Жалал-Абадский государтсвенный университет имени Б.Осмонова Педагогический факультет им.Э.Уметова Кафедра Автоматизированные системы управления

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Ажыкулов
Сапарбек Мажитович

ЖАГУ

СОДЕРЖАНИЕ

Лекция 1. Вводная лекция

План курса "Операционные системы" предполагает рассмотрение следующих тем:

- вводная лекция, которая будет посвящена **истории операционных систем и этапам становления технологии операционных систем**, в том числе сверхновейшей истории, которая касается последних 10-15 лет.

- некоторые технологические вопросы, **технологии, которые используются в операционных системах** и предлагаются для пользователей компьютеров. Когда они появились, что они из себя представляют, что с ними сейчас и т.д.

- **тематика управлению процессами внутри операционных систем**, этой тематике посвящено 6 лекций, потому что этот курс предполагает, что студентов интересует программирование. Тематика процессов крайне интересна и важна для программистов, потому что использование процессов, взаимодействие процессов и пр. - это, с одной стороны, исключительно техника, которая позволяет повышать производительность приложений, а с другой стороны - она опасная, потому что такое многопроцессное программирование стимулирует делать ошибки, очень трудно находимые и неповторяющиеся, которые проявляются крайне редко, их трудно искать.

- **управление памятью**, потому важно то, что касается процессов, а с точки зрения управления ресурсами - процессы тесно связаны с процессорами. Мы живем в то время, когда мультипроцессорные системы стали повседневной реальностью, а процессоры multi-core стоят везде, включая ноутбуки. Лет 30 назад трудно было поверить, что так будет, однако ресурс процессор - процессор, который может выполнять команды - это наиболее дефицитный ресурс до сих пор. Представители компания Huawei заявили, что они уже видят приближающееся тысячеядерный процессор, а это уже совсем другой масштаб, где совсем другие проблемы и задачи. И все равно, процессоры останутся самым дефицитным ресурсом для использования на компьютере, даже в этом смысле. Вторым компьютерным дефицитным ресурсом была, есть и, видимо, навсегда останется основная память/main memory, то есть та память, в которой могут храниться данные и команды, непосредственно выполняемые на процессорах. Поэтому вторая по важности тематика - это управление памятью, причем здесь необходимо говорить про **управление физической основной памятью и про управление виртуальной памятью**. Виртуальная память - это традиционное средство, которое облегчает программирование приложений, создавая довольно реалистическое ощущение для программистов и для выполняемых программ о том, что на самом деле физическая основная память почти безгранична. Виртуальная память потому и виртуальна, что она управляется в действительности программами операционной системы, которая старается

создать такую иллюзию, которая чрезвычайно близка к реальности. Так что управление виртуальной памятью - это очень важная тематика данного курса.

- Напоследок мы поговорим про **управление данными внутри операционных систем**, про понятие файла и реализацию файловых систем.

Обратите внимание, что совсем ничего не будет рассмотрено про сети, потому что это отдельная специфика, те части операционных систем, которые отвечают за сетевые взаимодействия - крайне специальные. Почти ничего не будет про управление внешними устройствами, потому что в действительности там все тривиально, управление устройствами внутри операционных систем может быть простым или сложным в зависимости от простоты или сложности устройства, которым необходимо управлять. То, что нужно от операционной системы для управления устройствами - крайне незначительно, то есть там необходимо отдельное программирование внутри операционной системы для конкретных устройств.

Последнее, что важно отметить в этом предводном изложении курса - вас не должна удивлять толщина книг про операционные системы. **Эндрю Стюарт Таненбаум** - очень интересный и уважаемый разработчик компьютерных систем, который очень много сделал в области операционных систем, просвещения и пр., но увеличение объема его трудов от года к году, при том, что ничего особенного в области операционных систем не происходит, говорит о том, что на западе оплата труда ученого зависит от размера книг. Я советую читать Э. Таненбаума, но моя любимая книга в области операционных систем давно не переиздавалась, она переведена на русский язык – это **"Операционные системы", Цикритзис Д., Бернстайн Ф.** Бейгстайн является одним из крупнейших специалистов в области транзакций, последние не менее 30-ти лет он работает в компании Майкрософт. В книге "Операционные системы" нет темы, которая бы не была хорошо освещена и раскрыта. Очень популярен двухтомник **"Операционные системы. Основы и принципы", Дейтел Х., Дейтел П.Дж., Чофнес Д.Р.** (Harvey M. Deitel, Paul J. Deitel, David R. Choffnes). Сегодня мы познакомимся с тем, что такое операционные системы, далее будет исторический очерк истории развития операционных систем, её этапы, потом мы поговорим про функции операционных систем, а напоследок рассмотрим подходы к построению операционных систем.

Понятие "Операционная система"

Структура вычислительной системы: если говорить про вычислительные системы, компьютерные системы вообще, то они делятся на 2 класса компонентов:

- **hardware** - аппаратное обеспечение (процессор, основная память, устройство взаимодействия с пользователем, терминал, дисковые устройства и т.д., объединенные шиной, позволяющей процессору выполнять обмены с этими устройствами);

- **software** - программное обеспечение, которое в свою очередь делится на прикладное и системное, но это разбиение чрезвычайно условно.

Единственное, с чем все согласны - это то, что **операционные системы относятся к категории системного программного обеспечения**. С точки зрения, например, разработчика операционных систем система управления базами данных - это приложение, потому что это некая программа, которая пользуется услугами операционных систем, с точки зрения разработчика СУБД - это, конечно, системное программное обеспечение, потому что СУБД в действительности очень во многом повторяют то, что делают операционные системы в более узком контексте применительно именно к специфике работы с данными. Для разработчиков СУБД компиляторы безусловно являются приложениями. Все компиляторы являются приложениями, все системы программирования вне зависимости - используют ли они технологию баз данных или нет. С точки зрения разработчика-компилятора – это, безусловно, системное программное обеспечение, потому что они работают и действительно позволяют отобразить языки высокого уровня системы команд процессора. Какое же это прикладное программное обеспечение, когда разработчики компиляторов должны знать систему команд процессора не хуже, чем разработчики процессора. В России трудно делать с нуля компилятор, потому что негде взять документацию такого уровня, который требуется для разработчиков. У нас настолько популярно GCC, потому что документация, которая нужна для построения команд на выходе компиляторов, доступна в Америке.

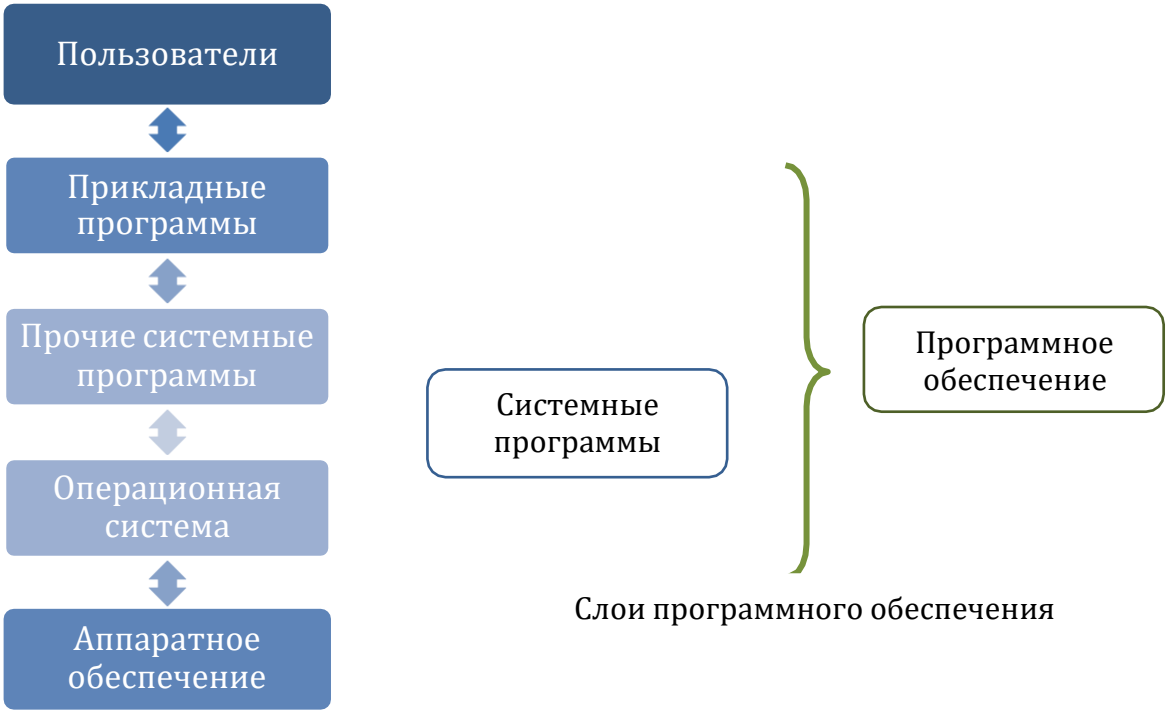

Рис. 1.1. Слои программного обеспечения компьютерной системы

Кто такие **конечные пользователи** - вопрос условный: либо это реальный человек, который взаимодействует с приложениями с помощью каких-либо интерактивных устройств, либо это приложение ещё более высокого уровня, которое опирается на сервисы других приложений. В любом случае - это тот уровень, который непосредственно обслуживает потребности пользователя. Этот уровень опирается на уровень стандартного прикладного программного обеспечения, который в свою очередь опирается на системные программы, которые все используют операционные системы. Операционные системы, соответственно, работают с программным обеспечением.

Точки зрения на операционные системы (ОС):

Операционная система как виртуальная машина. Когда-то В. П. Иванников написал: "Бытует мнение, что операционные системы представляют для пользователей некоторую виртуальную машину". В этом смысле имеется ввиду совсем не та виртуализация, которая сейчас использует понятие "в облаках" и т.д., имеется ввиду, что операционная система повышает уровень представления компьютера для разработчиков других программ. Она скрывает, например, особенности устройств внешней памяти, не заставляя разработчиков приложений впрямую работать иногда с неудобными интерфейсами этих устройств. От разработчиков других программ (и системных, и прикладных) скрываются детали обработки прерываний, распределение памятью, управление обычной памятью, основной и виртуальной. При управлении виртуальной памятью создается впечатление - иллюзия неограниченного размера оперативной памяти, при управлении процессами - иллюзия неограниченного числа процессоров. С такой виртуальной машиной, которую представляет для пользователя операционные системы, проще иметь дело разработчикам приложений. В действительности нормальному человеку, который использует ноутбук в основном для того, чтобы писать тексты и пр., в принципе нет никакого дела до того, как устроена операционная система, которая стоит в его ноутбуке. Однако, это не так, и Билл Гейтс, и Линус Торвальдс не дают нам забыть, что мы имеем дело с операционной системой, так или иначе нам об этом напоминая, например, когда мы видим сообщение от Microsoft, что виртуальная память закончилась, и это на компьютере с 64-разрядной адресацией.

Операционная система как менеджер ресурсов. Одна из основных задач операционной системы - это управление ресурсами. Например, если компьютер работает в многопользовательском режиме и у него всего одно устройство печати, которое доступно для пользователей, то необходимо создать ощущение, что в действительности принтеров столько, сколько необходимо. Для этого необходима некоторая виртуализация, простая - очереди печати, главное, что у людей при этом не возникает никаких трудностей. Особенно это важно для многопользовательских компьютеров, но далеко не только, потому что и на ноутбуке при работе Windows 10 поддерживаются сотни процессов, которые работают при том, что пользователь не

просит об этом, Windows сама для них распределяет ресурсы, иначе она не могла бы запустить столько процессов. Конечно, если система многопользовательская, то это распределение процессоров необходимо, необходимо делать упорядоченным и контролированным работу основной памяти и других ресурсов, там не должно быть влияния одного пользователя на другого пользователя через какие-то ресурсы.

Операционная система как защитник пользователей и программ. Это касается и ситуации с использованием одной вычислительной системы, когда на одной системе работает одновременно несколько пользователей и возникает проблема обеспечения безопасной совместной работы:

- никто из пользователей в оперативном режиме не должен иметь возможности удалять или повреждать чужие файлы;
- программы одних пользователей не должны произвольным образом вмешиваться в программы других пользователей;
- должны пресекаться попытки несанкционированного использования вычислительной системы, то есть должны проверяться полномочия пользователей.

В этом смысле задачей операционных систем является безопасность работы пользователей и их программ. С одной стороны, кажется, что в наш век персональных компьютеров, когда у каждого человека есть как минимум один, а как правило, больше 2-ух или 3-ех компьютеров - трудно утверждать, что требуется некая многопользовательская работа, но центры данных продолжают развиваться, на серверах, конечно же, работает несколько пользователей. Когда ресурсов много, нецелесообразно и глупо тратить их на одного человека. Это одна из современных проблем: люди не склонны выключать свои компьютеры, которые расходуют электроэнергию, бездумно растрачивая ресурсы. Если посчитать, сколько в мире в совокупности выходит на питание компьютеров, то становится страшно. Затея с биткоинами привела к тому, что ставятся специальные электростанции, чтобы их майнить.

Операционная система как постоянно функционирующее ядро. В действительности не бывает такого компьютера, на котором, когда никто ничего не делает, ничего бы не работало. В любом случае, то - что работает на компьютере, когда ему нечего делать - это так или иначе операционная система, то есть это какой-то процесс, который её представляет. Во многих операционных системах на компьютере постоянно присутствует только часть - **ядро/kernel** (в системах управления базами данных ядро - это engine). Проще показать не то - что из себя представляет операционная система и для чего она нужна, а то - что она делает. Чтобы это стало понятно, мы кратко рассмотри историю вычислительных систем, начиная с самого начала.

История эволюции вычислительных систем

Первый период: 1945 - 1955 года. На протяжении многих лет аппаратное программное обеспечение - hardware и software развивались совместно и оказывали взаимное влияние друг на друга. Прекрасным примером была организация работы в Институте точной механики и вычислительной техники, где работали потрясающие инженеры-электронщики, системотехники, архитекторы и прекрасные программисты. Это было "золотое племя", специалисты все время работали в связке, то есть инженеры проектировали и создавали новые компьютеры, а программисты участвовали в разработке оборудования и одновременно, до того времени как оборудование было готово, готовили для него системное программное обеспечение. Такая система существовала во многих компаниях, которые разрабатывали компьютеры, например, в Hewlett-Packard и Digital Equipment Corporation, IBM. В результате новые технические возможности приводили к созданию новых, удобных, эффективных и безопасных программ, а новые идеи в области программного обеспечения стимулировали инженеров в поиске новых технических решений. Интересно, что программистов инженеры-электронщики называли математиками. Очень ценным в то время был высокий уровень конкуренции. Удобство, эффективность и безопасность руководили этой эволюцией, естественным отбором, выживало то, что в действительности было лучше.

История вычислительных машин современного поколения, которые называются ЭВМ - электронные вычислительные машины, началась в конце Второй мировой войны этот период довольно трудно точно датировать, но условно это время с 1945 - 1955 годы.

Основная характеристика - это были ламповые компьютеры, которые делались на электронных вакуумных лампах, операционные системы отсутствовали (еще 10-15 лет назад в Московском физико-техническом институте студенты сами паяли лампы и откачивать из них воздух).

Принято считать, что первые ламповые вычислительные устройства были созданы в середине 40-х годов, они возникли в результате идеи **Джона фон Неймана**, их так и называют - Фон-Неймановскими компьютерами. Идея заключалась в том, что машина должна уметь выполнять команды, которые хранятся в основной памяти компьютера. Принцип фон Неймана означает, что команды выполняются одна за другой до тех пор, пока какая-то из команд явно не скажет, что следующей команде надо выбрать какую-то другую команду. Обращение к основной памяти производится по явным адресам, то есть - есть устройство управления памятью, которое получает от устройства выполнения команд адрес данных, хранящихся в основной памяти, интерпретирует этот адрес, как ему полается, и вытаскивает необходимый элемент данных из памяти. Совершенно поразительно, что современные компьютеры работают не так, тем не менее их продолжают считать Фон-Неймановскими. В них команды не выполняются одна за другой и совершенно точно, когда команды вырабатывают адрес,

не происходит обращение сразу к устройству управления памятью, а работает кэш, причем совсем не так, как полагается по фон Нейму.

В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации, и в программировании вычислительных машин. Это была скорее научно-исследовательская работа. С компьютерами работали исключительные энтузиасты, которые делали многие вещи, которые существенно опережали свое время. Мониторы, которые мы сегодня используем везде, появились ещё в 50-е годы прошлого века, в эти же годы на компьютерах разрабатывали устройства распознавания и синтеза речи. Честно скажем, что компьютеры тогда использовались исключительно в военных целях. Основной движущей силой становления этой технологии были атомная бомба и средства доставки, так как в конце Второй мировой назрела научная мысль, которая привела к появлению атомного оружия, а необходимых средств расчетов, которые существовали до войны, не хватало. В Центральном аэрогидродинамическом институте имени профессора Н.Е. Жуковского довоенные самолеты Туполева рассчитывались бригадами расчетчиков - это 10-15 человек во главе с бригадиром, который, получая задачу, разбивал её на кусочки и раздавал каждому расчетчику. Расчетчики на арифмометре считали свою часть и отдавали результаты бригадиру - это типичный MPI (Message Passing Interface), только вместо кластера - бригада расчетчиков. Когда появились реактивные самолеты, таких бригад расчетчиков стало не хватать, так как задачи стали сложнее, а бригада при эффективной работе не может превышать 30-ти человек, иначе не справляется бригадир - то есть это кластер с большим числом узлов. Программировать параллельные приложения на кластер с большим числом узлов очень трудно, поэтому компьютеры были необходимы и для самолетостроения.

В это время отсутствовали операционные системы, программирование велось исключительно на машинном языке. Все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления компьютера. Программа и данные для обработки загружались в память машины в лучшем случае с колоды перфокарт - картонных карточек, на которых пробиты дырки, которые соответственно означают 1. Устройство ввода их преобразовывало в цифровую форму и записывало в память компьютера. Альтернативой были перфоленты - это ленточка с дырками, которая прокручивалась и читалась. Вычислительная система выполняла только одну операцию, никакого совмещения ни с чем не было. Отладка велась с пульта управления. В действительности пульт управления позволял делать то же самое, что современные отладчики, то есть остановку по адресу, остановку обращения к памяти по указанному адресу прямо на аппаратном уровне. Компьютер замирал, а с помощью пульта управления можно было посмотреть, что и где находится в памяти, на псевдо dump.

В это же время начало появляться первое системное программное обеспечение. В 1951-1952 годах возникают прообразы первых компиляторов с языков

символического кодирования языков, прежде всего Fortran, который является языком-долгожителем, ему практически 70 лет, сейчас он поддерживается и работает. В 1954 году **Нэт Рочестер** разрабатывает ассемблер для раннего компьютера IBM-701. В 50-е годы был разработан первый компилятор фортрана в России, в Вычислительном центре МГУ одним из основных его разработчиков был **Е. А. Жеголев**. Первый период характеризуется крайне высокой стоимостью вычислительных систем. Вакуумные электронные лампы взрывались, половина времени уходила на постоянную замену ламп. Машин было очень мало, они использовались малоэффективно (с последовательной обработкой).

Второй период: 1955 - начало 60-х годов. Это западная шкала, шкала 1945 - 1955 года совпадает для Запада и Востока, и в США, и в Великобритании, и в СССР начало этой работы совпадало. Во второй период, когда на Западе начали появляться транзисторы, в Советском Союзе их ещё не было, в то время они отстали уже на 5 лет.

Основная характеристика - появились компьютеры на основе транзисторов и пакетные операционные системы.

Силиконовые полупроводники способствовали повышению надежности компьютеров, так как твердый транзистор гораздо более надежный, чем лампа, а это снижение потребления энергопотребления, потому что транзисторы потребляют меньше памяти и не греются как лампы. Следовательно, происходит упрощение систем охлаждения. В середине 80-х годов делался компьютер Электроника СС БИС - это был суперкомпьютер, который был начат слишком поздно, его не успели сделать до конца Советского Союза. Он делался уже на больших интегральных схемах (БИС), тем не менее потреблял настолько много электроэнергии, что для построения системы охлаждения была создана специальная лаборатория, которая разрабатывала специальное программное обеспечение охлаждающих компьютеров, которые тоже грелись.

Происходит уменьшение размеров компьютеров, удешевление эксплуатации и обслуживания. Наконец, это время - начало широкого использования компьютеров коммерческими компаниями. Произошло бурное развитие алгоритмических языков: ALGOL-58, из существующих - LISP - это целое направление, целая парадигма программирования, функциональное программирование, которое современной молодежью воспринимается как открытие, а этот язык появился 60 лет назад, COBOL - язык IBM, который продолжает активно использоваться в бизнесе, ALGOL-60 - один из самых красивых языков, PL-1 - язык IBM, развитый Fortran и многие другие. Появились компиляторы и технологии компиляции, редакторы связи (чтобы было можно компилировать модули), библиотеки модулей, библиотеки математических и служебных подпрограмм. Все это привело к упрощению процесса программирования и к разделению персонала, который работает с компьютерами на: программистов, операторов, специалистов по эксплуатации, которые были очень важны, потому что

компьютеры были не очень надежными, и разработчиков новых вычислительных машин.

