и процессами. Любой поток может осуществить запуск процесса; единственные налагаемые здесь ограничения вытекают из основных принципов защиты (правила доступа к файлу, ограничения на привилегии и т. д.).

Запуск процесса из командной строки

Например, при запуске процесса из командного интерпретатора вы можете ввести командную строку:

```
$ program1
```

Это указание предписывает командному интерпретатору запустить программу program1 и ждать завершения ее работы. Или вы могли набрать:

```
$ program2  &
```

Это указание предписывает командному интерпретатору запустить программу program2 без ожидания ее завершения. В таком случае говорят, что программа program2 работает в фоновом режиме.

Если вы пожелаете скорректировать приоритет программы до ее запуска, вы можете применить команду nice – точно так же, как в Unix:

$ nice program3

Запуск процесса из программы

Нас обычно не заботит тот факт, что командный интерпретатор создает процессы – это просто подразумевается. В некоторых прикладных задачах можно положиться на сценарии командного интерпретатора (пакеты команд, записанные в файл), которые сделают эту работу за вас, но в ряде других случаев вы пожелаете создавать процессы самостоятельно.

Например, в большой мультипроцессорной системе вы можете пожелать, чтобы одна главная программа выполнила запуск всех других процессов вашего приложения на основании некоторого конфигурационного файла. Другим примером может служить необходимость запуска процессов по некоторому событию.

Рассмотрим некоторые из функций, которые обеспечивает запуск других процессов (или подмены одного процесса другим):

· system();

· семейство функций exec();

· семейство функций spawn();

· fork();

· vfork().

Какую из этих функций применять, зависит от двух требований: переносимости и функциональности. Как обычно, между этими двумя требованиями возможен компромисс.

Обычно при всех запросах на создание нового процесса происходит следующее. Поток в первоначальном процессе вызывает одну из вышеприведенных функций. В конечном итоге

функция заставит администратор процессов создать адресное пространство для нового процесса. Затем ядро выполнит запуск потока в новом процессе. Этот поток выполнит несколько инструкций и вызовет функцию main().

Запуск потока

Теперь, когда мы знаем, как запустить другой процесс, давайте рассмотрим, как осуществить запуск другого потока.

Любой поток может создать другой поток в том же самом процессе; на это не налагается никаких ограничений (за исключением объема памяти, конечно!) Наиболее общий путь реализации этого — использование вызова функций pthread_create():

#include <pthread.h>

int int

pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);

Функция pthread_create() имеет четыре аргумента :

· thread - указатель на pthread_t, где хранится идентификатор потока;

· attr - атрибутная запись;

· start_routine - подпрограмма, с которой начинается поток;

· arg - параметр, который передается подпрограмме start_routine.

Отметим, что указатель thread и атрибутная запись (attr) — необязательные элементы, вы можете передавать вместо них NULL.

Параметр thread может использоваться для хранения идентификатора вновь создаваемого потока. Обратите внимание, что в примерах, приведенных ниже, мы передадим NULL, обозначив этим, что мы не заботимся о том, какой идентификатор будет иметь вновь создаваемый поток.

Если бы нам было до этого дело, мы бы сделали так:

pthread_t tid;

pthread_create (&tid, ...

printf («Новый поток имеет идентификатор %d\n», tid);

Такое применение совершенно типично, потому что вам часто может потребоваться знать, какой поток выполняет какой участок кода.

Небольшой тонкий момент. Новый поток может начать работать еще до присвоения значения параметру tid. Это означает, что вы должны внимательно относиться к использованию tid в качестве глобальной переменной. В примере, приведенном выше, все будет корректно, потому что вызов pthread_create() отработал до использования tid, что означает, что на момент использования tid имел корректное значение.

Новый поток начинает выполнение с функции start_routine (), с параметром arg.

Атрибутная запись потока

Когда вы осуществляете запуск нового потока, он может следовать ряду четко определенных установок по умолчанию, или же вы можете явно задать его характеристики.

Прежде, чем мы перейдем к обсуждению задания атрибутов потока, рассмотрим тип данных

Синхронизация

Самый простой метод синхронизации — это «присоединение» (joining) потоков. Реально это действие означает ожидание завершения.

Присоединение выполняется одним потоком, ждущим завершения другого потока. Ждущий поток вызывает pthreadjoin():

#include <pthread.h>

int

pthread_join (pthread_t thread, void **value_ptr);

Функции pthreadjoin() передается идентификатор потока, к которому вы желаете присоединиться, а также необязательный аргумент value_ptr, который может быть использован для

сохранения возвращаемого присоединяемым потоком значения. (Вы можете передать вместо этого параметра NULL).

Где нам брать идентификатор потока?

В функции pthread_create() в качестве первого аргумента указатель на pthread_t. Там и будет сохранен идентификатор вновь созданного потока.

3.2. Текст программы

```
#include <stdio.h>
#include <pthread.h>
#include <sys/neutrino.h>

pthread_t thread_id1;
pthread_t thread_id2;

void * long_thread1(void *notused)
{
 int n;
    for(n=0;n<5;n++)
    {
      printf("Eto pervii potok , TID %d -  N povtora %d \n", thread_id1, n );
    sleep(2);
    }
}

void * long_thread2(void *notused)
{
 int m;
   for(m=0; m<5; m++)
    {
     printf("Eto vtoroi potok , TID %d -  N povtora %d \n", thread_id2 , m );
```

```
    sleep(1);
    }
}

int main(void)
{

printf("Prog threads  PID %d \n",getpid());

pthread_create(&thread_id1, NULL, long_thread1, NULL);
pthread_create(&thread_id2, NULL, long_thread2, NULL);

sleep(40);

return(1);
}
```

3.3. Последовательность действий

В программе создаются и запускаются на исполнение два потока. Когда один поток приостанавливается, сразу начинает работу другой. Приостановка реализована функцией sleep(n), которая останавливает процесс на n секунд. На экране можно наблюдать, как по очереди работают два процесса.