Изменился процесс прогона программ: пользователь приносит программу с входными данными в виде колоды перфокарт (задание) и указывает требуемые для нее ресурсы; оператор загружает задание в память машины и запускает его на исполнение; полученные выходные данные печатаются на принтере, и пользователь получает их обратно. Все в этом режиме хорошо, кроме того, что операторов много эксплуатируют, для того, чтобы облегчить их работу и повысить эффективность работы компьютеров, появился пакетный режим обработки, то есть задания с похожими требуемыми ресурсами собирались вместе, образуя **пакет заданий**. Соответственно, появились системы, которые поддерживали пакетную обработку - это были прообразы операционных систем. Они автоматизировали обработку одной программы из пакета за другой без задержек, это увеличивало коэффициент загрузки процессора. Для того, чтобы было можно в одном задании (в одной колоде перфокарт) описать несколько шагов своей работы, например, откомпилировать программу, если она откомпилируется без ошибок, то построить исполняемый код и выполнить его с указанными входными данными. Чтобы описать такую последовательность работы - появились специальные формализованные языки управления заданиями, то есть каждый программист мог сообщить системе и оператору, что он хочет сделать на компьютере. Все это поддерживалось системными программами, которые являлись прообразами современных операционных систем, то есть системными программами, предназначенными для управления вычислительным процессом. Этот режим продлился до начала 60-х годов, а у нас в стране до середины 70-х.

Третий период: начало 60-х - 1980 год.

Основная характеристика - появилась технология на основе интегральных микросхем, сначала среднего масштаба, потом больших и первые многозадачные или многопользовательские операционные системы.

Интегральные микросхемы ничего принципиально нового не внесли, так как это уменьшение транзисторов и возможность компоновки их большого количества в одной небольшой плате. Платы стали печатными, их можно стало производить сразу со всеми транзисторами, то есть производить готовую схему. Вычислительная техника становилась все более надежной и дешевой, росла сложность и количество задач, которые решали компьютеры. Повышалась производительность процессоров, а повышению эффективности использования процессорного времени естественным образом стала мешать низкая скорость механических устройств ввода-вывода. На первых компьютерах, когда они работали без операционных систем, они спокойно стояли, пока выполнялся обмен с внешним устройством, например, задавалось чтение какого-то блока с магнитной лентой, пока магнитная лента крутилась, пока она читалась - процессор стоял, пока устройство не скажет, что все выполнено. На первых

компьютерах это можно было как-то терпеть, но при коммерческом использовании это стало сильно мешать эффективному использованию машин.

Аналогичная ситуация происходила с буферизацией ввода/вывода, но прежде всего вывода. Как быть с печатью? Вначале стали делать буферизацию на компьютере, где программа должна была что-то напечатать, вначале реальные операции ввода-вывода осуществлялись в режиме off-line, буфер набирался и отдавался на другой компьютер, который последовательно печатал. Потом это стали делать на том же компьютере, который производит вычисления, просто разгрузка буфера стала происходить на фоне каких-то других процессов и задач. Это стало называться **Spooling** (Simultaneous Peripheral Operations On Line) - **подкачка-откачка данных**, онлайновое выполнение внешних операций, которое позволяло совместить реальные операции ввода-вывода одного задания с выполнением другого задания. Естественно, для того, чтобы это можно было делать, потребовался механизм прерывания, коль скоро мы разрешаем внешним устройствам работать асинхронно с процессором - нужно, чтобы они имели возможность каким-то образом сами сказать, когда им это будет необходимо. Для этого надо силком прервать выполнение программы на центральном процессоре и заставить какую-то служебную, системную программу обратить внимание на то, что произошло - это стали называть прерыванием. Про прерывание мы поговорим далее в связи с разными вещами: в связи с внешними устройствами и в связи с эволюцией механизма прерывания.

Первыми устройствами внешней памяти были магнитные ленты и магнитные барабаны. **Магнитная лента** - это длинная, пластиковая и довольно узкая намагниченная ленточка, которая наматывается на катушку, в устройстве она перематывается с одной катушки на другую. Лента проходит между магнитными головками, которые могут читать и писать её, когда будет выполнено аппаратное позиционирование на ленте. Хорошо в магнитных лентах то, что они могут быть очень емкими, сколько данных поместится на одну ленту, зависит от того, сколько ленты поместится на катушку, а это зависит от того, какой толщины лента. Самые первые магнитные ленты, когда технология производства пластмасс была ещё неразвитой, были толстыми. Плохо - то, что в перематывающиеся катушки могли эффективно работать только в последовательном режиме. Если мы хотим найти данные, которые находятся на другом конце магнитной ленты, то необходимо 10 мин. ждать перемотку, так как быстро ленту мотать нельзя, она рвется. Второй вид устройств - это **магнитные барабаны**, массивные железные цилиндры, у которых намагничены только внешние поверхности, из-за массивновности их можно было очень быстро крутить, что позволяло выполнять быстрый обмен. На БЭСМ-6 были очень небольшие магнитные барабаны - 96 килобайт, при этом они весили полтонны, зато время произвольного доступа к блоку данных было всего в 10 раз меньше, чем время доступа к основной памяти. Магнитные ленты использовались для пакета заданий, при обработке пакета заданий на магнитной ленте очередность запуска заданий определялась порядком их

ввода (последовательно подряд записываем - подряд читаем), магнитные барабаны использовались в основном для того, чтобы поддерживать виртуальную память.

В 60-е годы появились **магнитные диски** как компромисс между магнитными лентами и магнитными барабанами. Это были диски с подвижными головками, то есть они менее емкие, чем магнитные ленты, менее быстрые, чем магнитные барабаны, но зато они более емкие, чем магнитные барабаны и появлялся последовательный доступ, которого нет на лентах. Поэтому, когда появились магнитные диски - появилась возможность при обработке пакета заданий операционной системе или системе, которая планировала выполнение задания, выбирать очередное задание для выполнения, то есть **пакетные системы начали заниматься планированием заданий.** То или иное задание выбиралось в зависимости от наличия запрошенных ресурсов, приоритетности и срочности вычислений.

Это привело к появлению мультипрограммирования - дальнейшего повышения эффективности использования процессора. **Мультипрограммирование означает**, что в основной памяти компьютера одновременно присутствует несколько программ, которые могут выполняться на процессоре до тех пор, пока какая-то программа выполняется подряд. Программа выполняется, если она хочет выполнить операцию, например, ввода данных с какого-то внешнего устройства, то процессор берет на выполнение другую программу из числа тех, которые сейчас готовы и наличествуют в памяти, и выполняет её на фоне работы внешнего устройства. Этот режим назывался совмещением обмена со счетом. Когда запущенная операция ввода-вывода заканчивается, процессор возвращается к выполнению первой программы и т.д. Для этого требуется наличие в основной памяти нескольких программ одновременно. В этом случае каждая программа загружается в свой раздел оперативной памяти и не должна влиять на выполнение другой программы.

Появление мультипрограммирования явилось революцией в области архитектуры вычислительной системы. Для обеспечения мультипрограммного режима потребовалась соответствующая аппаратная поддержка, защита, чтобы программы от разных пользователей никаким образом не могли навредить друг другу. Программы пользователей, поскольку у каждого ограниченный набор ресурсов, который контролируется операционной системой, никак не должны были иметь возможность самостоятельного доступа к распределению ресурсов, только через операционную систему. Для этого потребовалось ввести привилегированные и непривилегированные команды, то есть команды, которые можно выполнять обычным пользователям, и программы, которые можно выполнять только привилегированным программам, которые уже тогда стали называться операционными системами. Переход обращения от прикладной программы к операционной системе с помощью аппаратуры сопровождался контролируемой аппаратурой сменой режима с обычного на привилегированный, и обратно. Появились средства защиты памяти, чтобы было можно изолировать одновременно выполняемые и конкурирующие пользовательские

программы одну от другой, а операционную систему от программ пользователей. В данном случае русское слово "конкурирующие" - это калька с английского слова "concurrent", concurrent user programs означает, что программы выполняются либо с чередованием команд - как бы квазипараллельно, если есть всего один процессор, либо реально параллельно - если процессоров больше, чем один. Слово **"concurrent"** очень емкое, слово "конкурирующее" его не заменяет, так как отражает только один аспект, если далее я буду в какой-то момент говорить "одновременно выполняемые", необходимо иметь ввиду, что это означает либо параллельно выполняемые, либо с произвольным чередованием команд.

Для всего этого требовалась **поддержка механизма прерываний**, которые делились на внешние и внутренние.

- **Внешние прерывания** - это прерывания работы процессора по запросу какого-то внешнего устройства, которое таким образом сообщает, что в нем произошло нечто, например, о том, что завершилась операция ввода-вывода. То есть это асинхронное событие, которое возникает независимо от желания центрального процессора, который спокойно выполняет какие-то программы.

- **Внутреннее прерывание**, которое по-другому называют исключительными ситуациями, а иногда exception - это ситуация, когда при выполнении программы возникает нечто, что требует вмешательства операционной системы. Например, это может быть деление на 0 или программа пытается нарушить защиту, пытается каким-то образом обратиться к чужой памяти, в этом случае процессор должен сообщить операционной системе, что что-то происходит не так, он должен перестать выполнять эту программу и начать выполнять нечто, что должно разобраться с ситуацией. Под эту же категорию попадают совсем не исключительные ситуации, когда процесс работает на своей виртуальной памяти и хочет обратиться к странице, которой в данный момент нет в основной памяти. Это нормальная ситуация - это неявный запрос страницы виртуальной памяти, который обрабатывается за счет того же самого механизма внутренних прерываний.

Все эти нововведения привели к **наиболее существенным изменениям в архитектуре, строении, функциональности и пр. операционных систем**. В это время, в 70-е годы произошел взрыв публикаций на тему операционных систем, появилось множество интересных работ, методов и алгоритмов. Это был очень плодотворный период для развития операционных систем, в который Россия на 10 лет отставала от Запада в области электроники - у них были интегральные схемы, а у нас транзисторные компьютеры, время которых закончилось в начале 60-х годов. Но что касается операционных систем - мы шли наравне. По большому счету, мы стали отставать в области программных систем от Запада примерно тогда же, когда стали делать свои компьютеры, это было уже следствием того, что программистам стало не для чего делать новые программы.

Появились соответствующие интерфейсы между приложениями/прикладной программой и операционной системой, которые назывались по-разному в разных операционных системах, но фактически это были наборы **системных вызовов**, в том смысле, как это принято в мире Unix. Я буду в дальнейшем называть Linux и все Unix - Unix. Организация очереди из заданий в основной памяти и выделение процессора какому-то одному из заданий требуют **планирования** заданий. Для переключения процессора с одного задания на другое возникает потребность в сохранении содержимого регистров и структур данных, необходимых для продолжения выполнения задания, иначе говоря, необходимо уметь сохранять и восстанавливать **контекст** задания.

Поскольку основная память является ограниченным ресурсом, требуются **стратегии управления памятью,** упорядочивающие процессы размещения, замещения и выборки информации из памяти**.** На БЭСМ-6 изначально было 32 килослова основной памяти, каждое слово по 6 байт, то есть 132 килобайта основной памяти - это было очень мало, программисты, особенно программисты, которые занимались решением численных задач, а это одна из самых важных областей программирования и использования компьютеров, просили хотя бы в два раза больше памяти. Когда заканчивалась эксплуатация БЭСМ-6, то они обладали уже 256-ю килословами, то есть больше, чем мегабайтом памяти. Когда в России появились PDP-11/70 (старшая модель PDP) компании DEC, то в них было 2 мегабайта основной памяти. Эти компьютеры использовали ядерщики для обработки экспериментов и пр., им этого количества памяти не хватало. Когда появились первые гигабайты основной памяти, стало не хватать и их, сейчас можно поставить 32 гигабайта, но и этого недостаточно. Численным программистом необходимо больше памяти, если они получат больше памяти, то начнут дробить мельче сетку и начнут получать более точные результаты, возможно, получать их быстрее.

Чтобы обеспечить обмен данными между программами необходимо **средства коммуникации**, общение. Для корректного санкционированного обмена данными требуется предусмотреть координацию программами своих действий, то есть **средства синхронизации.** Мультипрограммные пакетные системы обеспечивают окружение, в которых разные системные ресурсы используются эффективно, то есть, видимо, пакетный режим - это самый эффективный режим для обеспечения оптимального использования компьютерных ресурсов. Система подбирает набор одновременно выполняемых заданий так, чтобы все ресурсы использовались, и чтобы их хватало. При этом отсутствует возможность реального взаимодействия пользователя с выполняемой программой, то есть необходимо предусмотреть с помощью этих управляющих карт все возможные ситуации. Советский клон мэйнфреймов IBM - ЕС ЭВМ был проектом-ошибкой, потому что позаимствовать у IBM документацию, которая позволяла бы воспроизвести на отечественных ресурсах настолько точно мэйнфреймы IBM, чтобы на них можно было выполнять программы IBM - было плохой идеей. Эти компьютеры не работали, это произошло по двум причинам: во-первых, потому что они

изготавливались на советской элементной базе, во-вторых, у них были категорически плохие внешние устройства, особенно дисковые устройства, которые делали в Болгарии. Когда появились две старших модели ЕС-1060 и ЕС-1065, то они уже были гораздо более надежными и оригинальными. Программистам было тяжело работать на компьютерах подобного качества и с Job Control Language - языком, который описывает управление заданиями. Это был громоздкий, некрасивый язык, на котором надо было дополнительно программировать, то есть писалась программа, потом было необходимо написать управляющую программу на Job Control Language, чтобы отдать программу на счет. Было трудно отлаживать программы, отладка программ занимала много времени, требовала изучения распечаток содержимого памяти и регистров или использования отладочной печати.

Логическим развитием идей мультипрограммирования стали **системы разделения времени**. Первая книга про системы разделения времени, с которой мне довелось познакомиться, была издана в 1964 году. Идея заключалась в том, чтобы в многопользовательском режиме, когда с компьютером работает одновременно несколько пользователей, обеспечить для каждого пользователя видимость, что как будто он работает с компьютером в интерактивном режиме в одиночку. В этом случае процессор переключается между задачами, которые обслуживают разных пользователей не только в время операций ввода-вывода, но и просто по прошествии определенного интервала времени - кванта времени. Эти переключения происходят настолько часто, что пользователи могут успевать взаимодействовать со своими программами во время их выполнения в прямом режиме, то есть интерактивно. Это обеспечивает возможность одновременной работы многих пользователей на одной компьютерной системе.

Во всех многопользовательских системах для каждого пользователя, который сейчас обслуживается компьютером, в памяти должна присутствовать хотя бы одна программа или часть программы. Идея, что можно не полностью держать в основной памяти программу, но, чтобы её тем не менее можно было выполнять, явилась очень плодотворной и позволила снизить число ограничений на число пользователей. Реализуется эта идея на основе механизма **виртуальной памяти**, что означает, что у каждого пользовательского процесса имеется свое отдельное виртуальное адресное пространство, которое может быть объемом больше, чем имеющаяся физическая основная память, может необязательно отражаться на основной памяти. В результате для каждого пользователя создается иллюзия неограниченной основной памяти, которая в действительности является виртуальной. Работа в режиме разделения времени позволяет вести отладку в интерактивном режиме, то есть в это время появились интерактивные отладчики, которые поначалу просто воспроизводили то, что можно было делать в ранних компьютерах с пульта и записывать информацию на диск непосредственно с клавиатуры, просто командами клавиатуры. В среде многопользовательских компьютеров это потребовало разработки развитых **файловых систем** - первых систем управления данными. Их основное назначение заключалось,

прежде всего, в избавлении людей от потребности самостоятельно распределять внешнюю память, в именовании кусочков внешней памяти. На БЭСМ-6 магнитные ленты оставались лентами, которые перематывались с одной кассеты на другую, но при этом они были блочными, то есть по интерфейсу это были диски, можно было прочитать определенный блок с указанной ленты, но работа с лентами происходила напрямую. В то время каждый молодой программист участвовал в нескольких программных проектах, в каждом из которых было несколько фаз: фаза текста - написания и совершенствования текста, фаза версии - предыдущие версии, фаза откомпилированных программам - программы, которые можно выполнять, и набор тестовых данных. Для каждого проекта приходилось иметь отдельную магнитную ленту, чтобы понять - где и что записано на магнитной ленте - приходилось иметь набор перфокарт. Каждая карточка содержала разбивку магнитной ленты по кусочкам. Потеря колоды перфокарт была невосполнимой потерей. Поэтому именование и автоматическое распределение памяти - это очень важный момент в файловых системах.

Внешняя эволюция вычислительных систем. В это время в компании IBM возникла идея создания семейств программно-совместимых компьютеров. Они были совместимы на уровне системы команд, что позволяло выполнять на них одну и ту же операционную систему. Первым таким семейством компьютеров, которые были построены на интегральных микросхемах, явилась серия машин IBM/360 -это был массовый мэйнфрейм, который превосходил машины второго поколения по критерию цена/производительность. Совместимость сверху - вниз, старших моделей с маленькими моделями, с одной стороны, позволяла существенно экономить силы при разработке программного обеспечения при выпуске новых моделей одной и той же серии, с другой - очень сильно ограничивала возможности развития архитектуры. Одним из косвенных результатов такой политики явилось то, что IBM разработало первый рисковый процессор - reduced instruction set processor, но его не стали реализовывать, а продали. Процессоры SPARC компании Sun Microsystems - это в действительности процессоры IBM, которая таким образом упустила сектор рынка, позволив выйти вперед ряду компаний на рынке клиент-сервера. Когда IBM к этому вернулась и стала делать AIX и свою серию Unix серверов - рынок уже был занят. Аналогичная история произошла с Digital Equipment Corporation (DEC), которая закончила свое существование более 20-ти лет назад, это была очень хорошая компания, которая для российского научного сообщества была очень полезной. У компании DEC была своя серия 16 разрядных PDP-11, которые были миникомпьютерами, среди них был настольный компьютер LSI-1, на нем Oracle показывал свои первые реализации СУБД. Ситуация стала плачевной, когда появился 32 разрядный VAX. Все компьютеры были семейством, совместимым по системе команд, то есть 32 разрядные VAX могли выполнять 16 разрядные приложения, которые были написаны для PDP. Потом компания сделала рисковый процессор DEC Alpha, возможно, лучший продукт из reduced instruction set computing. У DEC все

компьютеры были микропрограммными, это означает, что каждая команда - это микропрограмма, которая запаяна в ПЗУ. Reduced instruction set processor - это облагороженная внутренняя система команд в микропрограммах компьютеров, но с разнообразными ухищрениями. DEC решила добиться, чтобы на процессоре Alpha работала операционная система микропрограммного VAX-11, чтобы туда можно было целиком поставить VMS операционную систему, и сделали двоичный компилятор из очень сложной системы команд VAX-11 в систему команд Alpha. Фактически это было вариацией на тему микропрограммирования, только совсем в другом стиле. Из этого ничего не вышло и компания DEC закончила свое существование.

Кроме того, что поддержка единой системы команд для разных моделей одного ряда, конечно, сильно влияла на ограничение архитектурных решений, с другой стороны, это приводило к тому, что операционная система становилась чрезмерно сложной и громадной по объему. Она должна была учитывать все мелкие различия между всеми моделями ряда (от миникомпьютеров до гигантских машин). Сила одного семейства была одновременно и его слабостью. Обилие разнообразной периферии, различное окружение порождали сложную и огромную операционную систему, а чем больше программная система - тем больше у нее системных ошибок. В конце существования OS/360 только известных и неисправленных ошибок в ней содержалось более 1000. Сколько осталось неизвестных ошибок - неизвестно, сколько ошибок в действующих системах IBM, LINUX и Microsoft – неизвестно. Тем не менее, сама эта мысль привела к здравой идее стандартизации операционных систем, которая в дальнейшем получила активное развитие.

Четвертый период: 1980-настоящее время.

Основная характеристика - появились персональные компьютеры и классические, сетевые и распределенные системы

В 80-е годы появились большие интегральные схемы (БИС), то есть те самые чипы, которые работают сейчас, они все больше и больше интегрированы, но идея та же. Это привело к возрастанию степени интеграции и удешевление микросхем (чем более массовое производство, тем лучше). Возник тандем Intel и Microsoft, начался подъем Intel. В середине 80-х годов началась эра персональных компьютеров, сначала это был однопрограммный режим. Первые персональные компьютеры обладали очень убогой архитектурой, происходила деградация архитектуры и операционных систем. Первыми операционными системами были системы семейства DOS - Disk Operating System - дисковая операционная система. Первая оконная система, которую производил Microsoft - Windows NT 3.1 была сделана как надстройка над DOS. В России в начале 90-х годов появились первые персональные компьютеры, которые были очень дорогими и попадали в академические институты по одной единице, простые люди не могли позволить себе их купить. Программы, в частности Word 6.0 были крайне неудобными. Массовость персональных компьютеров привела к двум вещам, во-

первых, потребность в "дружественном" программном обеспечении, которая естественно привела к тому, что первыми программами, которые появились на персональных компьютерах, были игры. Конец кастовости программистов. Коль скоро компьютеры стали персональными и доступными - появилось желание программировать у неспециалистов, в результате появилось большое количество некачественных программ.

Электроника СС БИС делалась по образу и подобию Cray-1, который был векторным конвейерным суперкомпьютером, он обладал очень хорошей пропускной способностью, если векторизовались операции. Он хорошо работал на программах, которые работали с регулярными массивами, когда каждый массив можно было представить, как вектор и отправить его на обработку. Но у Cray-1 не было никаких средств ввода-вывода, имелся только один сетевой адаптер, который позволял подсоединять к нему внешние машины. Было известно, что на российском компьютере будет среда Unix и фронтенды будут с ним работать, но было неизвестно, какой будет компьютер, когда дело дойдет до реализации, потому что в стране front-end компьютеров не делалось. Поэтому программное обеспечение для внешней машины писалось и отлаживалось на разных компьютерах, пока компания Borland не сделала прекрасный компилятор С, который поддерживал стандарт языка С, который уже тогда был. С его помощью отлаженная программа переехала на компьютер, который использовался в реальной системе Электроника СС БИС. Из этого можно сделать вывод, что стандартизация крайне полезна, а в то время появился ещё и стандарт POSIX, который начинал стандартизовать набор системных вызовов Unix. Со временем рост сложности и разнообразия задач, решаемых на персональных компьютерах, необходимость повышения надежности их работы привели к возрождению практически всех черт, характерных для архитектуры больших вычислительных систем. Современные персональные компьютеры, конечно же, не персональные, они просто используются в персональном режиме, все это повторяется на компьютерах, которые носят серверный характер, благодаря масштабированию аппаратных средств.

С середины 80-х активно стала развиваться технология локальных сетей ЭВМ, в том числе с включением персональных компьютеров. Появились термины Network and Distributed Operating Systems. **Билл Джой** был одним из основателей и вице президентом компании Sun Microsystems, Джой входил в команду, которая в Университете Беркли делала BSD Unix. Когда Sun Microsystems была молодой и растущей компанией, Бил Джой сказал крылатую фразу: "The Network is the Computer". На самом деле в мире Unix никогда не было разницы между отдельным компьютером и компьютером, который подключен к сети, в которой находятся другие компьютеры. Для Unix с использованием разных технологий, в особенности технологией сокетов на основе TCP/IP, в действительности доступ к другим компьютерам был настолько же естественным, как доступ к своим. Когда Microsoft ввела термин "Network and

Operating Systems", это для специалистов по Unix не было открытием, Unix был настолько же локальной системой, насколько и сетевой.

- **В сетевой операционной системе** пользователи знают о наличии другого сетевого компьютера и могут воспользоваться его ресурсами, в локальной операционной системе имеется программная поддержка сетевых интерфейсных устройств доступа к удаленным ресурсам. Эти дополнения существенно не меняют структуру операционной системы. Сетевая операционная система содержит только средства взаимодействия в сети, больше ничего не нужно, во всех остальных отношениях она ничем от локальной не отличается.

- **Распределенная сетевая операционная система** - одна из вещей, о которых мечтал **Эндрю Таненбаум**, который в свое время сделал проект "Медуза" в Амстердаме. Это была распределенная операционная система, которая отличалась от сетевой тем, что в ней пользователю не надо было знать, что это не компьютер, система сама решает, где будет выполняться его задача в сети, где будут храниться его данные и пр. Пользователь может не знать, подключен ли компьютер к сети. Внутреннее строение распределенных операционных систем принципиально отличается от строения автономных, но их никогда и не было таких, которые были бы доступны пользователям. Работающих распределенных файловый систем никогда не было, были только эксперименты. Автономные операционные системы мы будем называть **классическими.**

Все периоды развития операционных систем до начала нового тысячелетия сопровождались тем, что их в мире было много: системы Unix, практически каждая компания, которая производила компьютеры, имела свой вариант Unix, они были более-менее совместимыми по функциональности, но они были разные (у IBM был AIX, у Hewlett-Packard - HP VX, у Sun Microsystems - Solaris и т.д.); системы, которые не относились к категории Unix, не только Microsoft, многие компании делали свои операционные системы. Когда на рынке появилось множество операционных систем, то возникли опасения, что Intel и Microsoft задавят все остальные компании, потому что рынок персональных компьютеров казался слишком большим. В своем выступлении в Бухаресте в 1994 году **Линус Торвальдс** нахально и вызывающе заявил, что они ещё не знают, но очень скоро узнают, что Linux идет, пройдет ещё лет 5, и на компьютерах больше нельзя будет увидеть операционных систем Microsoft - Linux вытеснит их из персональных компьютеров. Прошло 25 лет, но этого не произошло, зато Linux вытеснил Unix. Сначала все компании поддерживали и свою систему, и Linux, потом, поразмыслив, решили больше не поддерживать свои системы, если Linux всех устраивает, в результате все остальные операционные системы категории Unix пропали, кроме BSD. Здесь можно казать, что в этом преимущество open source, BSD с лицензией, которая позволяет использовать программы с открытыми кодами как угодно, в коммерческих целях тоже, стала в этом отношении не убиваемой.

В результате мы имеем из универсальных операционных систем: Linux, Unix BSD, QNX - универсальная Unix подобная система реального времени (но не жесткого) и Microsoft. Появилась новость, что Microsoft переводит Internet Explorer на движок от Gnome, осталось только дождаться, когда внутрь windows поместят ядро Linux. Если это произойдет, то останется один Linux. Хорошо это или плохо - неизвестно, потому что даже на open source должна быть конкуренция, то есть одна операционная система - это плохо. В Москве (в основном в МГУ) была очень сильная группа Macintosh, которая работала с macOS, которая была до Unix подобного UNIX-based macOS - это была нетривиальная, очень интересная операционная система Macintosh и Apple. У Unix есть много недостатков, когда мы будем рассматривать возможности операционных систем, то специально на них остановимся, но у него есть одна черта, которая все отрицательное скрашивает - это прозрачная операционная система, где все видно и понятно, что происходит внутри. А современная система Apple этим не обладает, они смогли из Unix сделать абсолютно закрытую операционную систему, в которой ничего не видно. MacOS настолько же похабна как и Android, потому что и то, и другое - испорченный Linux. Человек, который приобретает продукцию на Android, будет гораздо более свободен и сможет сделать с ней все, что захочет, с продукцией Apple - человек будет зависеть от того, что захочет Apple. При этом в Apple активно используются объекты в C, альтернативы C++, одним из основных разработчиков которых является **Джеффри Кнаут**, который со своей стороны является поклонником **Ричарда Столлмана**.

Основные функции классических операционных систем

Перечень основных функций операционных систем близок к тому, что в свое время сформулировал известный голландский программист и computer science **Эдсгер Вибе Дейкстра**, разрабатывая свою операционную систему THE, у него есть замечательная статья "THE Operating System" в которой это описывается. Приведем немного расширенный вариант:

- планирование заданий и использования процессора;
- обеспечение программ средствами коммуникации и синхронизации;
- управление памятью;
- управление файловой системой;
- управление вводом-выводом;
- обеспечение безопасности.

О безопасности: при том, что в нашей стране все говорят про information security, computer security, неприкосновенность индивидуальных данных и т.д. - мы почти не будем рассматривать security в операционных системах, потому что это настолько отдельная тема. Похоже, среди множества подходов к безопасности все-таки единственным условно-надежным подходом является криптография. Условным, потому что, не смотря на Закон Мура и пр., производительность вычислительных

систем растет настолько быстро, что стойкость криптографической защиты остается надежной очень недолго, проходит несколько лет и необходимо увеличивать, удлинять ключи - это бесконечная борьба. В Unix, как и в Windows - парольная защита/password security, пароли хранятся в зашифрованном виде. Интересна закономерность, что злоумышленники могут взломать парольную защиту на любом компьютере приблизительно за день с помощью простого перебора. Когда в Unix увеличили длину ключа для шифрования, то злоумышленники стали догонять через год, то есть через год они получают способность взломать за день. Существующие механизмы обеспечения безопасности в действительности защищают весьма условно. Мы будем рассматривать безопасность только в контексте файловых систем, в контексте очень простых средств защиты.

Рассмотрим **общие принципы и алгоритмы реализации функций операционных систем**. Какие существуют основные понятия и концепции операционных систем? В любой операционной системе поддерживается некоторый механизм, который обеспечивает интерфейс между приложениями и услугами ядра, который позволяет пользовательским программам обращаться за услугами ядра. Сегодня его принято называть **"механизм системных вызовов"**, он существовал с тех пор, как появилась защита операционной системы от пользователей, как появились два режима выполнения команд - привилегированный и непривилегированный. Механизмом системных вызовов называлось: в БЭСМ-6 - экстракоды, в IBM - системные макрокоманды, в Unix - системные вызовы. В любом варианте для того, чтобы было можно добраться из пользовательской программы до некоторой службы, которая находится в ядре - необходимо, чтобы процессор изменил режим работы, перейдя в режим, когда можно выполнять привилегированные команды, потому что именно они требуются, чтобы выполнить такую программу, которую нельзя выполнить - просто загрузив её из библиотеки. Для того, чтобы это сделать - необходимо каким-то образом прервать обычное выполнение программы, чтобы через это прерывание начало работать ядро операционной системы. Оказывается, и все это давно поняли, самый простой механизм - это вызвать искусственное внешнее прерывание путем попытки выполнения команд с запрещенным кодом операции. Команды процессора устроены следующим образом: у каждой команды есть некоторое поле постоянной или переменной длинны, которое говорит, что эта команда делает, она называется "код операции". После нее следуют какие-то дополнительные поля, которые содержат информацию о данных, которые будет обрабатывать эта команда. В любой системе команд среди кодов операции есть неиспользуемые, один из них специальным образом выбирается именно для того, чтобы вызывать внутренние прерывания, которые понимаются операционной системой как обращение к её интерфейсам.

- **в БЭСМ-6** - это называлось **"экстракод"**, что означало дополнительный код операции, там был один код операции, которого не было в системе команд, именно он интерпретировался операционной системой как обращение приложения к ней за какой-то услугой.

- **в IBM** - это называлось **"системной макрокомандой"**, потому что основным системным языком программирования был язык макроассемблера, то есть там был очень мощный макрогенератор. Он был настолько мощным, что при взгляде со стороны этот язык ассемблера иногда казался языком высокого уровня. IBM закрыли вызовы, запрещенные команды - макросами, которые как бы делали их как будто обращениями к подпрограмме, как будто мы вызываем подпрограмму на уровне самого приложения, и назвали это системной макрокомандой.

- **в Unix** - это называлось **"системный вызов"**, потому что в действительности там используется один код операции на каждой запрещенной архитектуре, но есть специальная библиотека, которая называется библиотекой системных вызовов, которая содержит ровно столько кода, сколько разных слов поддерживает ядро операционной системы. Каждый модуль этой библиотеки с точки зрения приложения - это отдельная функция, которая выполняет необходимый сервис, а на самом деле - это функция, которая только готовит параметры для того, чтобы потом выдать запрещенную команду. Unix - это C, то есть язык C - это обязательное использование библиотек.

В любом случае - это интерфейс между операционной системой и пользовательской программой, то есть запрос сервиса у операционной системы. Когда системный вызов обрабатывается, то переходит в привилегированный режим или в режим ядра. Системный вызов осуществляется командой программного прерывания (trap, INT), запрещенную команду иногда называют interact, но прямо её никто не использует, она всегда прикрыта каким-то стандартным кодом. В этом случае при обработке прерывания система распознает, что это на самом деле обращение к какому-то сервису ядра, работает код операционной системы в контексте того процесса, который выполнил системный вызов. Он имеет полный доступ к памяти пользовательской программы. Программное прерывание - это **синхронное событие**, оно происходит внутри процессора при выполнении программы, при этом происходит переход в режим ядра и обрабатывается нужный сервис.

Лекция 2. Архитектурные особенности и классификация операционных систем

Прерывания. Файловые системы

На прошлой лекции мы рассмотрели, как происходит обращение из пользовательских программ к ядру операционной системы. Несмотря на то что было много названий способов обращения из приложения к ядру - суть одна: среди машинных инструкций выделяется один запрещенный код операции, который принято называть trap или interact, при выполнении происходит внутреннее прерывание, ядро операционной системы, которое обрабатывает прерывания, о которых мы сегодня поговорим более подробно, распознает код операции и понимает, что это в действительности попытка приложения обратиться за каким-то сервисом к ядру операционной системы. У разных системных вызовов есть определенные соглашения о том, как пользовательская программа придает параметры, обычно они готовятся в стеке/stack пользовательского процесса. Ядро имеет полный доступ к любой памяти или памяти любого процесса, оно как бы продолжает выполнение того же процесса, в котором произошло это прерывание, достает аргументы вызова из стека и обрабатывает системный вызов.

Прерывания делятся на два класса - внешние и внутренние. **Внешнее прерывание** - это способ внешних устройств, внешнего оборудования, которые не входят в состав центрального процессора компьютера, проинформировать его о том, что возникло какое-либо событие, требующее немедленной реакции, либо сообщает о завершении асинхронной операции ввода-вывода. Это могут быть события, которые означают, что в устройстве возникли какие-нибудь неполадки и необходимо, чтобы ядро операционной системы обратило на это внимание и устранило их с помощью специальных обращений к этому же устройству (у каждого устройства есть специальные операции, которые его приводят в чувство, если оно в состоянии в него прийти). Либо это сообщения о том, что ранее запущенные операции на внешнем устройстве завершились, тогда ядру (обычно это делают специальные компоненты ядра, которые последние десятки лет принято называть драйверами устройств, ранее их называли программами обработки прерываний или программами управления устройствами) или драйверам необходимо обработать это событие, либо запустить следующую операцию на этом устройстве, если у него такие имеются в очереди, либо сообщить в пользовательский процесс, от имени которого эта операция была инициирована, о том, что эта операция завершилась (успешно или неуспешно).

Отдельным классом, хотя он состоит всего из одной сущности, являются прерывания по таймеру. В любом случае **прерывание таймера** - это способ дать знать процессору о том, что на астрономических часах "протикал" некоторый промежуток времени. Таймеры были устроены по-разному в разное время: когда-то все существовавшие компьютеры просто получали некоторый тактовый сигнал, обычно это

было 50 Гц, то есть процессор прерывался 50 раз в секунду, потому что ему было необходимо прибавить к своему времени ещё одну пятидесятую доли секунды; в настоящее время таймеры устроены более сложно и хитро, там можно внешнему устройству задать тот промежуток времени, через который он должен сообщить о том, что это время прошло, можно задать не один, а сколько угодно промежутков. Так или иначе, аппаратные прерывания, которые называются внешними - это события **асинхронные** по отношению к тем программам, которые выполняются на процессоре. Они происходят в любой момент времени, но в действительности не совсем, потому что это может управляться ядром операционной системы, процессор волен или не волен обращать на них внимание, если он хочет, чтобы внешние устройства работали нормально, то должен обращать внимание обязательно.

В этом курсе не будет отдельно рассматриваться **управление устройствами**, поэтому мы частично поговорим об этом сейчас, частично - в процессе курса, потому что управление устройствами в очень большой степени связано с тем, как обрабатываются внешние прерывания. Начнем с самого начала, с того, как они обрабатывались на заре этого класса средств вычислительной техники, которые умели работать с прерываниями и проектировались именно в расчете на то, что внешние устройства работают с процессором асинхронно и могут сообщать о себе асинхронным образом через внешние прерывания. На БЭСМ-6 было запаяно два физических адреса, на один из которых процессор автоматически переходил, когда возникало внешнее прерывание от какого-то устройства, это был как бы безусловный переход со стороны на физический фиксированный адрес. При этом при выполнении этой операции на БЭСМ-6 автоматически устанавливался привилегированный режим выполнения команд, то есть мы автоматически попадали в некоторую фиксированную точку ядра - это была точка, которая называлась "первичным входом в обработку внешних прерываний". Начиная с этой точки, ядро операционной системы БЭСМ-6 (которых у нее было несколько, все они делали это единообразно) начинало разбираться, что же все-таки произошло. Для этого был специальный аппаратный регистр, доступный только в привилегированном режиме, который назывался регистр внешних прерываний", операционная система смотрела - какой класс, то есть какое устройство выдало это внешнее прерывание по этому регистру (там стоял флажок, то есть единичка), после чего с помощью дополнительных регистров, которые были близки этому устройству, начинало разбираться, что произошло. Это была полностью программная поддержка обработки внешних прерываний. Прерывания от таймера были частным случаем внешнего прерывания, то есть 50 раз в секунду, чтобы не происходило, такая дешифрация повторялась просто из-за того, что внешний таймер запрашивал прерывание.

Аналогичным образом обрабатывались внутренние прерывания, про которые мы будем говорить далее. То есть был другой вход в то же самое ядро операционной системы, в частности так были устроены **экстракоды** - это внутренние прерывания, то

есть переход на фиксированный физический адрес внутри ядра, которое далее разбирается, что же случилось, и предпринимает необходимые действия.

Этот механизм был плох тем, что ядро в программном режиме делает множество работы, которую можно было бы каким-то образом возложить на аппаратуру, чтобы она поточнее сообщала: кто стучится в ядро, с какой целью, что необходимо по этому поводу сделать. Повторим: первый шаг - это фиксированный адрес и автоматическая установка привилегированного режима, соответственно, последней командой при обработке внешнего прерывания была команда возврата прерывания - это аппаратная команда, которая выбрасывала в ту точку программы, в которой внешнее прерывание возникло, где прервалась пользовательская программа, которая продолжалась дальше с этой точки. В действительности - это довольно грубая схема, А.Н. Томилин, который был у истоков первых операционных систем БЭСМ-6, в основном он написал первую операционную систему, которая называлась Д-68, которая была написана в машинных кодах, когда ещё не было языка Assembly.

Первый серьезный шаг на пути к разгрузке ядра операционной системы от этой рутинной работы был сделан в семействе суперкомпьютеров IBM System/360, где было введено на аппаратном уровне 8 классов внешних прерываний, для каждого класса был отдельный критический вход, критический адрес, запаянный для входа в ядро. Опять же, отдельный класс составляли прерывания таймера, как наиболее частый, отдельный класс, отдельный обработчик и т.д. Отдельный класс составляли дисковые устройства, потому что они требуют достаточно серьезной и быстрой реакции - система должна следить за ними в первую очередь. Фактически схема была такая же, как и на БЭСМ - 6, только немного меньше дешифрации на программном уровне в ядре, аппаратура немного помогала, чтобы не было необходимости начинать с нуля: смотреть на общий регистр прерываний, выделять там bitrix, смотреть к какому классу это относится, какой вызывать обработчик и т.д. Путь IBM немного жизнь облегчал. Наконец первой ввела в обиход ту схему, которая используется по настоящее время, компания Digital Equipment Corporation (DEC), они придумали механизм, который называется **"вектор прерывания"**. Идея чрезвычайно простая: вместо того, чтобы разбивать внешние прерывания на классы и каждому классу назначать свой адрес входа в ядро, механизм векторов прерывания позволял каждому устройству (в действительности каждому контроллеру или группе контроллеров) назначать свой адрес входа в ядро. То, что находилось по этому адресу в DEC, стали называть вектор прерывания - это было машинное слово, включающее всего 2 команды, из которых первая команда - была командой установки уровня выполнения процессора, вторая - командой перехода на обработчик прерывания, на соответствующий вход драйвера (как они уже стали называться с того времени). Обратим внимание на то, что все это произошло чрезвычайно быстро, потому что первые PDP-11 появились в конце 70-х годов. Если учесть, что БЭСМ - 6 и её родичи того же поколения появились в 60-е, IBM System/360 в конце 60-х, а механизм векторов прерываний появился меньше, чем через 10 лет.

Привилегированный режим. Уровень выполнения процессора: самое простое - это "да" или "нет"/двоичная логика, то есть мы либо разрешаем процессору выполнять только пользовательские команды, которые могут быть в пользовательских программах, либо мы говорим, что могут выполняться все команды, какие есть в системе команд процессора, и этот режим привилегированный. Оказалось, что этот режим не очень удобен, потому что в действительности в зависимости от специфики внешних устройств требуются разные набора привилегированных команд. Начиная с PDP-11, было введено 8 уровней привилегированности с самого высокого - нулевого, заканчивая самым низким, пользовательским - седьмым. Программисты операционной системы решали на каком уровне привилегированности надо обрабатывать прерывания от соответствующего класса устройств - таким образом формировались вектора прерываний. Уровень привилегированности означает больше. Привилегированный режим на БЭСМ-6: тот режим, который аппаратно входил в процессор после внешнего прерывания, был не только привилегированным, но ещё был запретом на прерывание, потому что иначе на обработке прерываний можно было зациклиться. После того, как возникало внешнее прерывание, на какое-то время ядро работало так, что прервать его никакое устройство не могло. В какой-то момент, оставаясь в привилегированном режиме, оно могло сказать, что теперь прерывания разрешены - это означает, что зацикливания гарантированно не будет. Принцип этого уровня выполнения процессора означает следующее: если процессор выполняется на уровне i, где i меняется от 7 до 0, то никакое устройство, которое запрашивает прерывание с уровнем привилегированности больше i, то есть с меньшим уровнем - пропускаться не будет до тех пор, пока драйвер не скажет, что он снимает уровень привилегированности и остается только в режиме ядра для того, чтобы было можно выполнять привилегированные команды.

В действительности при этих условиях, при наличии такого механизма - написание программы, которая позволяет работать с устройством, становится чрезвычайно простой задачей. Самое сложное в драйвере - это помнить позиции флажков, которые необходимо правильно устанавливать и проверять. Основная сложность любого драйвера заключается в том, что должно быть очень хорошее и точное описание возможностей аппаратуры. Написание драйвера отличается от написания основного кода ядра операционной системы тем, что в ядро никого со стороны не пускают, как в случае Linux, где один человек - Линус Торвальдс либо разрешает, либо не разрешает вносить правки в ядро, больше никто это сделать не может. Драйверы, поскольку они допускают отдельное подключение к ядру, пишутся многими, поэтому в них ошибок больше, чем в ядре. В Институте точной механики и вычислительной техники давно существует "Центр верификации Linux", который в основном занимается верификацией драйверов, потому что их очень много, как и устройств. Поскольку все драйверы работают в той же памяти, что и все программы ядра, то из драйвера, к сожалению, порушить ядро довольно легко, не смотря на все технические ухищрения, которые применяются в Linux для того, чтобы защитить

основную часть ядра. Интересно, что одна из статей Эндрю Таненбаума была про работу операционной системы Minix, для которой придумывались специальные механизмы восстановления работы системы после того, как она потерпела крах из-за ошибки в драйвере. Это очень трудно представимая задача - когда в операционной системе ошибка, а система восстанавливается после ошибки только за счет того, что эта ошибка не в основном ядре, а в драйвере.