3.4. Результаты выполнения программы

```
# gcc pthread.c
# `pwd`a.out
Prog threads  PID 852000
Etot pervii potok , TID 0 -  N povtora 0
Etot vtoroi potok , TID 0 -  N povtora 0
Etot vtoroi potok , TID 3 -  N povtora 1
```

```
Etot pervii potok , TID 2 -  N povtora 1
Etot vtoroi potok , TID 3 -  N povtora 2
Etot vtoroi potok , TID 3 -  N povtora 3
Etot pervii potok , TID 2 -  N povtora 2
Etot vtoroi potok , TID 3 -  N povtora 4
Etot pervii potok , TID 2 -  N povtora 3
Etot pervii potok , TID 2 -  N povtora 4
#
```

§4. Лабораторная работа №4 «Обмен сообщениями»

4.1. Теория

Архитектура и структура обмена сообщениями

Три ключевые выражения:

- «Клиент посылает (sends) сообщение серверу»;
- «Сервер принимает (receives) сообщение от клиента»;
- «Сервер отвечает (replies) клиенту».

Эти выражения в точности соответствуют действительным именам функций, которые используются для передачи сообщений в QNX/Neutrino.

Минимальный полезный набор функций включает в себя функции ChannelCreateQ, ConnectAttach(), MsgReplyQ, MsgSend() и MsgRecievef.

Разобьем обсуждение на две части: отдельно обсудим функции, которые применяются на стороне клиента, и отдельно — те, что применяются на стороне сервера.

Клиент

Клиент, который желает послать запрос серверу, блокируется до тех пор, пока сервер не завершит обработку запроса. Затем, после завершения сервером обработки запроса, клиент разблокируется, чтобы принять «ответ».

Это подразумевает обеспечение двух условий: клиент должен «уметь» сначала установить соединение с сервером, а потом обмениваться с ним данными с помощью сообщений — как в одну сторону (запрос — «send»), так и в другую (ответ — «reply»).

Установление соединения

Первое, что должны сделать — это установить соединение с помощью функции ConnectAttach(), описанной следующим образом:

```
#include <sys/neutrino.h>
int ConnectAttach
(int nd,pid_t pid, int chid, unsigned index, int flags);
```

Функции ConnectAttach() передаются три идентификатор:

- nd – дескриптор узла (Node Descriptor),
- pid – идентификатор процесса (process ID),
- chid – идентификатор канала (channel ID).

Вместе эти три идентификатора, которые обычно записываются в виде «ND/PID/CHID», однозначно идентифицируют сервер, с которым клиент желает соединиться. Аргументы index и flags мы здесь просто проигнорируем (установим их в ноль).

Итак, предположим, что мы хотим подсоединиться к процессу, находящемуся на нашем узле и имеющему идентификатор 77, по каналу с идентификатором 1. Ниже приведен пример программы для выполнения этого:

```
int coid;
coid = ConnectAttach (0, 77 „ 1, 0, 0);
```

Можно видеть, что присвоением идентификатору узла (nd) нулевого значения мы сообщаем ядру о том, что мы желаем установить соединение на локальном узле.

Соединиться надо с процессом 77 и по каналу 1.

С этого момента есть идентификатор соединения – небольшое целое число, которое однозначно идентифицирует соединение моего клиента с конкретным сервером по заданному каналу.

Теперь можно применять этот идентификатор для отправки запросов серверу сколько угодно раз. Выполнив все, для чего предназначалось соединение, его можно уничтожить с помощью функции:

ConnectDetach (coid);

Передача сообщений (sending)

Передача сообщения со стороны клиента осуществляется применением функции MsgSend(). Мы рассмотрим это на примере:

#include <sys/neutrino.h>

int MsgSend (int coid,

const void *smsg, int sbytes, void *rmsg, int rbytes);

Аргументами функции MsgSendQ являются:

- идентификатор соединения с целевым сервером (coid);
- указатель на передаваемое сообщение (smsg);
- размер передаваемого сообщения (sbytes);
- указатель на буфер для ответного сообщения (rmsg);
- размер ответного сообщения (rbytes).

Передадим сообщение процессу с идентификатором 77 по каналу 1:

```
#include <sys/neutrino.h>
char *smsg = «Это буфер вывода»; char rmsg [200]; int coid;
// Установить соединение
coid = ConnectAttach (0, 77, 1, 0, 0);
if (coid == -1) {
fprintf (stderr, «Ошибка ConnectAttach к 0/77/1!\n»);
perror (NULL);
exit (EXIT_FAILURE);

// Отправить сообщение
if(MsgSend(coid, smsg,strlen (smsg) + 1,rmsg,sizeof(rmsg)) == -1)
```

```
{ fprintf (stderr, «Ошибка MsgSendXn») ;
 perror (NULL) ;
exit (EXIT_FAILURE) ;
if (strlen (rmsg) > 0)
 {
printf («Процесс с ID 77 возвратил \«%s\»\n», rmsg);
}
```

Предположим, что процесс с идентификатором 77 был действительно активным сервером, ожидающим сообщение именно такого формата по каналу с идентификатором 1. После приема сообщения сервер обрабатывает его и в некоторый момент времени выдает ответ с результатами обработки. В этот момент функция MsgSendQ должна возвратить ноль (0), указывая этим, что все прошло успешно. Если бы сервер послал нам в ответ какие-то данные, мы смогли бы вывести их на экран с помощью последней строки в программе.

Сервер. Создание канала

Сервер должен создать канал — то, к чему присоединялся клиент, когда вызывал функцию ConnectAttach(). Обычно сервер, однажды создав канал, приберегает его «впрок».

Канал создается с помощью функции ChannelCreate() и уничтожается с помощью функции ChannelDestroyQ:

```
#include <sys/neutrino.h>
int ChannelCreate (unsigned flags);
int ChannelDestroy (int chid);
```

Таким образом, для создания канала сервер должен сделать так:

```
int  chid;
chid = ChannelCreate (0);
```

Теперь у нас есть канал. В этом пункте клиенты могут подсоединиться (с помощью функции ConnectAttachQ) к этому каналу и начать передачу сообщений:

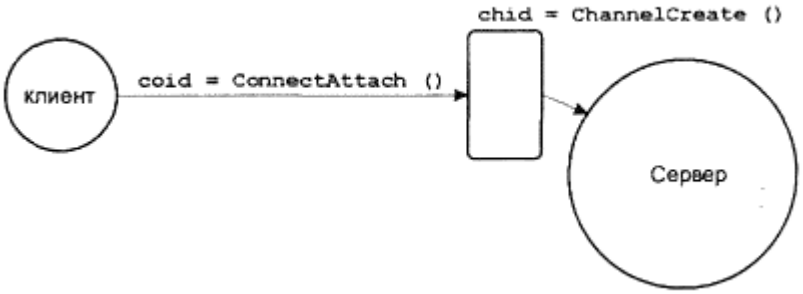


Рис. 3.1. Связь между каналом сервера и клиентским соединением

Обработка сообщений

В терминах обмена сообщениями, сервер отрабатывает схему обмена
в два этапа:

- этап «приема» (receive);
- этап «ответа» (reply).

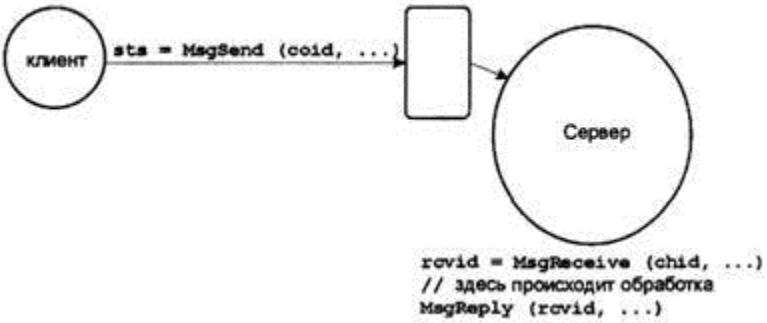


Рис. 3.2. Взаимосвязь функций клиента и сервера
при обмене сообщениями

Обсудим два простейших варианта соответствующих функций, MsgReceiveQ и MsgReply().

```
#include <sys/neutrino.h>
int MsgReceive (int chid,void *rmsg, int rbytes, struct _msg_info *info);
int MsgReply (int rcvid, int status, const void *msg, int nbytes);
```

Четыре основных элемента:

1. Клиент вызывает функцию MsgSend() и указывает ей на буфер передачи (указателем smsg и длиной sbytes). Данные передаются в буфер функции MsgReceiveQ на стороне сервера, по адресу rmsg и длиной rbytes. Клиент блокируется.

2. Функция MsgReceiveQ сервера разблокируется и возвращает идентификатор отправителя rcvid, который будет впоследствии использован для ответа. Теперь сервер может использовать полученные от клиента данные.

3. Сервер завершил обработку сообщения и теперь использует идентификатор отправителя rcvid, полученный от функции MsgReceiveQ, передавая его функции MsgReply(). Заметьте, что местоположение данных для передачи функции MsgReply() задается как указатель на буфер (smsg) определенного размера (sbytes). Ядро передает данные клиенту.

4. Наконец, ядро передает параметр sts, который используется функцией MsgSend() клиента как возвращаемое значение. После этого клиент разблокируется.

Для каждой буферной передачи указываются два размера (в случае запроса от клиента это sbytes на стороне клиента и rbytes на стороне сервера; в случае ответа сервера это sbytes на стороне сервера и rbytes на стороне клиента). Это сделано для того, чтобы разработчики каждого компонента смогли определить размеры своих буферов – из соображений дополнительной безопасности.

4.2. Текст программы

```
// server.c
#include <stdio.h>
#include <pthread.h>
#include <inttypes.h>
#include <errno.h>
#include <sys/neutrino.h>

void server(void )
{
 int rcvid;         //Ykazivaet komy nado otvechat
 int chid;          //Identifikator kanala
 char message[512]; //

 printf("Server start working \n");

chid=ChannelCreate(0);         //Sozdanie Kanala
printf("Chanel id: %d \n", chid);
printf("Pid: %d \n", getpid());
// vipolniaetsa vechno- dlia servera eto normalno
while(1)
  {
  // Polychit i vivesti soobshenie

  rcvid=MsgReceive(chid,message,sizeof(message), NULL);
  printf("Polychili soobshenie, rcvid  %X \n",rcvid  );
  printf("Soobshenie takoe : \"%s\". \n", message  );

  // Podgotovit otvet

  strcpy(message,"Eto otvet");

  MsgReply(rcvid, EOK, message, sizeof(message)) ;
```

```
   printf("\"%s\". \n", message  );
   }
}

int main(void)
{
printf("Prog server  \n");
server();
sleep(5);
return(1);
}

//client.c
#include <stdio.h>
#include <pthread.h>
#include <inttypes.h>
#include <errno.h>
#include <sys/neutrino.h>

int main(void)
{
char smsg[20] ;
char rmsg[200];
int coid;
long serv_pid;
printf("Prog client ,  Vvedite PID servera \n");
scanf ("%ld",&serv_pid);
printf("Vveli %ld  \n", serv_pid);
coid=ConnectAttach(0,serv_pid,1,0,0);
printf("Connect res %d  \n, vvedite soobshenie ", coid);
scanf("%s",&smsg);
printf("Vveli %s  \n", smsg);
if(MsgSend(coid,smsg,strlen(smsg)+1,rmsg, sizeof(rmsg))==-1)
 {
```

```
 printf("Error MsgSend  \n");
}
return(1);
}
```