Внутренние прерывания, которые по-другому называют исключительными ситуациями, а иногда exception - это исключительные ситуации, это синхронные события, которые возникают внутри процессора при выполнении пользовательских программ. Если устройство управления распознает команду с недопустимым кодом операции или недопустимого фермата - если на аппаратном уровне выявляется попытка доступа к некоторому ресурсу при отсутствии достаточных привилегий или - если при работе с виртуальной памятью какая-то программа пытается обратиться к странице виртуальной памяти, которая отсутствует в памяти физической. Trap и INT (прерывания, которые возникают при системных вызовах) тоже относятся к этому классу - это прерывания, которые возникают при выполнении команды с недопустимым кодом операции. События синхронные, они возникают при работе программы, прерывают её выполнение и могут быть двух классов: **исправимые и неисправимые исключительные ситуации**.

- **Неисправимые исключительные ситуации** возникают тогда, когда ядро сделать ничего не может, например, предположим, что основная память работает с контролем четности - это традиционный подход в дешевой основной памяти, когда она занимает 64 разряда, а в действительности память хранит 65 разрядов, дополнительный разряд - это циклическая поразрядная сумма 64 разрядного слова, которое пишется в память. Когда происходит чтение из памяти, которое соответствует 65 разрядам - система делает циклическое побитовое сложение памяти и проверят, совпадает ли получившийся бит с битом четности или не совпадает. Бит четности называют так потому, что при четном числе единиц в слове - это 0, при нечетном - это 1. Это очень простая и дешевая схема. Что делать ядру, если возникает прерывание, которое поступает из устройства управления памятью - что у него сработал контроль четности при обращении к слову памяти? В этом случае ничего сделать нельзя, это означает, что память сбоит, если начинает сбоить основная память, то процессору работать не должно, а необходимо поменять память либо как следует её протестировать. В случае неисправимых исключительных ситуаций самое разумное решение для операционной системы - это выйти на аппаратный стоп, на команду halt и перестать работать. Другая ситуация: представим, что в программе возникает деление на 0, в этом случае внутреннее прерывание вырабатывает арифметическое устройство процессора, которое распознает, что оно никак не может выполнить эту операцию. Но выходить "на стоп" при этом неразумно, в действительности необходимо выбросить ту программу, в которой

возникло деление на 0. Сама программа ничего сделать не может, она никаким образом не может это исправить, потому что деления на 0 быть не должно, но закончить её выполнение - это возможно и разумно, с диагностикой.

- **Исправимые исключительные ситуации** – это нормальное явление. Могут быть такие внутренние прерывания, которые вызываются ошибками в программе, на которые сама программа может разумным образом отреагировать, если у нее есть обработчик исключительных ситуаций. В этом случае система распознает, что это исправимая ошибка (причем исправимая без принудительного закачивания этой программы) и говорит программе об этом. Если программа на это реагировать умеет - хорошо, если нет, то она закончит свое выполнение командой exit. Абсолютно нормальной ситуацией является внутреннее прерывание по отсутствию страницы виртуальной памяти - это обычный случай при управлении виртуальной памятью. Не просто так основной механизм управления ею называется paging by demand/запрос по требованию, то есть прерывание, которое возникает при работе процесса, который пытается обратиться к странице виртуальной памяти, отсутствующей физической памяти - операционная система интерпретирует как неявный запрос новой страницы от процесса. Эту ситуацию можно не называть исключительной, page fault - это неудачное обращение к странице.

Внутренние прерывания обрабатываются точно так же, как и внешние: к каждой категории внутреннего прерывания привязан свой вектор прерывания, то есть дополнительной дешифрации внутри ядра никакой не требуется. Это зависит от того, как это сделали конфигураторы самого ядра операционной системы, как они привязали вектора прерывания к категориям синхронных событий. Можно подумать, что в операционных системах все делается тривиально, но это, конечно, не так, потому что есть масса технических проблем, которые в этих случаях приходится решать, частично мы их рассмотрим.

Файл - очень важное понятие, под файлом понимают именованную часть пространства на носителе информации. Когда-то был только единственный вид внешней памяти, на которой можно было сохранять свою работу - магнитные ленты (речь идет о программистах) без какой-либо помощи со стороны операционной системы. Лента в нашем распоряжении, есть набор блоков или секторов, мы имеем экстракоды, такие же как системные вызовы, которые позволяют читать ленту и писать на неё. Программисты были вынуждены хранить каждый проект на отдельной магнитной ленте, структура которой могла быть довольно сложной: несколько версий программы, несколько версий откомпилированного кода, несколько версий тестовых данных и пр. Для того, чтобы помнить нахождение всего этого на ленте - использовались наборы перфокарт, потерять перфокарту было катастрофично. Одна перфокарта в таком режиме служила 1-2 года, это была чрезвычайно полезная для разных целей вещь, использование её для ввода данных - это одна небольшая

составляющая для парка использования. Первичная цель файловых систем состояла в том, чтобы перестать заставлять людей самих следить за тем, где у них что лежит, чтобы они могли положиться на часть операционной системы, чтобы она делила память и связывала с теми именами, которые удобны пользователям. У Минск-32, которые делались в Беларуси, была файловая система на магнитных лентах, которые не приспособлены для этой цели, в отличие от магнитных дисков.

Главная задача **файловой системы -** скрыть особенности ввода-вывода, если мы работаем с файлами, то мы совершенно не обязаны знать о том, как устроено дисковые устройства и их контроллеры. Задачей файловой системы также является обеспечение абстрактной модели файлов, независимых от устройств - это утверждение истинно для первых файловых систем, а они были сделаны компанией IBM, которая придумала магнитные диски, в IBM 360 файловая система уже существовала. У программистов на протяжении многих лет была проблема: как написать программу, которая неким образом общается с внешним миром, но не привязывать её при этом к конкретным типам устройств? Чтобы ей можно было сказать, с чем мы будем работать, когда будем её выполнять, а не тогда, когда мы её пишем. Это требуется часто, например, какая-то программа считает и выводит результаты, при одном запуске - удобно, чтобы она выводила их на экран монитора, при другом - чтобы она их печатала на принтере. Сейчас может показаться, что это делается тривиально, но это происходит потому, что в свое время это сделал Unix, хотя и не очень универсально. Первая попытка решить эту проблему была предпринята в IBM 360, тогда было придумано понятие "файл", а также было разрешено использование этого понятия в достаточно абстрактном смысле, не привязанном к конкретной реализации того, что это такое. Было разрешено использовать файл как именованный файл внешней памяти. Когда программа готовилась к запуску, было можно сказать, что именно сейчас под этим именованным файлом мы будем понимать устройство передачи данных по каналу связи (если это, конечно, файл, в который происходит запись) или мы будем понимать - как устройство отображения на мониторе, и т.д. Идея была вполне здравая, но не было учтено, что конкретное понятие файла как именованной области внешней памяти - довольно специфично, у него вполне определенный набор и системных вызовов, и параметров, и пр. Если мы пытаемся сделать его абстрактным, то фактически приходится вводить в потенциальный набор параметров все параметры, которые могут быть у каждой конкретной реализации, то есть одни для канала связи, другие для монитора и т.д. Из-за этого понятие файла в IBM стало очень громоздким, что привело к использованию Job Control Language (JCL)/языка управления заданиями, на котором при запуске программы было необходимо написать, что понимается под каждым файлом, который используется в запускаемой программе. Это было громоздко и неудобно и для программистов, и для тех людей, которые программу запускали.

Основная революция Unix - это представление данных как потока байтов, то есть без всякой структуры. Файлы IBM - это были не просто области внешней памяти, а структурированные области, каждый файл состоял из набора записей (C или

Pascal) - структуры с именованными полями. У каждого файла была своя структура записи и можно было объявлять файлы чисто последовательными - это последовательность записей, когда можно писать только в хвост памяти, а читать только сначала, это файлы прямого доступа, которые позволяли обращаться к записи просто по её номеру, а также индексно-последовательные файлы, когда какое-то поле записи объявлялось ключевым, и система позволяла обращаться к файлу, указывая значение ключевого поля интересующей нас записи. Это ещё более усложняло универсальное использование понятия файла как замену любого устройства. В Unix файлы тривиальной структуры - это последовательность байт, довольно много устройств тоже работает с последовательностью байт или можно интерпретировать их работу как последовательность байт: это каналы связи, устройства вывода на печать и т.д. Поэтому в Unix было два основных решения: во-первых, все устройства, которые поддерживаются драйверами операционной системы Unix - это файлы, которые называются специальными файлами, об этом мы поговорим, когда будем рассматривать файловые системы, они имеют такие же интерфейсы как у обычных файлов, то есть это именно последовательность байт. Поэтому замена файла внутри Unix на устройство - это тривиальная замена, по сути мы вместо файла, который является областью внешней памяти - говорим, что мы будем работать с файлом, который является специальным файлом, то есть устройством. Это стало возможным благодаря тому, что файлы очень простой структуры. Я считаю, что лучше всех в свое время это было сделано при создании вычислительных комплексов АС-6 в 70-х - начале 80-х годов под впечатлением от языков с абстрактными типами данных.

В начале 70-х годов прошлого века возникли два языка, один из которых назывался **CLU**, а другой **Alphard**. Alphard был разработан Уильямом А. Валфом, Ральфом Л. Лондоном и Мэри Шоу, а CLU делала одна из самых известных женщин в компьютерном мире - **Барбара Лисков**. Её основная идея была настолько свежая, если бы её впервые открыли сегодня, то она бы совершенно не выглядела устаревшей. Лисков предложила две вещи:

1. Расширить язык программирования средствами, которые позволяют определять новые типы данных. Тип данных, как мы понимаем это в обиходе - это, прежде всего, множество значений, которое может быть явно выписанным. Примером типа данных, у которых явно выписываемое множество значений - это перечислимые типы/ enumerated types. Мы говорим, что определяем тип, значение которого, например - каждый охотник желает знать, где сидит фазан/цвета радуги: красный, оранжевый, желтый, синий, фиолетовый - это множество значений. В типе данных существует набор операций, который работает над этим множеством значений. Последнее - это литералы, то есть как можно обращаться к этим значениям со стороны. Если мы хотим где-нибудь обратиться к значению пять из типа целых чисел, то мы просто пишем строку из одного символа 5 - это литерал, видя эту пятерку, система понимает, что необходимо взять значение 5 из множества значений целого типа. Барбара

Лисков предложила сделать так, чтобы было можно на уровне программиста определять множество значений и определять операции, а главное, что после того, как это программистом проделано, любым типом данных можно пользоваться так, как если бы он был предопределен в системе в этом языке программирования.

2. Вторая часть идеи ещё более сильная - Барбара Лисков предложила разделять в определении такого типа данных (которые следует называть абстрактными типами данных) фазу спецификации, когда мы на фазе спецификации фактически говорим - какое множество значений у этого типа данных. Грубо говоря, операциями назначения этого типа сигнатурой - это название операции, тип возвращаемого значения и набор типов параметров операции - это спецификация. Вторая часть - это реализация, когда пишутся все коды операции. Лисков говорила, что это хорошо, потому что, если мы определили какой-то абстрактный тип на уровне только спецификации - мы уже можем писать программы, которые пользуются этим типом данных. Более того, для одной спецификации может быть несколько реализаций, и замена одной реализации на другую не должна влиять на выполнение программы, которая использует спецификацию этого абстрактного типа данных.

Барбара Лисков предложила это 1973 году, идея настолько воодушевила, что была сделана операционная система, в которой были использованы понятия настраиваемого модуля, который был фактически привязанным к операционным системам в вариации абстрактного типа данных. Было можно писать программу, которую использую некоторые устройства, управляемые некоторыми настраиваемыми модулями, все модули делились на спецификацию и реализацию, при запуске и даже при выполнении программы было можно сказать - какую реализацию мы хотим видеть. Очевидно, что это развитие идеи IBM, потому что мы фактически подменяем одно понятие другим, но только гораздо более универсальным и гибким, потому что вместо одного понятия файла мы можем разрешить сколько угодно такого рода понятий, которые имеют гораздо больше общего, чем разные инкарнации файлов у IBM.

Файловые системы - часть ядра операционной системы, если говорить про традиционные файловые системы, так оно в Linux, в Unix, в Microsoft. В текущих файловых системах основные системные вызовы - это создание файла, уничтожение файла, открытие файла как начало сессии работы с файлом в данном процессе, закрытие файла, как конец такой сессии, чтение файла, запись файла/create, delete, open, close, read, write. Для того, чтобы файлы можно было гибким образом именовать, а это уже было у IBM - использовалось понятие составных имен, то есть каждый файл - это цепочка простых имен, где первое имя - это имя некоторого элемента, предопределенного корневого каталога, в этом элементе написано - где искать следующее имя, и т.д., пока мы не дойдем до файла (каталог, текущий каталог, корневой каталог, путь). Это очень ценное изобретение для одного пользователя,

которое позволило работать с файлами удобно, страшно подумать, что творилось бы без многоуровневого наименования даже на ноутбуке, потому что найти что-либо в 20-50 тыс. файлов - невозможно. Для многопользовательских систем очень важным понятием является "текущий каталог", то есть каталог, который временно используется в качестве корневого. Путь - это цепочка имен, которая позволяют дойти от корневого каталога до того фала, который нас интересует. В Unix традиционно, что очень правильно, самым низким уровнем для работы с файлом является библиотека stdio - стандартная библиотека ввода-вывода, которая более "человеческая", чем библиотека системных вызовов для работы обычных пользователей.

В Computer Science имеется несколько понятий, которые, видимо, невозможно определить, например, к таковым относится понятие "объект" в смысле объектно-ориентированного подхода или объектно-ориентированного программирования. Сказать, что такое объект никак не получается, не используя какого-нибудь другого названия этого же понятия. Например, в объектно-ориентированном моделировании объект определяется как некоторая сущность, которая делает некоторые вещи. Сущность - это другое наименование того же самого, интересно, что в моделях сущность-связь понятие "сущность" определяется как объект с некоторыми свойствами, потому что объект определить в действительности невозможно. Есть один специалист, который считает, что возможно дать формальное определение объекта, но это делается крайне сложно, доходя до λ-исчисления. При этом от этого никак не становится легче, потому что λ-исчисление - это очень низкоуровневый, очень мощный механизм, с использованием которого можно сделать массу вещей, но понять их невозможно. Понятие "процесс" из той же категории, в следующих лекциях мы рассмотрим понятие "thread"/понятие управление потоками или нити.

Архитектурные подходы к организации операционных систем

Монолитное ядро. Первая организация, когда все компоненты являются не самостоятельными модулями, а составными частями одной большой программы, которая называется **ядро**/operating system kernel. Это, с одной стороны, очень хорошая организация - монолитная, в этом случае у всех программ ядра имеется общая память, все компоненты являются составными частями одной программы, используют общие структуры данных (общие таблицы, разнообразные описатели и т.д.). Главное, что у разных компонентов очень дешевый способ взаимодействия, они часть одной программы, поэтому могут обращаться и взаимодействовать друг с другом путем непосредственного вызова подпрограмм, то есть обычного перехода с возвратом аппаратной команды. Плюсы - это предельная эффективность, минусы - это громоздкость, если по каким-то причинам требуется изменить какие-то компоненты ядра, добавить в ядро новые компоненты или исключить неиспользуемые, то в обязательном порядке требуется как минимум перекомпиляция, скорее всего, потребуется ещё длительная работа по настройке.

Монолитное ядро - это старейший способ организации операционных систем, который применялся исключительно во всех ранних операционных системах (IBM, PDP, БЭСМ-6, и был унаследован в Unix). Основные ветки операционной системы Unix просуществовали с монолитным ядром практически до настоящего времени, фактически монолитное ядро осталось и в текущем Linux, но с некоторыми, довольно существенными усовершенствованиями: появилась возможность динамической подгрузки компонентов внутри ядра путем головоломной техники, которая не опирается на какую-то архитектурную поддержку. Это то, что касается возможности структуризации, потому что, чем старше операционная система - тем больше у нее ядро, чем больше ядро - тем больше в нем ошибок, тем сложнее его сопровождать, тем сложнее его модифицировать и т.д. Монолитное ядро было хорошо на заре операционной системы Unix, вероятно, сейчас это унаследованная и не очень хорошая комбинация. Но уверенности, что Линус Торвальдс когда-нибудь решится полностью изменить структуру ядра Linux - нет.

Слоеные системы. Это идея, которая впервые возникла у Эдсгера Дейкстры применительно к операционным системам или возникла ближе к сетевым протоколам. Модель **OSI/ISO** устроена следующим образом: это стек протоколов, который состоит из 7 уровней, известно с каким уровнем он взаимодействует, для каждого уровня он определяет набор сервисов, которым можно пользоваться программам более высокого уровня, они пользуются только сервисами программ более низкого уровня, которые на нем определены. Та же идея лежала в архитектуре операционных систем, которую в 1968 году предложил **Эдсгер Дейкстра** (Edsger Wybe Dijkstra), которую он назвал система **THE**. Это аббревиатура от Технического университета Эйндховена/Technishe Hogeschool Eindhoven, где он тогда работал. Идея состояла в том, что вся операционная система разбивается на ряд уровней с хорошо определенными связями между ними, чтобы объекты уровня N могли вызывать только объекты из уровня N-1, то есть ровно тоже самое, что и в OSI/ISO. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль.

Система имеет следующие уровни:

5. Интерфейс пользователя
4. Управление вводом-выводом
3. Драйвер устройства связи оператора и консоли
2. Управление памятью
1. Планирование задач и процессов
0. Hardware

Слоеные системы хорошо реализуются, потому что каждый слой можно программировать изолированно от других. При программировании операционных систем это является стандартной практикой - это практика заглушек. Если мы хотим какой-то компонент операционных системы отлаживать автономно, то мы должны сделать заглушку (вместо того, чем этот компонент будет пользоваться) - простую

модель, которая будет возвращать правдоподобные результаты на обращение, при этом её не надо отлаживать, так как она слишком проста для этого. Слоистые модели хорошо тестируются. Чем больший объем кода необходимо тестировать, тем тестирование менее надежно, тем меньше его покрытие. Такие системы легко модифицируются, каждый уровень модифицируется отдельно от других. Но они менее эффективны, чем монолитные и достаточно жесткие, потому что созданная иерархия требует соблюдения последовательности. Сегодняшние операционные системы по уровню не вписались бы в иерархию по уровням, неизвестно - возможно ли создать сегодня такую слоистую архитектуру для систем класса Lunux или Windows 10.

Виртуальные машины. Это несколько другой подход в смысле операционных систем, который предназначен для того, чтобы на базе одного экземпляра аппаратуры и некоторого специального программного обеспечения, которое поддерживает виртуализацию, организовать несколько копий аппаратуры обеспечения, включая процессор, привилегированные и непривилегированные команды, устройства ввода-вывода, прерывания и т.д. В такую виртуальную машину можно загрузить свою собственную операционную систему CP/CMS или VM/370. Если в машине где-то происходит нарушение, например, привилегированности, то происходит системный вызов реальной операционной системы, который разбирается с тем, что же произошло. Важно отметить снижение эффективности, громоздкость и разные приложения.

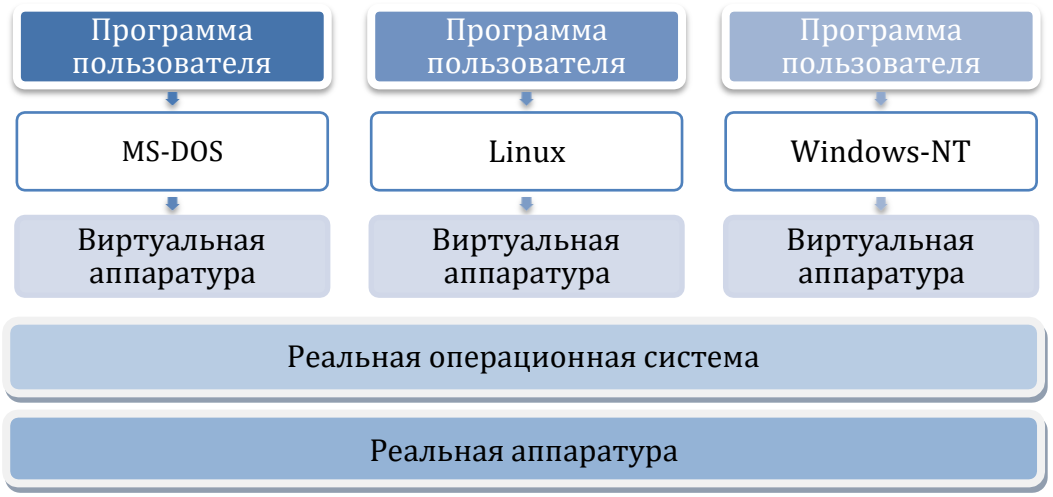

Рис. 2.1. Схема устройства виртуальной машины

Схема показывает, как может быть устроена виртуальная машина: есть слой реальной аппаратуры, на нее ложится слой, который выбирается в качестве основы для виртуализации - это некоторая реальная операционная система, над ней строится конкретная система, выше - программы пользователя.