4.3. Последовательность действий

После компиляции программ сервера и клиента у нас будет 2 исполняемых файла. Назовём их server и client соответственно.

Программу server необходимо запустить в фоновом режиме **# server &**. Она начнёт работать: создаст канал, выведет номер канала и идентификатор процесса сервера, будет ждать сообщения от клиента.

Потом необходимо запустить клиента. Он попросит ввести идентификатор процесса сервера, для установления соединения с ним, и само сообщение (20 символов). Далее можно наблюдать, как сервер получит сообщение, выведет его и пошлёт ответ.

На этом клиент закончит свою работу, но его можно запустить ещё раз и послать другое сообщение.

Остановить работу сервера можно функцией **kill < идентификатор процесса сервера >.**

4.4. Результаты

```
# `pwd`/server &
[3] 2019364
# Prog server
Server start working
Chanel id: 1
Pid: 2019364
# `pwd`/client
Prog client ,  Vvedite PID servera
2019364
Vveli 2019364
```

```
Connect res 3
, vvedite soobshenie Hello_server
Vveli Hello_server
Polychili soobshenie, rcvid   2
Soobshenie takoe : "Hello_server".
"Eto otvet".
# `pwd`/client
Prog client ,  Vvedite PID servera
2019364
Vveli 2019364
Connect res 3
, vvedite soobshenie I_snova_Privet
Vveli I_snova_Privet
Polychili soobshenie, rcvid   2
Soobshenie takoe : "I_snova_Privet".
"Eto otvet".
# kill 2019364
#
```

§5. Лабораторная работа № 5 «Тайм - ауты»

5.1. Теория

Тайм-ауты ядра

QNX/Neutrino позволяет вам получать тайм-ауты по всем блокированным состояниям. Наиболее часто у вас может возникнуть потребность в этом при обмене сообщениями: клиент, посылая сообщение серверу, не желает ждать ответа «вечно». В этом случае было бы удобно использовать тайм-аут ядра. Тайм-ауты ядра также полезны в сочетании с функцией pthreadjoin (): завершения потока тоже не всегда хочется долго ждать.

Ниже приводится декларация для функции TimerTimeout(), которая является системным вызовом, ответственным за формирование тайм-аутов ядра.

```
#include <sys/neutrino.h>
int
TimerTimeout (clockid_t id,
int flags,
const struct sigevent *notify,
const uint64_t *ntime,
uint64 t *otime);
```

Видно, что функция *TimerTimeout()* возвращает целое число (индикатор удачи/неудачи; 0 означает, что все в порядке, -1 — что произошла ошибка и ее код записан в *errno*). Источник синхроимпульсов (CLOCK_REALTIME, и т.п.) указывается в id, параметр *flags* задает соответствующее состояние (или состояния). Параметр notify всегда должен быть событием уведомления типа SIGEV_UNBLOCK; параметр ntime указывает относительное время, спустя которое ядро должно сгенерировать тайм-аут. Параметр otime показывает предыдущее значение тайм-аута и в большинстве случаев не используется (вы можете передать вместо него NULL).

Важно отметить, что тайм-ауты «взводятся» функцией TimerTimeout(), а запускаются по входу в одно из состояний, указанных в параметре flags. Сбрасывается тайм-аут при возврате; из любого системного вызова. Это означает, что вы должны заново «взводить» тайм-аут перед каждым системным вызовом, к которому вы хотите его применить. Сбрасывать тайм-аут после системного вызова не надо — это выполняется автоматически.

Тайм-ауты ядра и функция pthreadjoin()

Самый простой пример для рассмотрения – это использование тайм-аута с функцией pthreadjoin().

Применим макроопределение SIGEV_UNBLOCK_INIT() для инициализации структуры события, но можно было установить sigev_notify в SIGEV_UNBLOCK и «вручную». Можно было даже

сделать еще более изящно, передав NULL вместо struct sigevent – функция TimerTimeoutQ понимает это как знак, что нужно использовать SIGEV_UNBLOCK.

Если поток (заданный, в thread_id) остается работающим более 10 секунд, то системный вызов завершится по тайм-ауту – функция pthreadjoin () возвратится с ошибкой, установив errno в ETIMEDOUT.

Вы можете использовать и другую «стенографию», указав NULL в качестве значения тайм-аута (параметр ntime в декларации выше), что предпишет ядру не блокироваться в данном состоянии. Этот прием можно использовать для организации программного опроса. (Хоть программный опрос и считается дурным тоном, его можно весьма эффективно использовать в случае с pthreadjoin(), периодически проверяя, завершился ли нужный поток. Если нет, можно пока сделать что-нибудь другое.)

Ниже представлен пример программы, в которой демонстрируется неблокирующий вызов pthreadJoinQ:

```
int
pthread_join_nb (int tid, void **rval)
{
TimerTimeout (CLOCK_REALTIME, _NTO_TIMEOUT_JOI
N, NULL, NULL, NULL) ;
return (pthread_join (tid, rval) ) ; )
```

Тайм-ауты ядра при обмене сообщениями

Все становится несколько сложнее, когда вы используете тайм- ауты ядра при обмене сообщениями. На момент отправки клиентом сообщения сервер может как ожидать его, так и нет. Это означает, что клиент может заблокироваться как по передаче (если сервер еще не принял сообщение), так и по ответу (если сервер принял сообщение, но еще не ответил). Основной смысл здесь в том, что вы должны предусмотреть оба блокирующих состояния в параметре flafs функции TimerTimeout(), потому что клиент может оказаться в любом из них.

Чтобы задать несколько состояний, сложите их операцией ИЛИ (OR): TimerTimeout (_NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY).

Это вызовет тайм-аут всякий раз, когда ядро переведет клиента в состояние блокировки по передаче (SEND) или по ответу (REPLY). В тайм-ауте SEND-блокировки нет ничего особенного — сервер еще не принял сообщение, значит, ничего для этого клиента он не делает. Это значит, что если ядро генерирует тайм-аут для SEND-блокированного клиента, сервер об этом информировать не обязательно. Функция MsgSendQ клиента возвратит признак ETIMEDOUT и обработка тайм-аута завершится.

Однако, если сервер уже принял сообщение клиента и клиент желает разблокироваться, для сервера существует два варианта реакции. Если сервер не указал флаг _NTO_CHF_UNBLOCK на канале, по которому было принято сообщение, клиент будет разблокирован немедленно, и сервер не получит об этом никакого оповещения. У большинства серверов флаг _NTO_CHF_ UNBLIOCK всегда установлен. В этом случае ядро посылает серверу импульс, а клиент остается заблокированным до тех пор, пока сервер ему не ответит! Это сделано для того, чтобы сервер мог узнать о запросе клиента на разблокирование и выполнить по этому поводу какие-то действия.

5.2. Текст программы

```
#include <stdio.h>
#include <pthread.h>
#include <inttypes.h>
#include <errno.h>
#include <sys/neutrino.h>

#define SEC_NSEC 1000000000LL // 1 sekynda billion nanosekynd
```

```
void * long_thread(void *notused)
{
  printf("Etot potok vipolnaetsa bolee 10 sekynd \n");
 sleep(20);
}

int main(void)
{
uint64_t   timeout;
struct sigevent event;
int rval;
pthread_t thread_id;

printf("Prog timer  \n");
event.sigev_notify = SIGEV_UNBLOCK;
//SIGEV_UNBLOCK_INIT(&event);
pthread_create(&thread_id, NULL, long_thread, NULL);

timeout = 10LL*SEC_NSEC;
TimerTimeout(CLOCK_REALTIME,
_NTO_TIMEOUT_JOIN,&event, &timeout, NULL);
rval = pthread_join(thread_id, NULL);
  if (rval == ETIMEDOUT)
     {
       printf ("istekli 10 sekynd, potok %d vipolniaetsia!\n",
thread_id);
     }
sleep(5);

TimerTimeout (CLOCK_REALTIME, _NTO_TIMEOUT_JOIN,
&event, & timeout, NULL);
rval = pthread_join(thread_id, NULL);
  if(rval == ETIMEDOUT)
   {
```

```
  printf("potok %d >25 sek!", rthread_id);
  }
else
 {
  printf ("Potok %d zavershon kak nado \n", thread_id);
 }

return(1);
```

5.3. Последовательность действий

Запустить программу на исполнение и сопоставлять то, что она выводит на экран, с текстом программы.

5.4. Результаты

```
# cd ..
# cd lab2
# ls
.          ..          timer.c      timer.exe
# `pwd`/timer.exe
Prog timer
Etot potok vipolnaetsa bolee 10 sekynd
istekli 10 sekynd, potok 2 vipolniaetsia!
Potok 2 zavershon kak nado
#
```

§6. Лабораторная работа № 6 «Синхронизация процессов. Барьеры»

6.1. Теория

Применение барьера

Два метода синхронизации: один метод с применением функции *pthrea. dJoin(),* который мы только что рассмотрели, и метод с применением барьера.

Основной поток должен дождаться того момента, когда все рабочие потоки завершат работу, и только затем можно начинать следующую часть программы.

Однако с применением функции *pthreadJoin ()* мы ожидаем завершения потоков. Это означает, что на момент ее разблокирования потоков нет больше с нами; они закончили работу и завершились.

В случае с барьером мы ждем «встречи» определенного числа потоков у барьера. Когда заданное число потоков достигнуто, мы их все разблокируем (заметьте, что потоки при этом продолжат выполнять свою работу).

Сначала барьер следует создать при помощи функции *barrier_init0:*

```
#include <sync.h>
int
barrier_init (barrier t              *barrier,
const barrier attr t *attr,
int count) ;
```

Эта функция создает объект типа «барьер» по переданному ей адресу (указатель на барьер хранится в параметре *barrier)* и назначает ему атрибуты, которые определены в *attr* (мы будем использовать NULL, чтобы установить значения по умолчанию). Число потоков, которые должны вызывать функцию *barrier_wait(),* передается в параметре *count.*

После того как барьер создан, каждый из потоков должен будет вызвать функцию *barrier_wait(),* чтобы сообщить, что он отработал:

```
#include <sync.h>
int barrier_wait (barrier t *barrier) ;
```

После того как поток вызвал *barrier_wait(),* он будет блокирован до тех пор, пока число потоков, указанное первоначально в параметре *count* функции *barrier_mit(),* не вызовет функцию *barrier'_waU()* (они также будут блокированы). После того как нужное число потоков выполнит вызов функции *barrier_wait(),* все эти потоки будут разблокированы «одновременно».