В Microsoft сложно понять, какие корни у текущего варианта операционной системы Windows Архитектурно Windows-NT была, вероятно, самой интересной

разработкой - это была система, которая была сделана на принципах микроядерного ядра, то есть микроядерной организации. Когда эта система только вышла, то было озвучено намерение использовать набор системных вызовов Win64 как основы виртуализации, над которой будет построена поддержка POSIX - это был бы Unix, который строился бы над Windows-NT, над ним планировалось поддерживать MS-DOS (тогда это было вполне актуально) и поддерживать Win32 над Win64 в виде виртуальных машин. Тогда это называлось прикладными программными средами, но фактически это было тоже самое. Все было замечательно продумано и могло бы получиться, если бы тогда в Win64 не поместили всю графику Microsoft. После этого микроядерность поплыла, то есть микроядро стало таким громоздким, что эти идеи оказались не жизнеспособны.

Микроядерная архитектура. В этом случае вместо ядра операционной системы делается специальный модуль ядра, который обеспечивает некоторый минимальный набор сервисов, он называется микроядром. Большинство составляющих операционной системы являются самостоятельными программами, но только микроядро в такой системе работает в привилегированном режиме и обеспечивает, прежде всего базовый уровень взаимодействия между остальными программами, которые каким-то образом должны общаться, например, механизм передачи сообщений; обеспечивает управления процессором, то есть планирование использования процессора; первичную обработку прерываний (только самое начало), операции ввода-вывода (базовый уровень) и базовое управление памятью. Все остальные функции обычного ядра операционной системы выделяются в одеяльные менеджеры, иногда их называют серверами, которые программируются также, как обычные пользовательские программы, они сами взаимодействуют с микроядром и позволяют приложениям через него взаимодействовать с собой, это происходит с помощью механизма общения, который поддерживает микроядро.

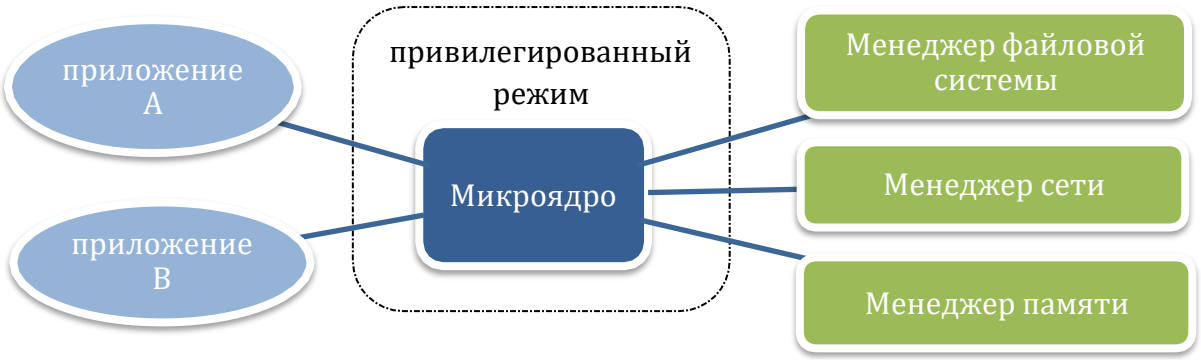

Рис. 2.2. Устройство микроядерной архитектуры

Самое главное, что они очень легко могут модифицироваться, потому что фактически в отдельные менеджеры выносятся те модули, которые в обычном монолитном ядре являются частью ядра, они обеспечивают высокую степень

модульности ядра, упрощается добавление новых компонентов, в таких системах можно легко и динамически загружать и выгружать новые драйверы, которые тоже (не считая базового уровня обработки прерываний) находятся на уровне пользовательских программ, файловые системы и т.д. Поскольку в этих серверах или менеджерах не используется привилегированный режим - упрощается процесс отладки, для нее можно использовать совершенно обычные отладчики, какими пользуются разработчики прикладных программ. Но микроядерная архитектура операционной системы вносит существенные накладные расходы, связанные с передачей сообщений, тут уже нет взаимодействия между компонентами с помощью обычных команд вызова программ, а все делается через механизмы передачи сообщений. Требуется тщательное проектирование.

В одном отношении микроядро отличается от обычного ядра:

- **Обычное ядро операционной системы** - это отдельно скомпонованный монолит, который работает со своей виртуальной памятью, но у него нет активности, то есть это просто построенный набор программ и данных. Ядро начинает работать только тогда, когда к нему кто-нибудь обращается, его можно считать динамически скомпонованной библиотекой, которая работает в привилегированном режиме. Либо она начинает работать, когда какая-то пользовательская программа выдает внутреннее прерывание, являющееся системным вызовом, тогда ядро работает от имени этой программы; либо это работа по прерыванию внешнего прерывания, то есть какое-то устройство запрашивает прерывание, программа прерывается, от её имени начинает работать ядро, которое прерывание обрабатывает. Это стандартный механизм, то есть ядро - это пассивный компонент операционной системы.
- **Микроядро - это активный компонент** - у него есть своя активность: оно может принимать сообщения, может посылать сообщения. В этом отношении микроядерные операционные системы устроены не так, как традиционные системы с обычными ядрами.

Прежде, чем мы рассмотрим, что происходит сейчас с микроядерными организациями, остановимся на том - что с ними могло бы произойти. В 80-е годы было несколько попыток создания микроядерных систем, которые имели шансы заменить Unix. Никто не заставляет в менеджерах повторять то, что делает Unix, возможности микроядра открыты и для приложений. У микроядра может быть свой набор системных вызовов, который совершенно не обязательно совпадает с системными вызовами Unix. Было 4 попытки создания подобной системы:

1. Система **Mach 3 -** первая хронологически, она была разработана в Университете Карнеги-Меллона. Это было микроядро, которое имело абсолютно все шансы заменить Unix, оно было достаточно большим и поддерживало около 250 системных вызовов.

2. Очень интересная работа по созданию микроядерной системы была выполнена в 80-е годы во Франции – система **Chorus**, которую развивала Chorus systems. Ядро называлось Nucleus. Она делалась при поддержке INRIA и имела следующий подход: для того, чтобы вокруг микроядра можно было легко наращивать разные среды, внутри него были реализованы все механизмы общения процессов, которые тогда были известны, над микроядром уже на прикладном уровне было можно строить разные системы (с разными видами процессов, с разными видами взаимодействий и т.д.). Была сделана поддержка TCP/IP на уровне микроядра - можно было строить окружение, имея ввиду, что сетевая поддержка внутри микроядра обеспечена. Одной из проблем разработчиков микроядерных архитектур было то, что, с одной стороны, всем было понятно, что предлагается некоторый вариант построения операционных систем, который лучше Unix, что необходимо уговорить мир разработчиков приложений перейти на новый вид операционной системы, с другой стороны - все понимали, что такая потребность у них отсутствует. Поэтому сначала все пытались сделать над микроядром свою реализацию Unix, чтобы привлечь пользователей с целью дальнейшего перехода на свой интерфейс, но из этого, к сожалению, ничего не вышло. В Chorus была сделана реализация Unix, которая полностью соответствовала стандарту, считалась, что это лучшая реализация в Европе, то есть это был хороший, чистый Unix. В 1997 году Sun Microsystems купила Chorus Systemes, хорошо отлаженная разработка Chorus по каким-то причинам попала в SUN в подразделение встраиваемых систем. С того времени, когда SUN была куплена компанией Oracle, о Chorus ничего не известно.

3. Российский проект делался в одно время с Chorus и назывался **КЛОС** - кластерная операционная система. Идея состояла в том, что микроядро поддерживает не просто механизм обмена сообщениями, а совершенно оригинальный механизм поддержки асинхронно выполняемых кластеров. Каждый кластер - это разновидность экземпляра абстрактного типа данных или объекта какого-то класса, только со своей собственной виртуальной памятью, со своей защитой, которая поддерживалась микроядром. КЛОС была очень интересной операционной системой, которая ни на что не была похожа. Разрабатывалась система 5 лет и никем не финансировалась, на совместном производстве завода ЗИЛ и НИИСИ Академии наук СССР делались маленькие рабочие станции "Беста", в них использовались процессоры Motorola и Unix, на них и была установлена КЛОС. Наряду с Unix на Бесте над КЛОС работал Unix.

4. **QNX** - принятого говорить, что это POSIX-совместимая система реального времени, в действительности это собственная реализация Unix, реального времени как такового в ней никогда не было, но она умеет достаточно реактивно обрабатывать внешние прерывания, то есть у нее есть некоторое гарантированное время на их обработку. QNX - исторически микроядерная система, в 80-е - начале 90-х годов это было самое маленькое микроядерное ядро

в мире, оно занимало 16 Кбайт кода, над ним был развернут полномасштабный Unix в виде наборов компонентов. Не очень крупная канадская компания могла бы выжить только при условии, что она очень сильно занимается маркетингом и подстраивается под запросы пользователей. Компания жива, но микроядро давно стало просто ядром.

Смешанные системы. В современных операционных системах используются различные комбинации подходов. Ядро Linux - это монолитная система с элементами микроядерной архитектуры, которые состоят именно в том, что появилась возможность динамической загрузки и выгрузка модулей. **BSD 4.4** (Berkeley Software Distribution) - это BSD-UNIX - общее названия вариантов Unix, восходящих к дистрибутивам университета Беркли. В конце существования Советского Союза была организована "Ассоциация пользователей операционной системы Unix", на ежегодной конференции ассоциации эксперты университета Беркли рассказывали про планы относительно BSD 4.4, которые основывались на микроядре Mach. Эта система в то время была на выходе, но её подкосила жалоба AT&T Inc. - вотчины Unix, места откуда вышли **Денис Ритчи и Кен Томпсон** и др. AT&T Inc. - крупный американский транснациональный телекоммуникационный конгломерат со своими традициями и сильными юристами, а BSD 4.4 - это свободная операционная система, в которой было много нелицензированного кода из старого Unix AT&T. На них подали в суд, который продолжался несколько лет, это сильно повредило работе над BSD 4.4. Этот суд сыграл на пользу Linux, потому что в то время Unix BSD точно был лучше Linux. Возможно, и сейчас он лучше, но по каким-то неизвестным причинам первый используется очень активно, а второй менее активно, хотя это более прогрессивная система, которая сделана на более правильных архитектурных решениях. MkLinux (Microkernel Linux) - это отдельная работа.

Микроядро - управление виртуальной памятью и работа низкоуровневых драйверов, все остальные функции осуществляются монолитным ядром. У BSD на основе Mach - условное микроядро, остается почти целиком монолитное ядро, а не набор сервисов, которые над ним работают. Это оправдывается тем, что они стремятся использовать преимущества микроядерных архитектур, по возможности сохраняя хорошо отлаженный код монолитного ядра.

Наиболее тесно элементы микроядерной архитектуры и элементы монолитного ядра переплетены в ядре Windows NT. Микроядро NT слишком сложно и велико (более 1 Мб). Компоненты ядра Windows NT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений, все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных. Чисто микроядерный дизайн коммерчески непрактичен, так как слишком неэффективен.

Классификация операционных систем

Существует несколько схем классификации операционных систем, например, классификация по некоторым признакам с точки зрения пользователя. Классификация исторически носит условный характер. Реализация многозадачности операционной системы имела два класса:

- **Многозадачные/многопользовательские** операционные системы - Unix, OS/2, Windows.
- **Однозадачные/однопользовательские** операционные системы, к ним относились те системы, которые работали на первых вариантах персональных компьютеров, например, MS-DOS.

Приблизительность классификации: даже в MS-DOS было можно иметь выполняемым больше, чем одни процесс, можно организовать запуск дочерней задачи и одновременное сосуществование в памяти двух и более задач.

OS/2 - отличная операционная система IBM, которая в действительности должна была существовать вместо Windows, у неё для этого были все шансы. Частично от OS/2 в свое время пошли Macintosh, интерфейс Apple также происходит от неё. Windows 3.x и 3.2 были однопользовательской системой, Windows NT и Unix - многопользовательские операционные системы, они могут работать на компьютерах, в которых подключено несколько рабочих мест, у них имеются механизмы защиты персональных данных каждого пользователя. Эти системы и сейчас многопользовательские, при этом Linux в этом режиме реально используется только на серверах, в центрах данных.

Многопроцессорные и однопроцессорные операционные системы. Сегодня не многопроцессорной обработки, вероятно, нет вообще, потому что после того, как все процессоры стали многоядерными, операционная система не может не поддерживать такую работу. Многопроцессорные операционные системы разделяют на симметричные и асимметричные, сейчас все системы симметричные. В симметричных системах на каждом процессоре функционирует одно и то же ядро и задача может быть выполнена на любом процессоре, все ядра работают на равных, нет главных и подчиненных. В асимметричных мультипроцессорах процессоры неравноправны, существовал главный процессор (master) и подчиненные (slave). К таким относились компьютеры компании Control Data Corporation (CDC), которая делала очень продвинутый в своё время компьютер **Cyber**. Они были многопроцессорные, без прерываний, все работало под управлением основного процессора-мастера, следящего за остальными процессорами и за внешними устройствами.

Системы реального времени. Есть два класса систем, которые принято называть real-time operating system, RTOS:

1. **Системы жесткого реального времени** критически важны и используются для управления различными техническими объектами или технологическими процессами. Например, для управления ядерным реактором, где необходимо следить, чтобы стержни находились на высоте, чтобы была нормальная температура и т.д. Есть системы, которые управляют взлетом ракеты, отслеживают как работают двигатели. Системы жесткого реального времени характеризуются предельно допустимым временем реакции на внешнее событие, если такая система не отреагирует на какое-то событие за обещанный промежуток времени, то произойдет катастрофа.

2. **Системы мягкого реального времени** - это системы, которые рассчитаны на то, чтобы почти наверняка выдерживать те временные характеристики, которые они обещают. Характеристики обычно более слабые и с большим разбросом, чем у систем жесткого реального времени, реакция на какое-то внешнее событие может обещаться в диапазоне от 2 до 5 мс. Если в этих системах оказывается, что временные соотношения не выполняются, как правило, ничего страшного не происходит: например, две системы, которые управляют космическими аппаратами, имеют подсистемы обработки баллистической информации, то есть они отслеживают орбиту космического аппарата и считают, когда необходимо включать двигатели, чтобы орбиту либо корректировать, либо поддерживать (все аппараты тормозятся и их орбиты становятся всё более низкими, пока он в конце концов не сгорит, с целью предотвращения этого, необходимо включать двигатели, чтобы орбиту повышать). Баллистические комплексы отслеживают на каждом витке параметры орбиты и выдают бортовой аппаратуре информацию о том, что ей необходимо сделать. Известно, что если при обработке баллистической информации система не успевает передать на борт уставки, то космический аппарат не упадет - он точно продержится следующий виток и не сгорит, можно спокойно дождаться, пока он окажется в зоне видимости и передать информацию. Другой класс систем мягкого реального времени, которая тоже используется при управлении космическими аппаратами, - это обработка телеметрии. С борта любого космического аппарата в зоне видимости идет сильный поток телеметрической информации: передается состояние аппаратуры на борту, состояние космонавтов и т.д. Телеметрическая информация дублируется, коэффициент полезного действия не больше 0,001, если система не успеет справиться, то при такой избыточности почти наверняка ничего не потеряется.

Исходя из того, что к системам жесткого и мягкого реального времени разные требования, для них используются разные подходы. Как таковых систем жесткого и мягкого реального времени не бывает, бывают некоторые framework, в которых делаются приложения жесткого реального времени, они собираются вместе со всеми компонентами, которые должны создавать операционную систему. Все это вместе отлаживается в жестком режиме, происходит жесткое моделирование того, как это

будет работать, чтобы точно удостовериться, что время реакции будет предписанным, и никогда другим. Что касается систем мягкого реального времени, то они делаются на универсальных операционных системах обычными способами, про это мы поговорим позднее.

Российские эксперты активно занимаются внедрением операционной системы реального времени **JetOS**, основной производитель Государственный научно-исследовательский институт авиационных систем, который занимается бортовым программным обеспечением. Эту систему собираются ставить на обычные самолеты, она, вероятно, будет использоваться в тех компонентах самолетов, которым не требуется жесткая реактивность. В управлении бомбометанием используется жесткое реальное время, потому что если программа не среагирует за меньшее время, чем микросекунды, то бомба упадет не туда, куда полагается. В системах жесткого реального времени без глубокого моделирования с просчетом времени и воссоздании его везде - обойтись невозможно, там все жестко детерминировано и не должно быть никаких синхронностей. Есть международный стандарт систем реального времени для гражданской авиации ARINC 653 (регламентирует временное и пространственное разделение ресурсов авиационной ЭВМ в соответствии с принципами интегрированной модульной авионики), в нем запрещается использовать thread, потому что если мы их допускаем, то теряем детерминированность поведения процессора, то есть он может вести себя по-разному в зависимости от каких-то временных характеристик. Возможность внутреннего распараллеливания допускается только в каких-то расширениях стандартов. Даже в мягком реальном времени имеются свои ограничения.

Понятие процесса и его состояния, операции над процессами и связанные с ними понятия

На первой лекции я говорил в следующих терминах: "вычислительная система выполняет одну или несколько программ", "операционная система планирует задания", "программы могут обмениваться данными" и т.д. Все это в житейском смысле звучит нормально, мы можем говорить про какие-то вещи одновременно используя слова, которые обозначают одновременно и объекты в статике, и объекты в динамическом состоянии, находящиеся в процессе исполнения. Для того, чтобы было можно более детально обсуждать вопросы функционирования современных компьютерных систем - необходимо уточнить эту терминологию.

Предположим, что имеется готовая к выполнению программа вычисления квадратного корня из натуральных чисел, и предположим, что одному пользователю требуется извлечь квадратный корень из 1, а другому - из 4. В результате у нас будут произведены два разных вычислительных процесса: будут иметься разные исходные данные, будет производиться разные последовательности вычислений, будут получены разные результаты, которые могут быть выведены на разные устройства ввода-вывода. Для того, чтобы в действительности описать, что происходит внутри компьютера - нельзя использовать термин **"программа"** в пользовательском смысле этого слова.

Аналогичная ситуация с термином **"задание"**, потому что задание - это с совокупность программы/набора команд языка управления заданиями, который требуется для её выполнения и входных данных. Если имеются разные входные данные, то мы будем иметь дело с двумя разными заданиями. Предположим, что есть идентичные задания, каждое направлено на то, чтобы извлечь квадратный корень из 1, но их загружают в систему со сдвигом во времени. Можно ли сказать, что эти задания в данный момент времени внутри вычислительной системы идентичны? Нет, потому что в конкретный момент времени состояние процесса их выполнения различно: то задание, которое было запущено первым - уже выдает на устройство печати результаты извлечения квадратного корня, а второе - только приступает к тому, чтобы начать его извлекать. Проблема в том, что термины "программа" и "задание" предназначаются для описания статических, то есть неактивных объектов. Программа, когда она выполняется, является динамическим, активным объектом. В компьютере выполняются различные команды, обрабатываются различные значения переменных. В операционной системе при разных исполнениях программ по-разному выделяется основная память, по-разному запрашиваются устройства ввода-вывода или файлы, поэтому вместо терминов "программа" и "задание" для активных объектов вычислительной системы используется термин "процесс". Для простоты обычно предлагается думать о процессе как об абстракции, характеризующей выполняющуюся программу. Это не совсем корректно. Необходимо говорить не о том, как определить процесс, а о том, что с этим понятием связано.

Процесс - это некоторая совокупность набора исполняющихся команд (может быть одна или несколько программ, которые выполняются последовательно), ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы, устройства ввода-вывода и т.д.) и текущего контекста его выполнения - того, что характеризует выполнение программы на процессоре (значения процессорных регистров, значение счетчика команд, то есть адреса очередной выполняемой команды, текущее состояние стека и значения переменных и т.д.), находящаяся под управлением операционной системы. Взаимно однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами - нет. В некоторых операционных системах для работы определенных программ может организовываться более одного процесса, и наоборот - один и тот же процесс может последовательно исполнять несколько различных программ. Даже в случае обработки только одной программы в рамках одного процесса - нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода её ядра, как в случаях, которые специально запланированы авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ими ситуациях (например, при обработке внешних прерываний).

Представим, что есть пассивное ядро, то есть некая совокупность программы данных/динамически скомпонованная библиотека, какой-то процесс пытается обратиться туда системным вызовом, у него есть поток управления, в котором управляются его команды. Возникает обращение к ядру и поток управления продолжается, просто в нем уже начинают выполняться команды из ядра, то есть условно можно считать, что мы как бы вызвали подпрограмму, поэтому это и называют системным вызовом, хотя в действительности это переход в режим ядра. В случае активного ядра, так просто не получается, потому что ядро само управляет тем, когда оно готово кого-нибудь обслужить, то есть можно считать, что в случае, когда микроядро дожидается получения сообщения от какого-то процесса - оно продолжает его выполнение, в этом случае поток управления ядра переключается в хвост потоку управления процессом, хотя это не очень очевидно.

Всё, что выполняется в вычислительной системе, даже определенные части операционных систем - организуется как набор процессов.

- На однопроцессорной компьютерной системе в каждый момент времени на процессоре реально может исполняться только один процесс, то есть команды только одного процесса. Есть много веток со своими потоками управления, которые все проходят через одно узкое горло, куда может поместиться только одна ветка. Все потоки пытаются пробиться к процессору, но пройти может только один. Если горлышек больше, то больше процессов может выполняться реально, но в любом случае, пока один процесс исполняется, остальные ждут своей очереди на получение процессора.
- В мультипрограммных вычислительных системах обеспечивается одновременная или псевдопараллельная обработка нескольких процессов, которая достигается с помощью переключения процессора с одного процесса на другой, когда программы, выполняемые в разных процессах на процессоре - чередуются: немного от одного процесса, немного от другого и т.д. Это действительно псевдо или квазипараллельное выполнение смеси процессов, потому что с точки зрения внешнего наблюдателя - оно параллельное, с точки реализации - оно называется interleaving/чередование, так как мы выполняем процессы на процессоре по кусочкам.

Диаграмма состояний процесса

Процесс, который находится в состоянии "процесс исполняется", может завершиться (тогда это выход) или быть приостановлен операционной системой и переведен в состояние "процесс не исполняется". Приостановка процесса может произойти по следующим причинам: для работы потребовалось возникновение события (завершение операции ввода-вывода) или наоборот - ничего внешнего не произошло, сама операционная система решает заменить процесс, которому дается процессор, потому что истек временной интервал, отведенный операционной системой в режиме чередования для работы этого процесса на процессоре. В этом случае

операционная система выбирает один из числа процессов, которые сейчас готовы для исполнения и находятся в состоянии "процесс не исполняется" - и переводит его в режим "процесс исполняется".

Рис. 2.3. Простейшая диаграмма состояний процесса

Новый процесс, появляющийся в системе, первоначально помещается в состояние "процесс не исполняется", пока не будет выбран операционной системой для выполнения. В диаграмме состояний процесса не учитывается, что в действительности процесс, который выбирается для исполнения, возможно, ещё сам не готов. Предположим, что мы его приостановили, потому что он ждет конца обмена с внешним устройством, и к тому моменту, когда подошло его время для исполнения - он может быть к нему не готов.

Уточненная диаграмма состояний процесса

Более точная диаграмма показывает, что стоит разбить состояние "процесс не исполняется" на два новых состояния: **готовность и ожидание**.

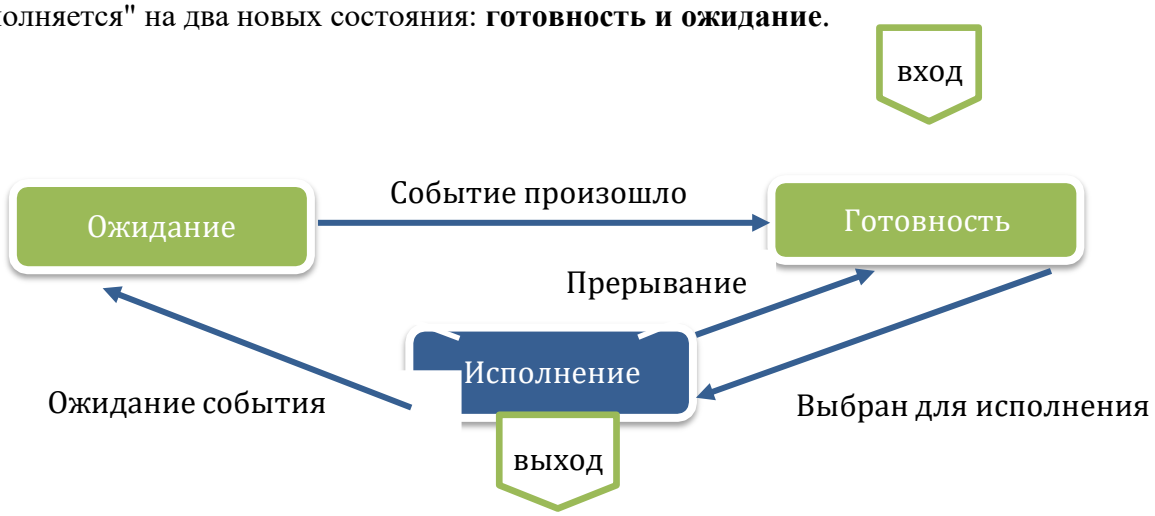

Рис. 2.4. Более подробная диаграмма состояний процесса

Всякий новый процесс - готов, он ничего не ждет, операционная система выбирает один из готовых процессов и переводит его в состояние "исполнение", в

котором происходит непосредственное выполнение программного кода процесса. Процесс выполняется на процессоре и у него три исхода:

1. процесс может закончить свою деятельность - тогда это выход;
2. процесс может по каким-то причинам прерваться, например, прерывание от таймера по истечении дозволенного времени выполнения, процесс возвратят в состояние "готовность";
3. процесс не может продолжать свою работу, пока не возникнет какое-то событие, и операционная система не переведет его в состояние "ожидание".

Состояния "ожидание" и "готовность" - разные, потому что из состояния "готовность" процессы можно выбирать на исполнение, а из состояния "ожидание" - нельзя, потому что они не готовы. Процесс переходит из состояния "ожидание" в состояние "готовность", когда происходит то событие, которое он ожидает.

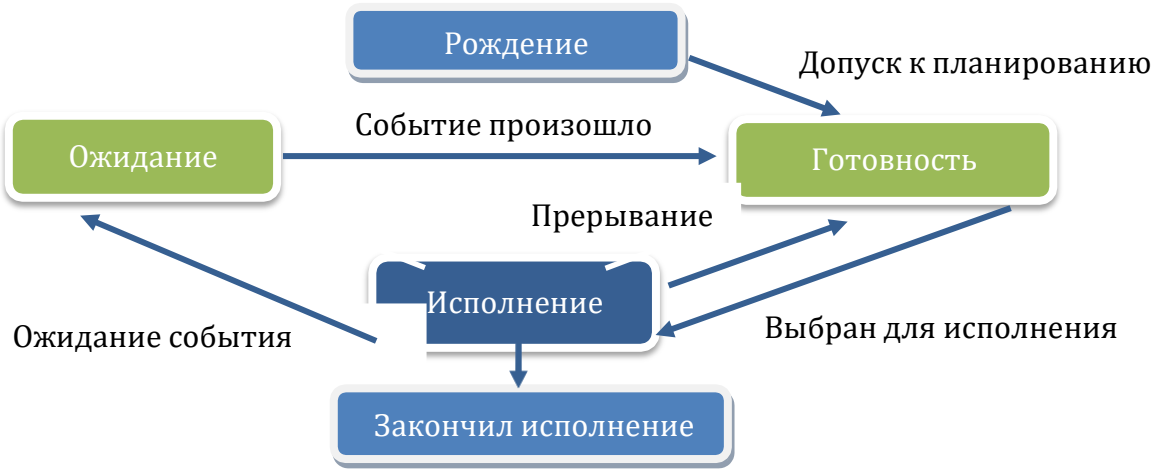

Рис. 2.5. Диаграмма состояний процесса, принятая в курсе

Для полноты картины нам необходимо ввести ещё два состояния процессов: **рождение и закончил исполнение**, потому что в действительности состояния "вход" и "выход" не являются атомарными, то есть процессу требуется нечто, чтобы стать готовым, и требуется нечто, чтобы престать существовать. В диаграмму добавляется состояние "рождение", из которого процесс попадает в состояние "готовность", добавляется состояние "закончил исполнение", после которого процесс исчезает.

Для того, чтобы появиться в вычислительной системе процесс должен пройти через состояние "рождение" - это существенная часть процесса, при котором он получает в своё распоряжение от операционной системы адресное пространство, в которое загружается программный код процесса, ему выделяется стек и системные ресурсы, устанавливается начальное значение счетчика команд или программного счетчика процесса (то есть откуда ему начать выполнение) и т.д. После всего, когда он

стал оформившимся, полноправным процессом, он переводится в состояние "готовность". В конце необходимо выполнить обратное действие - не так легко оформленный процесс сделать несуществующим, отобрать у него всё необходимое, вернуть все ресурсы в общие закрома операционной системы. Для этого предусмотрено довольно длительное состояние "закончил исполнение".

В конкретных операционных системах состояния процесса могут быть ещё более детализированы. Могут появляться некоторые новые варианты переходов из одного состояния в другое. В Unix было 9 состояний процессов, потому что было ещё состояние, когда процесс вроде ничего не ждет, вроде по своему внутреннему состоянию он готов, но у него нет памяти, которую отобрала операционная система. Это процесс, который сама операционная система сделала неполноценным. В принципе, все известные операционные системы в том или ином виде этой модели подчиняются.

Управление процессами внутри операционной системы. Блок управления процессом и контекст процесса. Изменением состояния процессов занимается операционная система - это число операций, которое совпадает с числом стрелок на диаграмме состояний, удобнее всего разделить их на три пары (в дальнейшем, когда мы будем говорить об алгоритмах планирования, в модели добавится ещё одна операция, не являющаяся парной: **изменение приоритета процесса**. Набор операций:

- Создание процесса - завершение процесса.
- Приостановка процесса (перевод из состояния "исполнение" в состояние "готовность") - запуск процесса (перевод из состояния "готовность" в состояние "исполнение").
- Блокирование процесса (перевод из состояния "исполнение" в состояние "ожидание") - разблокирование процесса (перевод из состояния " ожидание " в состояние "готовность"). Обратим внимание, что не назад в "исполнение", а в "готовность", потому что из готовых процессов выбирается процесс для исполнения.

Операции "создание процесса - завершение процесса" являются одноразовыми, то есть применяются к каждому процессу не более одного раза, потому что в большинстве операционных систем есть такие виды процессов, которые оформляются как бы до запуска самой операционной системы. Это предопределенные процессы, то есть им ресурсы даются на стадии генерации операционной системы - она запускается, а по всем параметрам эти процессы уже существуют. Остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются многоразовыми.

Рассмотрим подробнее, как операционная система выполняет операции над процессами. Каждый процесс представляется в операционной системе некоторой

структурой данных, которая называется **PCB (Process Control Block)** или блоком управления процессом:

- включает в себя минимум состояний управления процесса: процесс блокирован; процесс готов, то есть ожидает событие; процесс исполняется.
- запоминается программный счетчик процесса, который соответствует тому моменту, когда процесс снимается с процессора, то есть с какого момента его необходимо возобновить, когда он попадет на процессор в следующий раз.
- сохраняется содержимое регистров процесса, то есть то состояние регистра, которое существовало к моменту его снятия с процессора.
- данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, который определяет его права на занятие процессора, насколько он может остальных опередить или наоборот - отстать от них; размер и расположение адресного пространства и т.д.).
- учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т.д.), не все данные действительно необходимы операционной системе, идентификационный номер процесса ей необходим для того, чтобы внутри различать процессы по их идентификаторам, а пользователь ядру не нужен, но если вдруг это система, которая считает деньги (платная система), то ей это знать необходимо (кому выписывать счет).
- информация об устройствах ввода-вывода, связанных с процессом (таблица открытых файлов, таблица устройств, которые каким-то образом процессом захвачены и т.д.).

Конкретный состав и строение структуры PCB зависят от особенностей конкретной операционной системы. Во многих из операционных систем имеется несколько связанных структур данных, которые описывают один и тот же процесс Очень красиво это сделано в Unix, где с каждым процессом связаны две структуры данных: одна из которых - маленькая, в ней пишутся состояния процесса и то, что необходимо управлять процессором, выделять процессу время на процессоре; маленькие описатели процесса хранятся в фиксированной памяти ядра Unix, то есть в памяти, которая никаким образом никогда не удаляется - это физическая память процессора, физическая память операционной системы Второй описатель хранится в виртуальной памяти ядра операционной системы. Он сделан так, что информация, которая в нем находится, необходима только для тех процессов, которые в данный момент выполняются на процессоре, и вот тогда, используя свойства виртуальной памяти для всех процессов, для каждого большого описателя, они заходят с одного и того же виртуального адреса. Когда процесс запускается, сама операционная система меняет виртуальную память так, чтобы у вновь запущенного на процессоре процесса

описатель находился по тому же самому виртуальному адресу. Это очень красивая идея, она очень техническая, но очень полезная: операционной системе никогда не нужно искать описатель процесса, который сейчас работает на процессоре, так как она всегда знает, где он сейчас находится. Кроме того, большие описатели могут даже удаляться из основной памяти и вновь туда помещаться. Для простоты будем считать, что структура PCB одна. **Process Control Block** - это некоторая модель процесса для операционной системы. Любая операция, производимая операционной системой над процессом, вызывает изменения в PCB.

Содержимое всех регистров процессора, связанных с данным процессом, принято называть **регистровым контекстом процесса**. Можно сказать, что, когда процесс выполняется, у него все время меняется регистровый процесс: он выполняет команду за командой и регистр меняется, когда он уходит с процессора - контекст застывает, он определяет, что должно получиться регистрах, когда процесс в следующий раз будет запущен на процессоре. Счетчик команд, который показывает, какая сейчас выполняется последняя команда - входит в регистровый контекст, потому что это тоже регистр. Остальная часть PCB называется **системный контекст процесса,** он включает все, что ему выдала операционная система. Регистрового и системного контекстов процесса достаточно для выполнения операций над процессами, но они не полностью характеризуют процесс, потому что у него ещё есть нутро – то, что в процессе делается. Это определяется кодом и данными, которые находятся в адресном пространстве процесса, грубо говоря, в виртуальной памяти - это называется **пользовательским контекстом процесса**. Регистровый контекст - это чистая аппаратура, системный контекст - это то, что идет от операционной системы, пользовательский контекст - это то, что происходит от пользователя. Совокупность этих контекстов процесса для краткости принято называть просто контекстом процесса, то есть **контекст процесса** - это все, что его характеризует в данный момент времени.

Одноразовые операции. Жизненный путь процесса в компьютере начинается с его рождения. Любая операционная система, поддерживающая концепцию процессов, должна обладать средствами для их создания. В очень простых системах (в случаях специализированных систем) все **процессы могут быть порождены на этапе запуска системы**, например, даже не на этапе генерации. При запуске любой операционной системы, после её начальной раскрутки работает стартовый файл, который обычно называется startup cmd - это некоторый скрипт, который говорит, что необходимо сделать, прежде чем операционная система будет готова для использования пользователями. Но может так оказаться, что она никогда не будет готова для того, чтобы её использовали другие пользователи, и все процессы могут быть порождены на этапе первоначальной раскрутки, на этапе выполнения стартового файла (если эта система специализированная, то есть предназначена для выполнения специальных функций). В более сложных, в более универсальных операционных системах - **процессы создаются динамически, по мере необходимости**. Инициатором рождения нового процесса после старта операционной системы может выступать либо процесс

пользователя, совершивший специальный системный вызов, либо процесс самой системы. Процесс, инициировавший создание нового процесса, принято называть **процессом-родителем/parent process**, вновь созданный - **процессом-ребенком/child process**. Процессы-дети могут, в свою очередь, порождать новых детей и т.д. Набор генеалогических деревьев процессов - генеалогический лес.

На рисунке 2.6. изображен пример генеалогического леса процессов Unix. Все пользовательские процессы вместе с некоторыми процессами операционной системы принадлежат к одному и тому же дереву леса. Во многих операционных системах лес вырождается в одно дерево. Лес деревьев сводится к одному дереву пустым добавлением корня, если объединить все отдельные деревья одним корнем, то получится одно дерево, что иногда гораздо удобнее, чем придумывать лес.

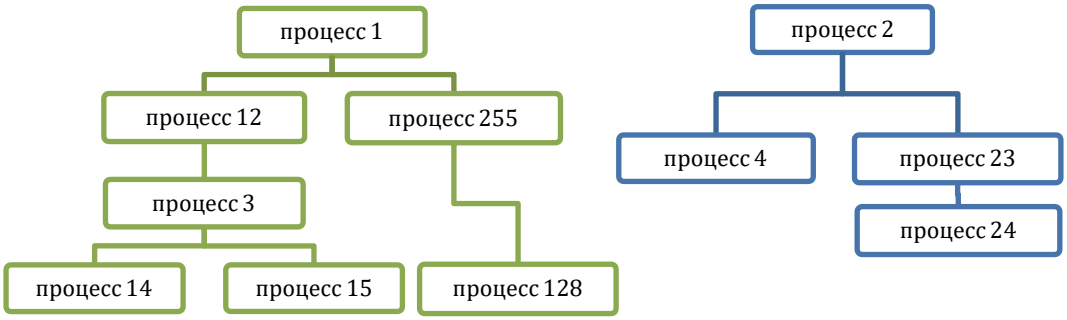

Рис. 2.6. Упрощенный генеалогический лес процессов

При создании процесса операционная система заводит новый PCB, выделяет память под новый блок управления процессом, записывает там состояние процесса "рождение" и начинает его заполнение. Новый процесс получает свой собственный уникальный идентификационный номер. Из-за того, что память ограничена, а номера должны быть уникальны, для соблюдения уникальности номеров число одновременно присутствующих в системе процессов должно быть ограничено. После завершения процесса его номер может быть повторно использован для другого процесса, то есть уникальность во времени здесь не требуется. Для выполнения своих функций процесс-ребенок требует ресурсов: памяти, файлов, устройств ввода-вывода и т.д. Есть два подхода к их выделению: можно получать в свое пользование некоторую часть родительских ресурсов, разделяя права на них с процессом-родителем и другими процессами-детьми - это подход Unix; можно получить ресурсы непосредственно от операционной системы, информация о выделенных ресурсах заносится в PCB, после выделения ресурсов требуется занести в адресное пространство процесса программный код, значения данных и установить программный счетчик. Возможны три решения:

1. Решение Unix: каждый порождаемый процесс-ребенок становится полной копией процесса-родителя (родитель раздваивается) по регистровому и пользовательскому контекстам (то же самое содержимое, то же адресное

пространство, тот же счетчик команд, содержимое регистров). Можно определить, кто является родителем: есть специальный системный вызов для создания нового процесса; ребенок, когда начинает работать, начинает с того, что получает возврат из системного вызова, который сделал его папа, код ответа на этот системный вызов у ребенка не такой, как у папы. После того, как они выполнили системный вызов fork, каждый анализирует ответ и узнает, кто он есть. Ребенок узнал, что он - ребенок и хочет выполнить другую программу, а не ту, которую запустил его папа, для этого есть специальный системный вызов exec, который позволяет заменить адресное пространство и загрузить туда ту программу, которая необходима.

2. Другой подход, возможно, более понятный: когда создается процесс, то операционная система сразу получает информацию о том, что в нем необходимо выполнять. Это разновидность таких языковых конструкций, которые когда-то были в параллельном Pascal, - parbegin, то есть с этого момента этот кусок кода выполняется в другом процессе. В одном процессе можно последовательно выполнить несколько различных программ. Процесс-ребенок сразу загружается новой программой из какого-нибудь файла.

3. Подход виртуального процессора. Если мы смотрим на то, что делается при создании процесса, то легко увидеть, что есть шаги, которые совершенно не зависят от того, какой будет выполняться код, какая будет выполняться программа. Эти шаги каждый раз повторяются, когда выполняется операция создания процесса. Возможно отдельно создать команду **"виртуальный процессор"**, то есть создается заготовка процесса, который пока ещё не может выполняться, но содержит все необходимое для того, чтобы при сообщении о выполнении он мог конкурировать за процессор. Системный вызов "создать виртуальный процессор" и системный вызов "запустить программу на виртуальном процессоре" - внешние команды, внешние системные вызовы. Это хорошо тем - что, когда завершиться программа, запущенная в виртуальном процессоре, он снова станет готов к тому, чтобы в нем можно было выполнять другую команду. То есть мы как бы выносим за скобки те накладные расходы, которые требуются для того, чтобы подготовить инфраструктуру для порождения процесса. Видно, что этот подход отличается и от подхода Unix, и от традиционного, в этой идее есть зародыш асинхронных кластеров, которые потом стали основой кластерной операционной системы.

После того, как процесс наделяется содержимым, в PCB дописывается оставшаяся информация, а состояние нового процесса изменяется на "готовность". Процесс-родитель может после этого одновременно продолжать свое выполнение, то есть чередуя команды с выполнением процесса-ребенка. Процесс-родитель может ожидать завершения работы некоторых или всех своих детей.

Лекция 3. Планирование процессов

Операции над процессами и связанные с ними понятия. Окончание

Одноразовые операции. Операция подготовки процесса к выполнению в основном заключается в том, что формируется блок описания процесса, заводится его виртуальная память и пр. После того, как сформирован весь контекст процесса (в некоторых системах иногда приходится формировать и регистровый контекст, потому что процесс начинает выполняться так, как будто он с процесса уже был снят, то есть ему необходимо иметь какое-то начальное значение регистров) - он может выполняться, то есть находится в состоянии готовности. Вторая одноразовая операция, которая является парной к операции подготовки, - это когда процесс завершает свою работу, то есть его контекст необходимо ликвидировать. Фактически это означает, что необходимо освободить все ресурсы, которые с ним были ассоциированы, не считая ресурсов, которые запрашивались динамически, потому что они могут остаться ещё привязанными к процессу (это файлы, устройства и т.д.), в первую очередь виртуальная память. Все это происходит достаточно долго. Именно поэтому мы в прошлый раз рассматривали идею о сохранении в системе процессов, которые ничего не делают, на предмет того, чтобы потом им давать новые программы на выполнение. Мы видим, что в действительности открывающие и закрывающие скобки достаточно дорогие. В основном это связано с тем, что приходится создавать новую виртуальную память, а потом её ликвидировать - это немаленькая работа. Есть близкая идея, которая касается thread: в некотором framework держится пул thread в которых можно в том же адресном пространстве выполнять программы, которые хочется в данный момент выполнять параллельно. Это похожая идея, но аналогия не полная, потому что thread - это легкий и дешевый процесс, у него нет виртуальной памяти, соответственно, нет и накладных расходов. Создать новый thread, вероятно, в 1 000 раз дешевле, чем создать тяжелый процесс, который обладает полным контекстом.