6.2. Текст программы

```
#include <stdio.h>
#include <time.h>
#include <sync.h>
#include <sys/neutrino.h>

barrier_t barrier;

//int data_ready = 0;
//int inf = 0;
//pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
//pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;

void *thread1 (void * not_used)
{
    time_t now;
    char buf[27];
    time(&now);
    printf("Potok 1, vremia starta %s \n", ctime_r(&now,buf));
    sleep(3);
    barrier_wait(&barrier);
    time(&now);
     printf("barier v potoke 1 , vremia srabativania %s \n", ctime_r(&now,buf));
}
```

```
void *thread2 (void * not_used)
{
    time_t now;
    char buf[27];
    time(&now);
    printf("Potok 2, vremia starta %s \n", ctime_r(&now,buf));
    sleep(6);
    barrier_wait(&barrier);
    time(&now);
     printf("barier v potoke 2 , vremia srabativania %s \n",
ctime_r(&now,buf));
}

main()
{
    time_t now;
    char buf[27];
  barrier_init(&barrier, NULL, 3);
  printf("Start \n");
  pthread_create(NULL,NULL, thread1 ,NULL);
  pthread_create(NULL,NULL, thread2 ,NULL);
    time(&now);
     printf(" Main(): oshidanie y bariera, vremia %s \n",
ctime_r(&now,buf));
    barrier_wait(&barrier);
    time(&now);
     printf("barier v main() , vremia srabativania %s \n",
ctime_r(&now,buf));
sleep(5);
}
```

6.3. Последовательность действий

Основной поток создал объект типа «барьер» и инициализировал его значением счетчика, равным числу потоков (включая себя!), которые должны «встретиться» у барьера, прежде чем он «прорвется». В нашем примере этот индекс был равен 3 — один для потока *main(),* один для потока *thread1 ()* и один для потока *thread2().* Затем, как и прежде, стартуют потоки вычисления графики (в нашем случае это потоки *thread 1()* и *thread2()).* Для примера вместо приведения реальных алгоритмов графических вычислений мы просто временно «усыпили» потоки, указав в них sleep (20) и sleep (40), чтобы имитировать вычисления. Для осуществления синхронизации основной поток *(main())* просто блокирует сам себя на барьере, зная, что барьер будет разблокирован только после того, как рабочие потоки аналогично присоединятся к нему.

Как упоминалось ранее, с функцией *pthreadJoin()* рабочие потоки для синхронизации главного потока с ними должны умереть. В случае же с барьером потоки живут и чувствуют себя вполне хорошо. Фактически, отработав, они просто разблокируются по функции *barrier•_wait().* Тонкость здесь в том, что вы обязаны предусмотреть, что эти потоки должны делать дальше! В нашем примере с графикой мы не дали им никакого задания для них — просто потому что мы так придумали алгоритм. В реальной жизни вы могли бы захотеть, например, продолжить вычисления.

6.4. Результаты

```
# root/a.out
Start
Potok 1, vremia starta Tue Oct 21 00:29:01 2003

Potok 2, vremia starta Tue Oct 21 00:29:01 2003
```

Main(): oshidanie y bariera, vremia Tue Oct 21 00:29:01 2003

barier v potoke 2 , vremia srabativania Tue Oct 21 00:29:07 2003

barier v main() , vremia srabativania Tue Oct 21 00:29:07 2003

barier v potoke 1 , vremia srabativania Tue Oct 21 00:29:07 2003

#/**

§7. Лабораторная работа № 7 «Синхронизация процессов. Условные переменные»

7.1. Теория

Условные переменные

Условные переменные (или «condvars») очень похожи на ждущие блокировки, которые мы рассматривали выше. В действительности, ждущие блокировки — это надстройка над механизмом условных переменных, и именно поэтому в таблице, иллюстрировавшей использование ждущих блокировок, у нас встречалось состояние CONDVAR. Функция *pthread_cond_wait()* точно так же освобождает мутекс, ждет, а затем повторно блокирует мутекс, аналогично функции *pthread_sleepon_wait().*

7.2. Текст программы

```
#include <stdio.h>
#include <pthread.h>

int data_ready = 0;
int inf = 0;
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;

void *consumer (void * notused)
{
        printf("Eto potrebitel \n");

     while(1)
     {
     pthread_mutex_lock (&mutex);
      printf("W1 \n");
      while (!data_ready)
          {

              printf("W2 \n");
                pthread_cond_wait (&condvar, & mutex);
              printf("W3 \n");
        }
     printf("dannie ot proizv = %d \n",inf);
   data_ready=0;
   pthread_cond_signal(&condvar);
     pthread_mutex_unlock(&mutex);
     }
}

void *producer (void * notused)
{
        printf("Eto ptoizvoditel \n");

     while(1)
     {
     sleep(2);
   printf("ptoizvoditel polychil dannie ot h/w = %d \n",inf);
     pthread_mutex_lock (&mutex);
```

```
printf("Wp1 \n");
      while (data_ready)
         {
         printf("Wp2 \n");
        pthread_cond_wait (&condvar, & mutex);

         }
     data_ready=1;
   inf++;
   printf("Wp3 \n");
   pthread_cond_signal(&condvar);
     pthread_mutex_unlock(&mutex);
     }
}

main()
{
 printf("Start \n");
 pthread_create(NULL,NULL, consumer,NULL);
 pthread_create(NULL,NULL, producer,NULL);
sleep(10);
}
```

7.3. Последовательность действий

Этот пример в значительной степени похож на программу с применением ждущей блокировки, с небольшими отличиями. Первое отличие, которое бросается в глаза, — здесь использован новый тип данных, pthread_cond__t. Это просто декларация для условной переменной; мы назвали нашу условную переменную *condvar*.

Следующее, что видно из примера, — это то, что структура «потребителя» идентична таковой в предыдущем примере с ждущей блокировкой.

Основное различие здесь состоит в том, что библиотека ждущих блокировок имеет скрытый внутренний мутекс, а при использовании условных переменных мутекс передается явно. Последний способ дает нам больше гибкости.

И, наконец, обратите внимание на то, что мы использовали функцию *pthread_cond_signal()* (опять же, с явной передачей мутекса).

2.6.4 Результаты

```
# root/a.out
Start
Eto potrebitel
W1
W2
Eto ptoizvoditel
ptoizvoditel polychil dannie ot h/w = 0
Wp1
Wp3
W3
dannie ot proizv = 1
W1
W2
ptoizvoditel polychil dannie ot h/w = 1
Wp1
Wp3
W3
dannie ot proizv = 2
W1
W2
ptoizvoditel polychil dannie ot h/w = 2
Wp1
Wp3
W3
```

dannie ot proizv = 3
W1
W2
ptoizvoditel polychil dannie ot h/w = 3
Wp1
Wp3
W3
dannie ot proizv = 4
W1
W2

§8. Лабораторная работа № 8 «Итоговая. Индивидуальное задание»

Тема: Разработка мультипроцессных приложений.

Цель работы: Закрепление работы с функциями ОС РВ для запуска параллельных процессов и организации межпроцессного взаимодействия посредством сообщений.

Индивидуальное задание выполняется только одно из пяти по номеру студента в журнале преподавателя. Выполнение индивидуального задания учитывается при получении «зачёта – автомата».

ЗАДАНИЕ 1

Разработать приложение, состоящее из трех взаимодействующих процессов. Требуется написать три программных модуля: M1, M2, M3. На базе модуля M1 из shell запускается стартовый процесс P1(M1).

Процесс P1 создает канал и, используя функцию семейства spawn*(), запускает процессы P2(M2) и P3(M3), передавая им в качестве параметра chid созданного канала.

Процесс P2 создает свой канал, устанавливает соединение с каналом процесса P1, отправляет ему сообщение о chid своего канала и переходит в состояние приема сообщений по созданному каналу.

Процесс P3 устанавливает соединение с каналом процесса P1 и посылает ему запрос на получение pid процесса P2 и chid его канала для присоединения к каналу процесса P2. Получив ответ (pid и chid), присоединяется к каналу процесса P2 и посылает сообщение "P2 загружен".

Процесс P2, приняв сообщение от процесса P3, добавляет к нему информацию "P1 принял сообщение от P2" и отправляет сформированное таким образом сообщение процессу P1.

Процесс P1, получив сообщение от P2, выдает его на экран терминала, посылает ответ "P1 OK" процессу P2 и завершает работу (терминируется).

Процесс P2, получив ответ от P1, выдает его на экран терминала, посылает ответ " P2 OK" процессу P3 и завершает работу (терминируется).

Процесс P3, получив ответ от P2, выдает его на экран терминала после чего выдает на терминал "P3 OK" и завершает работу (терминируется).

ЗАДАНИЕ 2

Разработать приложение, состоящее из трех взаимодействующих процессов. Требуется написать три программных модуля: M1, M2, M3. На базе модуля M1 из shell запускается стартовый процесс P1(M1).

Процесс P1 создает канал и, используя функцию семейства spawn*(), запускает процесс P2(M2), передавая ему в качестве параметра chid созданного

канала, и переходит в состояние приема сообщений от P2 по своему каналу.

Процесс P2 создает свой канал и, используя функцию семейства spawn*(), запускает процесс P3(M3), передавая ему в

качестве параметра chid созданного канала. Устанавливает соединение с каналом процесса P1, отправляет ему сообщение о pid процесса P3 и переходит в состояние приема сообщений по созданному каналу от P3.

Процесс P3 создает свой канал, устанавливает соединение с каналом процесса P2 и посылает ему сообщение "P3 загружен". Получив ответ, посылает ему chid своего канала и переходит в состояние приема сообщений по своему каналу.

Процесс P2, приняв первое сообщение, отправляет его процессу P1, получив ответ от P1, принимает chid от процесса P3, посылает ему ответ и передает chid процессу P1. Далее выдает на терминал "P2 загружен" и переходит в состояние приема сообщений по своему каналу от P3.

Процесс P1, получив первое сообщение от P2, выдает его на экран терминала, посылает ответ процессу P2 и принимает второе сообщение, устанавливает соединение с каналом процесса P3 и посылает по нему сообщение "stop", после ответа переходит в ожидание сообщения по своему каналу.

Процесс P3, получив "stop", отправляет его процессу P2, печатает "stop P3" и терминируется.

Процесс P2, получив "stop", отправляет его процессу P1, печатает "stop P2" и терминируется.

Процесс P1, получив "stop", печатает "stop P1" и терминируется.

ЗАДАНИЕ 3

Разработать приложение, состоящее из трех взаимодействующих процессов. Требуется написать три программных модуля: M1, M2, M3. На базе модуля M1 из shell запускается стартовый процесс P1(M1).