Для процессов, которые закончились, их описатели по техническим причинам продолжают оставаться в системе некоторое время. Это происходит исключительно из-за схемы Unix, потому что если процесс, который создан родительским процессом, аварийно или неожиданно для родительского завершается, то родитель необязательно об этом может узнать. В некоторых случаях он может запрашивать состояние своего потомка, даже если потомок уже умер. Необходимо, чтобы это можно было сказать, а для это нужно иметь информацию, то есть нельзя занимать описатель. Это одна из нескладных черт Unix и, видимо, неустранимая черта. Если процесс закончился, необходимо, чтобы тот процесс, который за ним следит, об этом сразу узнал, это нельзя откладывать на потом. В правильных системах (например, в VAX/VMS) есть иерархия и гибель предка приводит к гибели потомков, в этом смысл иерархии, иначе это не совсем иерархия. В Unix это не так: если родитель по каким-то причинам завершил свое выполнение, то его потомки продолжают существовать. В этом случае необходимо следить за тем, чтобы они оставались не безнадзорными, чтобы хотя бы

формально у них был родитель, как правило, они становятся потомками одного из дежурных системных процессов, что в целом нехорошо. Требуется изменение информации в PCB процессов-детей о породившем их процессе, чтобы генеалогический лес процессов оставался целостным. Одна из проблем операционной системы Unix заключается в том, что большинство её слабых мест (благодаря энтузиазму и пронырливости программистов) так или иначе успешно используется в приложениях. Даже если об этих слабых местах известно - их невозможно устранить. Вероятно, возможность продолжения жизни процессов, которые остались без родителя, реально используется в каких-то приложениях, потому что, строго говоря, это необходимо делать как полагается в иерархии.

Многоразовые операции: запуск процесса. Одноразовые операции всегда связаны с выделением или освобождением ресурсов, в этом их основной смысл. Многоразовые операции не приводят к изменению числа процессов в операционной системе и не обязаны быть связанными с выделением или освобождением ресурсов. Это зависит уже от реализации, в каких-то случаях это может делаться, но это не семантика этих операций. Для определенности мы говорим про одноядерную систему или однопроцессорную систему, потому что на многоядерную или многопроцессорную систему это легко обобщается, когда говорится "когда процессор стал свободным", то можно понимать "когда какой-нибудь процессор стал свободным из числа имеющихся". Кратко опишем действия, которые производит операционная система при выполнении многоразовых операций над процессами (более подробно мы будем об этом говорить, когда будем рассматривать критерии и алгоритмы планирования процессов):

- Если по каким-то причинам операционная система принимает решение сменить процесс на процессоре, то она выбирает из числа процессов, находящихся в состоянии "готовность", один процесс для исполнения. Этот выбор является одним из самых критических выборов в операционной системе, потому что, как правило, процессоров меньше, чем процессов, то есть это всегда критичный ресурс. Этот выбор определяет - какую политику проводит операционная система, обслуживая заказчиков.

- Для того, чтобы было можно запускать готовые процессы на процессоре, операционная система обеспечивает наличие в оперативной памяти данных, необходимых для дальнейшего выполнения процесса. Каким образом это происходит - зависит уже от алгоритмов управления виртуальной памятью, некоторые из них тесно связаны с алгоритмами управления процессами (про это более подробно при рассмотрении виртуальной памяти).

- Состояние процесса, который выбран для запуска на процессоре, изменяется на "исполнение". На основе содержимого блока управления процессом/PCB восстанавливаются значения регистров для данного процесса, так или иначе управление передается команде, на которую указывает счетчик команд

процессора. В действительности, в известных архитектурах запуск реального процесса на реальном процессоре выполняется путем выполнения команды из ядра операционной системы возврата с прерывания, до этого на регистр, который хранит счетчик команд процессора, устанавливается значение, которое заполнялось в блоке управления процессом. Это не goto, а операция возврата из процедуры, как бы возврата из системного вызова, которого в этом процессе может и не быть.

Многоразовые операции: приостановка процесса. Работа процесса на процессоре, когда он находится в состоянии "исполнение", может быть приостановлена только в результате какого-либо внешнего или внутреннего прерывания - исключительной ситуации. В этом случае процессор автоматически (может и не автоматически) сохраняет счетчик команд и, возможно, другие регистры в стеке исполняемого процесса и передает аппаратное управление по специальному адресу обработки данного прерывания. Фактически на слово с основной памяти, в котором находится вектор данного прерывания, там происходит установка уровня выполнения процессора, которая соответствует обработчику данного прерывания, а также команда перехода на реальный обработчик прерывания, фактически обращение к операции interrupt, если это внешнее прерывание соответствующего драйвера устройства. Если это внутреннее прерывание, то на соответствующий компонент операционной системы, который обрабатывает такие внутренние прерывания. До того, как прерывание реально обрабатывается, операционная система сохраняет динамическую часть системного и регистрового контекстов процесса в его PCB, переводит процесс в состояние "готовность" и приступает к обработке прерывания. Это самый простой способ, можно делать и не так, потому что неизвестно - какой процесс будет выбран следующим. Возможно сделать так, что в состояние "готовность" процесс будет переведен попозже, когда выясниться, что процесс должен освободить процессор, и будет выбран другой. Это уже оптимизация.

Многоразовые операции: блокирование процесса. Процесс блокируется, когда он не может продолжать свою работу, не дождавшись возникновения какого-либо события . Мы рассмотрим, что такое событие, когда будем говорить о средствах синхронизации процессов, когда они чего-то могут ждать. Пока достаточно абстрактно определим это как ситуацию, когда процесс обращается к операционной системе с помощью некоторого системного вызова, который может не давать возврата до тех пор, пока не будет выполнена какая-то внешняя операция. Операционная система обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет процесс в очередь процессов, дожидающихся освобождения устройства или возникновения события, и т.д.) В этом случае ядро сохраняет необходимую часть контекста процесса в его PCB и переводит процесс в состояние "ожидание". В Unix происходит именно так, в нем есть много системных вызовов, обращение к которым может блокировать процесс, которые могут перевести его в состояние "ожидание". Это плохо, потому что, когда в системе непредвиденно возникает некая тупиковая

ситуация, когда есть много процессов, все чего-то ждут и не могут работать - разобраться в том, что же происходит - чрезвычайно трудно. Если есть много источников блокировок, то понять по какой причине возникла такая ситуация крайне тяжело. Были попытки сделать так, чтобы источники блокировок были локализованы, то есть чтобы их было мало, чтобы было можно понять, что реально происходит в системе.

После возникновения в системе какого-либо события (например, завершение операции ввода-вывода или процесс дожидается поступления какого-то сообщения, которое он ждал, таких блокирующих вызовов довольно много), операционная система точно определяет, какое событие произошло. Система проверят, находился ли некоторый процесс в состоянии "ожидание" для данного события, если находился, то переводит его в состояние "готовность", то есть для него уже условие продолжения работы выполнено. Перед этим, в зависимости от того, какое это событие, могут выполняться какие-либо необходимые дополнительные действия, связанные с наступлением события. Например, если это операция записи на жесткий диск и у драйвера в очереди есть несколько запросов на запись блоков, то перед тем, как процесс будет переведен в состояние "готовность" - будет запущена следующая операция записи на диск, которая стоит в этой очереди. Эта операция будет детально рассмотрена, когда мы будем говорить про синхронизацию.

Переключение контекста

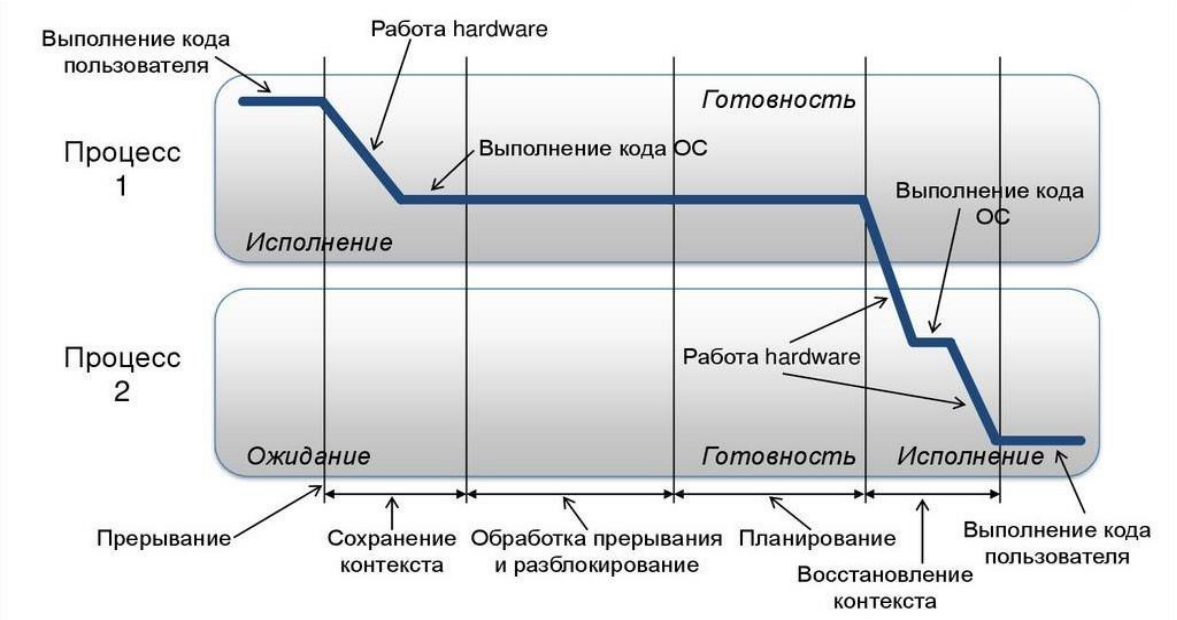

Рис. 3.1. Выполнение операции разблокирования процесса

Общая картина, как происходит переключение контекста с одного процесса на другой, показана на схеме: при исполнении процессором некоторого процесса 1

возникает прерывание от устройства ввода-вывода, которое означает, что возникло какое-то событие, после этого работает ядро операционной системы, которое это прерывание обрабатывает, если оно принимает решение, что необходимо переключиться на другой процесс, то восстанавливается контекст этого процесса, после чего некоторое время работает аппаратура для того, чтобы переключиться на новый процесс 2. В результате продолжает работать тот процесс, который система выбрала из числа тех процессов, которые находятся в состоянии "готовность".

Стоит поговорить о том, в чем тяжесть тяжелых процессов: то, что видно на поверхности - это то, что у тяжелого процесса своя виртуальная память, то есть когда происходит переключение с одного тяжелого процесса на другой тяжелый процесс, в действительности мы должны восстановить ту виртуальную память, на которой работает процесс, который мы хотим запустить. Любой процесс начинает эффективно (то есть быстро) работать на процессоре только после того, как он закачал в кэш (то есть в аппаратную часть сверхбыстродействующей памяти, которая поддерживается аппаратурой до доступа к основной памяти) достаточно данных и команд, чтобы было можно выполнять какую-то часть, по крайней мере до того, как реально потребуется следующее обращение к основной памяти. Тогда все происходит очень быстро, потому что кэши работают со скоростью процессора. Мы не говорим про уровни кэша, в целом они сделаны для того, чтобы вместо длинного обращения к основной памяти были короткие обращения к сверх быстродействующей основной памяти. Когда происходит переключение процессора с одного процесса на другой, мы должны сбросить кэш того процесса, который на процессоре выполнялся и запустить процесс (который не запускается по новой, а продолжает сове выполнение) на пустом кэше. Первая часть выполнения процесса на процессоре происходит крайне медленно, потому что все обращения к памяти идут в основную память до тех пор, пока снова не будет заполнен кэш, чтобы этого можно было избежать, по крайней мере с какой-то вероятностью. Основная тяжесть процесса заключается, вероятно, в том, что переключение процесса сбрасывает кэш. Мы не будем рассматривать сейчас архитектуры компьютеров без кэша и насколько хорошо на них переключаются процессы, потому что, во-первых, это существует в несколько идеализированной форме, в чистом виде таких архитектур ещё не существует. Достоинства кэша в том, что там хранятся копии кусков основной памяти, которые сейчас необходимы для выполнения процесса, достоинства одновременно являются недостатком архитектуры, потому что из-за этого мы очень много теряем при переключении процессора с одного процесса на другой.

Для корректного переключения процессора с одного процесса на другой необходимо сохранять контекст исполнявшегося процесса и восстановить контекст процесса, на который будет переключен процессор. **Процедура переключения контекста в ядре** тоже длинная - это может быть несколько сотен команд процессора, это накладные расходы на переключение, которые снижают производительность системы (от 1 до 1 000 микросекунд). Снизить накладные расходы на переключение позволяет расширенная модель процессов, включающая понятие **"нити исполнения -**

thread". Мое отношение к легковесным процессам, которые выполняются на общей виртуальной памяти, очень скептическое, но при этом необходимо отметить, что механизм фрэдов действительно позволяет достаточно дешево распараллеливать выполнение программ, если в этом есть смысл.

Заключение:

- Понятие **"процесс"** характеризует некоторую совокупность набора исполняющихся команд, собранных из одной или нескольких программ; ассоциированных с этим процессом ресурсов, которые могут быть привязаны к нему статически или выделены ему динамически по явным запросам; текущего момента его выполнения, а именно счетчика команд, который показывает на то, какая команда выполняется в данный момент времени процесса. Все вместе это находится под управлением операционной системы. В системах, в которых существует несколько процессов, которые в принципе могут конкурировать за процессоры, следует различать астрономическое время (которое показывает внешний таймер, если это старый таймер, в котором идут часы с внешней частотой в 50 гц, то сколько тиков прошло от одного момента до другого - это астрономическое время). Кроме того, у каждого процесса есть свое время, оно идет, когда процесс работает, когда он находится на процессоре. Оно измеряется тем же самым внешним таймером, если процессу удалось продержаться на процессоре, например, миллисекунду, значит к его личному времени жизни добавляется миллисекунда. Локальных таймеров, локальных показателей времени столько - сколько существует процессов в системе. Это очень важно, потому что операционной системе многие решения приходится принимать, исходя не из астрономического времени, которое проходит между какими-то двумя моментами, а из локального времени процессов, то есть из того, сколько прошло времени в жизни процесса от одной точки процесса до другой.

- Процесс полностью описывается своим **контекстом**, состоящим из регистровой, системной и пользовательской частей.

- В операционной системе все процессы представляются определенной структурой данных - **PCB (Process Control Block)**, отражающей содержание регистрового и системного контекстов. Блок управления процессом - важная структура, потому что, когда операционной системе приходится манипулировать процессом, она реально манипулирует описателями процесса. То есть она не может действовать на процесс так - как, например, на файл, потому что у файла кроме описателей есть ещё и внутренность, на которую она влиять по инициативе пользователя может. Что делает процесс - зависит от того, что хотел родительский процесс, который его запустил.

- Процессы могут находиться в 5 основных состояниях**: рождение, готовность, исполнение, ожидание, закончил исполнение.** Это разделение условно,

потому что в действительности мы увидим, что этих состояний может быть больше.

- Из состояния в состояние процесс переводится операционной системой в результате выполнения над ним **операций**. Это внутренние операции ядра, не считая, вероятно, операции "рождения", которая тоже является внутренней операцией ядра, но все-таки выполняется по инициативе какого-то пользовательского процесса.

- Между выполнением операций содержимое блока управления процессом (PCB) не изменяется.

- Операционная система может выполнять над процессом следующие операции: **создание процесса, завершение процесса, приостановка процесса** (снятие процесса с процессора операционной системой)**, запуск процесса (**изменение состояния из "готовность" на "исполнение")**, блокирование процесса** (когда, как правило, сам процесс решает, что необходимо дождаться некоего события, но бывают ситуации, когда система блокирует процесс, который к этому не стремился, когда он работает на процессоре или готов к работе, а система навязывает ему блокировку по техническим причинам)**, разблокирование процесса** (перевод процесса из состояния "блокирование" в состояние "готовность"), **изменение приоритета процесса** (операции, которые могут инициироваться пользователями, а могут выполняться по инициативе ядра, они приводят к тому, что этому процессу уделяется большее или меньшее внимание со стороны ядра, если он находится в состоянии "готов", насколько он имеет приоритет над другими процессами).

- Деятельность мультипрограммной операционной системы состоит из цепочек перечисленных операций, выполняемых над различными процессами, и сопровождается переключением контента.

- Время, затрачиваемое на переключение контекста, уменьшает полезное время работы процессора. Особенностью является подкачка кэша, пока процесс сможет работать эффективно.

Планирование процессов

Планирование процессов - важная тема, мы будем говорить про уровни планирования, про критерии планирования и требования к алгоритмам. Менее философский, но тоже не очень конкретный раздел - это параметры планирования, про две важные разновидности планирования: вытесняющее и невытесняющее, самый существенный раздел - это алгоритмы планирования.

При работе операционной системы, когда поддерживается мультипрограммный режим, мы имеем дело с тем, что есть некоторый ограниченный набор ресурсов,

прежде всего это процессоры и основная память, все остальное виртуализуется. Требуется обеспечить распределение ресурсов между потребителями, то есть планирование использования ресурсов В любом случае, планирование должно опираться на четко поставленные критерии и алгоритмы, соответствующие критериям и опирающиеся на параметры потребителей (какой разновидности ресурсов оно ни касалось бы). В этой лекции мы будем говорить о планировании исполнения процессов в мультипрограммных вычислительных системах или о планировании процессов.

Уровни планирования. В первой лекции мы рассмотрели два вида планирования в вычислительных системах: планирование заданий и планирование использования процессора. **Планирование заданий** - это процедура, в ходе которой принимается решение о том, какое очередное задание для загрузки в компьютер необходимо выбрать из пакета заданий, т.е. для какого задания породить соответствующий процесс. Такое планирование стало возможным после появления магнитных дисков. Пакетные системы появились раньше, чем магнитные диски, пакеты собирались на магнитных лентах, там планирование было тривиальным: следующим выбиралось следующее по порядку задание, поскольку магнитная лента чисто последовательное устройство. Магнитные диски обладают подвижными магнитными головками, то есть там время произвольного обмена более-менее приемлемое, чтобы было можно в онлайне выбирать блок прямого доступа. Когда они появились, то соответственно возникла возможность, перемещая магнитные головки, искать в пакете то задание, которое более рационально обрабатывать следующим.

Планирование использования процессора возникает в мультипрограммных вычислительных системах (правильнее говорить в мультипроцессных), где в состоянии "готовность" могут одновременно находиться несколько процессов. Процедуры выбора одного готового процесса, который получит процессор в свое распоряжение, - составляют суть этого планирования, т.е. необходимо выбрать процесс, который будет переведен в состояние "исполнение". Оба вида планирования: планирование задания и планирование процессов - будем рассматривать как разные уровни планирования процессов, потому что на первом уровне выбор задания приводит к образованию ещё одного процесса, на втором уровне - к тому, что выполняется на процессоре. Планирование заданий - это долгосрочное планирование процессов, потому что отвечает за порождение новых процессов в системе, фактически определяя степень мультипрограммирования, то есть число процессов, которые одновременно находятся в системе. Системы, с которыми мы сейчас работаем (что Windows, что Linux), фактически не обладают планированием заданий в обычном режиме использования компьютеров (если там нет каких-то фоновых процессов). Там неограниченная степень мультипрограммирования - пока хватит ресурсов. Если степень мультипрограммирования системы постоянна, то есть может быть "не более чем" процессов в системе, то новые процессы могут образовываться только после завершения ранее образованных процессов

Решение о выборе процесса для запуска на процессоре, то есть о выборе задания, для которого будет образован процесс, оказывает влияние на функционирование вычислительной системы на протяжении довольно длительного интервала времени. В некоторых операционных системах долгосрочное планирование сводится к минимуму, например, в системах разделения времени порождение процесса происходит сразу после появления соответствующего запроса. В настоящих системах разделения времени, которые исторически назывались time-sharing systems, максимальное число процессов строго определялось максимальным числом рабочих мест, которые подключаются к вычислительной системе. Сколько есть кассирш, которые продают авиационные билеты через одну систему - таково максимальное число процессов в этой системе. В таких системах поддержка разумной степени мультипрограммирования осуществляется за счет ограничения числа пользователей, то есть с одного рабочего места - один процесс.

Планирование использования процессора - это краткосрочное планирование процессов. Сама процедура планирования выполняется, когда процесс откладывается по собственной инициативе, и необходимо выбрать процесс из числа готовых, которые можно запустить на процессоре или при завершении некоторого интервала времени. Это планирование осуществляется весьма часто, как правило, не реже одного раза в 10 миллисекунд. Оно оказывает влияние на работу системы до наступления очередного аналогичного события, которое вызывает ядро операционной системы и ещё раз планирует процесс, то есть в течение короткого промежутка времени.