Процесс P1 создает канал и, используя функцию семейства spawn*(), запускает процессы P2(M2) и P3(M3), передавая им в качестве параметра chid созданного канала, и переходит в ожидание сообщений по своему каналу.

Процесс P2 создает свой канал, устанавливает соединение с каналом процесса P1, отправляет ему сообщение о chid своего канала и переходит в состояние приема сообщений по созданному каналу.

Процесс P3 создает свой канал, устанавливает соединение с каналом процесса P1, отправляет ему сообщение о chid своего канала и переходит в состояние приема сообщений по созданному каналу.

Процесс P1, приняв сообщение от процесса P2 или P3 о chid канала, устанавливает соединение и посылает по нему - "P1 принял сообщение от P?", принимает ответ и выдает его на терминал. После такого взаимодействия с P2 и P3, P1 терминируется.

Процесс P?, получив сообщение от P1, выдает его на экран терминала, посылает ответ "P? OK" процессу P1 и завершает работу (терминируется).

ЗАДАНИЕ 4

Разработать приложение, состоящее из пяти взаимодействующих процессов. Требуется написать три программных модуля: M1, M2, M3. На базе модуля M1 из shell запускается стартовый процесс P1(M1).

Процесс P1, используя функцию семейства spawn*(), запускает процесс P2(M2) с флагом P_NOWAIT и P3(M2) с флагом P_WAIT.

Процесс P2 создает свой канал и, используя функцию семейства spawn*(), запускает процесс P4(M3), передавая в качестве аргумента chid своего канала, и переходит в состояние приема сообщений по созданному каналу.

Процесс P3 создает свой канал и, используя функцию семейства spawn*(), запускает процесс P5(M3), передавая в качестве аргумента chid своего канала, и переходит в состояние приема сообщений по созданному каналу.

Процесс P?(M3) устанавливает соединение с каналом родительского процесса P?(M2) и посылает сообщение "P?-OK", получив ответ , выдает его на терминал и терминируется

Процесс P3(M2), получив сообщение от P?(M3), выдает его на экран терминала, посылает ответ "P? OK" процессу P?(M3) и завершает работу (терминируется).

Процесс P1, возобновив свою работу, выдает на терминал сообщение "STOP" и терминируется.

ЗАДАНИЕ 5

Разработать приложение, которое строится в виде цепочки процессов. Требуется написать два программных модуля: M1, M2. На базе модуля M1 из shell запускается стартовый процесс P1(M1), которому передается в качестве параметра длина цепочки N (количество процессов).

Процесс P1 создает свой канал и, используя функцию семейства spawn*(), запускает процесс P2(M2), передавая в качестве аргумента длину цепочки N, chid своего канала и переходит в ожидание прихода по нему сообщения, после чего отвечает и терминируется.

Процесс P2(M2) устанавливает соединение с каналом родительского процесса и, если N не равно 0, создает свой канал и, используя функцию семейства spawn*(), запускает следующий процесс P?(M2), передавая ему в качестве аргумента N-1 и chid своего канала, и переходит в состояние приема сообщений по своему каналу. Если N=0, то посылает предыдущему процессу сообщение "P2-OK" и, получив ответ, выдает его на экран и терминируется.

Аналогично ведет себя каждый вновь запущенный процесс цепочки.

Порядок отчета:

1. Проверка знания средств запуска процессов и механизмов канального взаимодействия.

2. Демонстрация работы мультипроцессного приложения.

3.Оформление письменного отчета по результатам выполнения лабораторной работы.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Кёртен, Р. Введение в QNX/Neutrino 2 / Р. Кёртен. – Санкт-Петербург: Петрополис, 2001.

2. Зыль, С.Н. Операционная система реального времени QNX: от теории к практике / С.Н. Зыль. – 2-е изд., перераб. и доп. – Санкт-Петербург: БХВ-Петербург, 2004. – 192 с.

3. Зыль, С.Н. QNX Momentics: основы применения / С.Н. Зыль. – Санкт-Петербург: БХВ-Петербург, 2005. – 256 с. Имеется электронный вариант книги по адресу http://swd.ru/index.php3?pid=499

4. Практика работы с QNX / Алексеев [и др.]. – Москва: Издательский Дом «КомБук», 2004.

5. Сулейманова, А.М. Системы реального времени: учебное пособие / А.М. Сулейманова. – Уфа: Уфим. гос. авиац. техн. ун-т. 2004. – 292 с.

6. Системы реального времени: конспект лекций / сост. А. С. Голубев. – Владимир: Изд-во Владим. гос. ун-та, 2010. – 127 с.

7. QNX Realtime Platform: Русский Портал http://qnx.org.ru/.

8. Христенсен, Д. Знакомство со стандартом на языки программирования PLC: IEC 1131-3 / Д. Христенсен // Мир компьютерной автоматизации. – 1995. – №1.

9. Хухлаев, Е. Операционные системы реального времени и Windows NT / Е. Хухлаев // Открытые системы. – 1997. – №5. – С. 48-51.

Учебное издание

Луканов Александр Сергеевич

СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Учебное пособие

Редактор Т.К. Кретинина
Компьютерная верстка А.В. Ярославцевой

Подписано в печать 07.08.2020. Формат 60×84 1/16.
Бумага офсетная. Печ. л. 9,75.
Тираж 25. Заказ . Арт. – 39(Р1У)/2020.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)
443086, Самара, Московское шоссе, 34.

Издательство Самарского университета.
443086, Самара, Московское шоссе, 34.