В некоторых вычислительных системах, к числу которых относятся те, которые основаны на классических вариантах операционной системы Unix (в какой-то мере сейчас это остается во всех вариантах Unix), бывает выгодно по каким-то причинам временно лишить выполняемый процесс памяти, то есть отобрать у него ресурс основной памяти, сохранив копию содержимого основной памяти на внешней памяти. Этот процесс со своей точки зрения может быть готов к выполнению, но у него отняли ресурсы, поэтому его выполнять нельзя. Его можно будет продолжить выполнять только потом, когда операционная система примет решение вернуть ему содержимое основной памяти в реальную основную. Эта процедура называется swapping, это именно "туда-сюда", то есть выкачка содержимого основной памяти во внешнюю память и наоборот - подкачка содержимого основной памяти из внешней. Поскольку такие процессы в системах являются живыми, то требуется дополнительный промежуточный уровень планирования процессов (кому дать возможность сидеть в памяти) - это **среднесрочное планирование процессов**.

Для каждого уровня планирования процессов можно предложить множество различных алгоритмов. Сложно сказать, сколько их всего было предложено к настоящему времени, не исключено, что кто-нибудь вроде Эндрю Таненбаума подсчитал это, потому что это чрезвычайно знающий человек, который всю свою жизнь занимается операционными системами. **Филипп Бернштейн** в своих книгах

совершено точно классифицирует алгоритмы управления транзакциями в системах управления базами данных. У него есть прекрасное утверждение, что все существующие алгоритмы управления транзакциями в той или иной мере относятся к одной из двух категорий - это довольно жесткая классификация. О существовании такой же классификации для алгоритмов планирования процессов мне неизвестно. То, что мы будем рассматривать в ходе лекций - это довольно произвольный набор алгоритмов, который дает какое-то представление о том, что в действительности используется, но только несколько из них (2-3) имеет отношение к реальной жизни, остальные более теоретические, они когда-то были хороши для статей, но на практике их использование зачастую невозможно. Выбор конкретного алгоритма определяется классом задач, которые решаются вычислительной системой, и целями, которые желательно достичь, используя планирование.

Операционная система - это "мать" всех программ, которые выполняются на компьютере, потому что все они выполняются под её надзором и управлением. Это, по-видимому, самая знающая из всех программ, которые работают на компьютере. Операционная система должна знать всё обо всех, так как она принимает решения: она должна знать, какого рода процессы выполняются под её управлением, как они ведут себя с памятью, как они требуют процессор, как часто они откладываются и т.д. И при этом знать она должна всё, а узнать неоткуда, потому что узнать необходимое система может только наблюдая за тем, что происходит. Человек, который принимает решения, опираясь на свои наблюдения, всегда принимает решения, которые не годятся для будущего, они годятся для прошлого. Операционная система принимает решения, касающиеся будующего - опираясь на прошлое, у нее нет другого источника знаний. Когда мы говорим, что выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, то возникает вопрос: откуда она знает про этот класс задач? Она будет знать про них, если ей кто-нибудь скажет об этом, но она должна самостоятельно убедиться в том, что она есть.

Если аналогично рассматривать системы управления базами данных, то они существенно менее общие программы, потому что работают для определенных целей и обслуживают конкретные приложения, и т.д. Они менее мудрые, чем операционная система, зато они гораздо умнее, потому что они в действительности знают такие вещи, которые неоткуда знать операционной системе. Это следствие того, что в СУБД выполняется только тот программный код, который в ней содержится, она его знает и знает, что она выполняет, что от этого кода требуется, у нее критерии, которые не меняются. В принципе, системе управления базами данных приходится решать задачи, которые очень похожи на те, которые приходится решать ядру операционной системы, но только она не может опираться в этом на операционную систему.

Цели могут быть абсолютно разными, все они абсолютно понятные, но противоречивые:

- **Справедливость:** гарантировать каждому заданию или каждому процессу некоторую часть времени использования процессора в компьютерной системе, не допуская таких ситуаций, когда процесс одного пользователя постоянно занимает процессор, а процесс другого пользователя фактически не выполняется. Процессор необходимо захватить и поделить поровну, такой справедливый дележ зачастую приводит к тому, что ничего не получается вообще ни у кого, потому что при дележе поровну у каждого может оказаться настолько мало, что ни у кого не хватит ни на что.

- **Эффективность:** более практическая цель, которая состоит в том, чтобы постараться так планировать процессы, чтобы процессор всегда был занят полезной работой на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению. Это хороший показатель, особенно он был востребован, когда компьютеры были дорогими, когда их было 3 шт. на Москву, каждый обслуживал по 200-300 человек, а их стоимость была равно стоимости самолета. Известно, что сейчас все современные центры данных сильно недозагружены. Центры данных - это аналог того, что когда-то называлось вычислительные центры, это специально подобранные вычислительные ресурсы, которые собраны для того, чтобы на них было можно выполнять различные полезные программы, которых не хватает, чтобы их загрузить. Одним из примеров являются наши замечательные суперкомпьютеры, которые находятся в ведении факультета ВМК МГУ - Ломоносов и Чебышев, которые обладают десятками тысяч узлов, существуют единичные задачи, которые занимают все узлы на этих компьютерах. Загрузка персонального компьютера составляет лучшем случае 3%, процессор практически всегда стоит, потому что ему нечего делать, возможно, это значение составляет доли даже процентов.

- **Сокращение полного времени выполнения (turnaround time):** если это пакетная обработка, то необходимо обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки - и его завершением. Если это пакетный режим, то это очень полезно.

- **Сокращение времени ожидания (waiting time):** минимизация времени, которое проводят процессы в состоянии **"готовность"** (по мере возможности процессы необходимо давать быстрее) или задания в очереди для загрузки.

- **Сокращение времени отклика (response time):** минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя. Это очень полезная цель в системах разделения времени и вообще в интерактивных системах. Это константа, которая осталась неизменной, она была получена ещё в 60-х голах прошлого века: если система обслуживает запросы пользователя, то допустимое время ожидания не должно превышать 4

сек. Этот отрезок времени человек ожидает сравнительно спокойно. Системы должны работать так, чтобы давать отклик за 4 сек. или быстрее.

Желательные свойства:

1. **Предсказуемость:** планирование должно осуществлять так (например, планирование заданий), чтобы одно и то же задание выполнялось примерно за одно и то же время.
2. **Минимальные накладные расходы:** минимизация накладных расходов всегда чрезвычайно полезна: если мы хотим минимизировать стоимость владения, то необходимо, чтобы для процесса при обработке задания минимизировались накладные расходы, чтобы полезная отдача была больше.
3. **Равномерная загрузка ресурсов:** загрузка вычислительной системы происходит с предпочтением процессов, которые будут занимать малоиспользуемые ресурсы. Предпочтительно, чтобы вся аппаратура стояла примерно равномерно, чтобы при необходимости апгрейда это можно было сделать для всей аппаратуры сразу.
4. **Масштабируемость:** свойство, которое очень важно для всех алгоритмов, они должны сохранять работоспособность, если нагрузка на них увеличивается. В этом смысле масштабируемость понимается именно так.

Многие приведенные цели и свойства являются противоречивыми, улучшение работы алгоритма с точки зрения одного критерия - ухудшает ее с точки зрения другого критерия: если мы будем пытаться улучшать работу алгоритмов с точки зрения, например, справедливости, то мы почти наверняка будем её ухудшать с точки зрения эффективности, и т.д. Приспособление алгоритма под один класс задач - приводит к дискриминации задач другого класса, который он будет крайне плохо обслуживать. Пример из области СУБД: можно оптимизировать систему управления базами данных, чтобы она эффективно поддерживала транзакции. **Транзакции** - это короткие последовательности операций, у которых примерно треть составляют операции изменения базы данных, которые, как правило, очень простые: например, выбрать товар по уникальному идентификатору, заказать некоторое количество этого товара и пр. Известно, как можно оптимизировать СУБД в расчете на то, чтобы она обеспечивала максимальную пропускную способность транзакции. Современные технологии позволяют на специализированном оборудовании обеспечивать пропускную способность миллиона транзакций в секунду, если вдруг в эту рабочую нагрузку вклеится какая-нибудь последовательность аналитических операций (что сейчас крайне модно, пустить аналитику прямо на сырых транзакционных данных для того, чтобы посмотреть некие аналитические показатели), то это может напрочь порушить всю транзакционную производительность, потому что аналитические запросы в базах данных выполняются совсем не так, для них необходим другой подход. Один аналитический запрос может в 100 0000 тыс. раз затормозить выполнение

транзакций. Аналогично: могут быть такие задачи, для которых оптимизированы алгоритмы планирования процессов, если вклинятся другие задачи, то они либо вообще не будут обслуживаться, либо будут обслуживаться крайне плохо.

Параметры планирования. Чтобы было можно достичь целей, которые ставятся перед алгоритмами планирования, они должны опираться на характеристики процессов в системе, характеристики заданий в очереди на загрузку, на состояния самой вычислительной системы, в целом все это называется параметрами планирования. Рассмотрим некоторые такие параметры:

- **Статические параметры** не изменяются в ходе функционирования вычислительной системы, динамические подвержены постоянным изменениям. Статические параметры вычислительной системы могут, например, включать предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, число подключенных устройств ввода-вывода и т.п.). Статические параметры - это параметры, которые никак не изменятся до тех пор, пока не будет произведен какой-то апгрейд вычислительной системы, а тем самым операционная система будет перезапущена.

- **Динамические параметры** системы описывают количество свободных ресурсов в текущий момент времени. Примером статических параметров являются: от имени какого пользователя запущен процесс или сформировано задание. Возможно, не очень хорошо, что задание так устроено, но так устроена жизнь, как говорил Джордж Оруэлл: "Все животные равны. Но некоторые животные более равны, чем другие". Это, безусловно, так - некоторых "равных" система должна обслуживать более качественно, чем других.

- Насколько важной является поставленная задача, каков её приоритет, откуда он берется? - это не дело алгоритмов планирования, строго говоря. Факт заключается в том, что необходимо учитывать, что у заданий и у процессов могут быть приоритеты, а задания с более высоким приоритетом должны выполняться вперед заданий с более низким.

- Сколько процессорного времени запрошено пользователем для решения задачи? Сейчас это воспринимается смешно, но с этим раньше приходилось жить, когда каждый раз было необходимо указывать в задании - сколько времени на задачу необходимо, на сколько времени операционная система будет рассчитывать. При написании, например, рекурсивных программ - оценить количество необходимого времени крайне сложно.

- Каково соотношение процессорного времени? Сколько проходит процессного времени между операциями, которые приводят к его откладыванию, например, для осуществления операций ввода-вывода?

- Какие ресурсы вычислительной системы и в каком объеме потребуются заданию? Это тоже требовалось указывать в пакетных системах.

В алгоритмах **долгосрочного планирования** используются статические и динамические параметры вычислительной системы: сколько у нее всего ресурсов и сколько сейчас свободных. Динамические параметры процессов на этапе загрузки заданий еще не известны. В алгоритмах краткосрочного и среднесрочного планирования дополнительно учитываются и динамические характеристики процессов

Для **среднесрочного планирования**, когда присутствует свопинг, то есть необходимо планировать, какие процессы сейчас находятся во внешней памяти, то есть образ процесса, а какие в основной. Для такого планирования в качестве **динамических характеристик** может выступать следующая информация:

- Сколько времени прошло со времени выгрузки процесса на внешнюю память и его загрузки в оперативную память? На сколько времени он пострадал, время астрономическое.
- Сколько оперативной памяти занимает процесс, сколько памяти ему необходимо реально, чтобы более-менее сносно себя чувствовать? Это оценивает операционная система.
- Сколько процессорного времени было уже предоставлено процессу? Сколько он прожил в своем локальном времени.

Для **краткосрочного планирования** требуется ввести еще два динамических параметра, которые операционная система не знает, она их оценивает:

- размер промежутка времени непрерывного использования процессора называется название;
- промежуток ввода-вывода – I/O burst, не обязательно только ввода-вывода, потому что это время блокирования процесса в действительности. Важными динамическими параметрами процесса являются значения последних и очередных CPU burst и I/O burst.

На рисунке пример того, как процесс выполняет присваивание переменным a и b и задает обмен с внешним устройством c, до этих пор - это был первый CPU Burst (столько времени он мог провести на процессоре), следующий промежуток I/O Burst - это время окончания ожидания события окончания ввода, потом идет очередной CPU Burst - это вычисление a на основе предыдущих значений a, b и введенного c, затем опять следует операция ввода-вывода print a, поскольку последняя операция print a - идет время ожидания операции её ввода-вывода.

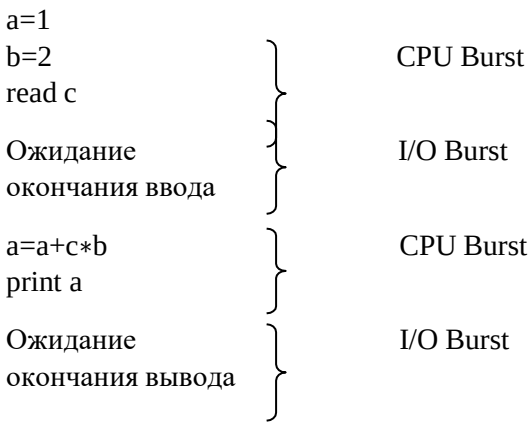

Рис. 3.2. Фрагмент деятельности процесса с выделением промежутков непрерывного использования процессора и ожидания ввода-вывода

Вытесняющее и невытесняющее планирование. Процесс планирования осуществляется частью операционной системы, которая называется планировщиком от английского scheduling/планирование. Планировщик может принимать решения о выборе для исполнения нового процесса в четырех разных случаях:

1. Когда какой-то процесс завершается, то есть он исчезает, переходит из состояния "исполнение" в состояние "завершение".
2. Когда процесс переводится из состояния "исполнение" в состояние "ожидание", то есть он перестает быть готовым.
3. Процесс переводится из состояния "исполнение" в состояние "готовность", то есть добавляется новый готовый процесс.
4. Когда процесс активизируется, то есть переводится из состояния "ожидание" в состояние "готовность", ещё один готовый процесс.

В случаях 1 и 2 процесс, который находился в состоянии "исполнение", не может дальше исполняться (либо закончился, либо отложился), и для выполнения всегда необходимо выбрать новый готовый процесс. В случаях 3 и 4 можно планирование выполнять, то есть пытаться выбрать какой-то другой процесс. Но планирование может и не проводиться, то есть процесс, исполнявшийся до прерывания, может продолжать выполнение после обработки прерывания. Если планирование в действительности обеспечивается только в случаях 1 и 2, когда процесс больше работать не может, то имеет место невытесняющее планирование/non-preemptive scheduling. Когда и в 3, и в 4, то это - вытесняющее планирование/preemptive scheduling.

Если какой-то процесс откладывается, то это означает, что он обратился к ядру - это было внутреннее прерывание при переходе на системный вызов, то есть работает ядро, оно узнает, что на самом деле процесс запросил синхронную операцию ввода-вывода, оно понимает, что этот процесс необходимо отложить. Тогда эта программа

сама обращается к планировщику процесса внутри ядра. Случай, когда наоборот - появляется новый процесс: возникает внешнее прерывание, например, драйвер определяет, что на устройстве закончилась операция ввода-вывода, система/ядро узнает, что завершение этой операции ждет какой-то процесс, переводит его в состояние "готовность" - и вот эта программа обращается к планировщику, чтобы он выбрал новый процесс на процессоре. То есть это всегда делается путем вызовов фактически функций внутри ядра операционной системы, планировщик никогда не работает из-за того, что его это сделать явно попросил пользователь. Пользователь просит чего-то другого, а система решает, что в результате этого необходимо перепланировать - кто сейчас будет работать на процессоре.

Бывают две архитектуры ядра операционной системы:

- **пассивное ядро** - это просто собранная библиотека, где много программ, которые находятся в общей виртуальной памяти (память у них отдельная), а работают они только в результате явного или неявного обращения пользовательского процесса, именно так в Unix. Если есть системный вызов, то аппаратно это происходит следующим образом: возникает прерывание, начинает работать ядро, как бы продолжая тот процесс, в котором возникло прерывание по поводу системного вызова. Ядро работает до тех пор, пока этот системный вызов не будет до конца обработан. Поскольку это продолжение того же самого процесса, который выполнил системный вызов, понятно, что бывают такие случаи, когда он может отложиться уже будучи в ядре, то есть ядро принимает решение, что сейчас оно его дальше выполнять не будет, он будет чего-то ждать. Из-за этого в действительности могут получиться такие ситуации, что в внутри ядра одновременно выполняются "наследники" нескольких пользовательских процессов, но все они работают в одной виртуальной памяти, так как у ядра своя память. В результате получается, что, начиная с какой-то версии Unix, такое поведение, когда внутри ядра может выполняться несколько системных вызовов от разных процессов, привело к тому, что в ядре Unix стало возможным выполнение несколько thread ещё до того, как это понятие появилось на пользовательском уровне. То есть тяжелые пользовательские процессы, если они выполняют системные вызовы и входят в ядро - становятся threads пассивного ядра операционной системы.
- **активное ядро** - это другая картина, все микроядерные системы обладали активным микроядром, которое работает в своей виртуальной памяти. В отдельном процессе оно работает независимо от пользовательских процессов, потому что оно поддерживало механизм их общения и обрабатывало операции, как правило, send-receive/посылки и приема сообщений. Чтобы было можно выполнять эти операции - необходимо обладать собственной активностью, по-другому это никак не получается. В системах с активными ядрами в ядре работает один тяжелый процесс, ядро само у себя может запускать threads, если

ему это требуется, если требуется какая-то параллельность при работе. Из действующих примеров можно привести только **QNX**, потому что он на вид Unix, но с другой архитектурой. Аналогичным образом был устроен французский **Chorus**, в котором не было процессов как таковых, а были кластеры.

Предположим, что ядро работает в таком режиме, когда в нем одновременно может обрабатываться несколько системных вызовов, считается, что внутри ядра есть несколько конкурентов на то, чтобы использовать процессор. Тогда любой из них, когда он выясняет, что он дальше работать не может (просто откладывается какой-то ядерный thread) - сам обращается к планировщику, который сам принимает решение, кому дальше продолжать работу на процессоре, какому хвостику, потому что у каждого thread есть свой небольшой контекст (об этом мы будем говорит позднее). Это сложно представить, потому что то, что мы сейчас рассматриваем - это некое абстрактное представление, проще всего это понимать на уровне аппаратуры, как это происходит реально и что делается. Может получиться так, что какой-то процесс влез в ядро и сам говорит операционной системе, что больше работать не может. Что должна сделать операционная система в этом случае? - она должна отдать процесс какому-то другому пользовательскому процессу, оставляя один процесс внутри ядра. Другой пользовательский процесс начинает работать, ведь его запустили на процессоре, он тоже выполняет системный вызов, тоже входит в ядро, тоже затыкается в ядре, говоря планировщику, что дальше работать не может. В результате в ядре может накопиться много таких недоделанных системных вызовов. Рано или поздно совершается какое-то внешнее событие (прерывание какого-то пользовательского процесса), соответственно, происходит вход в ядро от имени этого пользовательского процесса, эта ядерная часть пользовательского процесса может сказать какой-то функции ядра, что сейчас надо активизировать одну из этих отложенных веточек. Вот тогда это две ветки, два thread внутри ядра, которые одновременно становятся готовыми, их тоже необходимо планировать. Разговоры о том, чем системы с пассивным ядром отличаются от систем с активным ядром - абстракции, потому что аппаратура одна и та же, когда рассматриваешь, что происходит именно на аппаратном уровне, то оказывается, что это практически одно и тоже.

Невытесняющее планирование использовалось, например, в MS Windows 3.1 и ОС Apple Macintosh. Оно хорошо тем, что переключение контекста возникает только тогда, когда это реально требуется, то есть процесс занимает столько процессорного времени, сколько ему необходимо, пока он может работать. Переключение процессов возникает только при желании исполняющегося процесса. Если мы говорим о сокращении накладных расходов, например, то он грузит кэш по полной, то есть он работает постоянно и поддерживает в кэше то, что необходимо для работы. Поэтому метод планирования относительно просто реализуем и достаточно эффективен (редкая смена контекста). Есть проблема возможности полного захвата процессора одним процессом (программа может зациклиться), что несправедливо, хотя от каких

критериев отталкиваться, с точки зрения эффективности - это очень хорошо, потому что, например, в старых пакетных системах была задача максимально эффективно использовать вычислительную систему, для них такая работа была крайне подходящей. С другой стороны, поскольку фактически за процессом, который находится на процессоре, операционная система не следит (он попал на процессор и работает, пока сам не отложится, операционная система его не трогает) и не очень хорошо, когда эти процессы ведут себя плохо, например, процесс зациклился, а ядро радо, что он так хорошо держит процессор, а в действительности процесс ничего не делает. Спасает только перезагрузка вычислительной системы.

Вытесняющее планирование необходимо там, где требуется хотя бы какая-нибудь справедливость. Обычно она используется в системах разделения времени в этом случае процесс в любой момент своего выполнения может быть приостановлен, переведен из состояния "исполнение" в состояние "готовность". Это делается за счет квантования времени, то есть каждому процессу дается время на процессоре, которое не больше некоторого, установленного в системе кванта. Процесс запускается на процессоре, операционная система устанавливает специальный таймер, считает сколько локального времени процесс проводит на процессоре, когда время кончается - его с процессора снимают и выбирают "планировать другой процесс". Квантование гарантирует приемлемое время отклика, то есть это некоторая справедливость, а также предотвращают нехорошие ситуации, которые связаны с плохим поведением программы (зависание из-за зацикливания какой-либо программы).

Алгоритмы планирования. Гарантированное и приоритетное планирование

Существует большой набор алгоритмов планирования, которые предназначены для достижения различных целей и эффективны для разных классов задач. Некоторые из них могут быть использованы на нескольких уровнях планирования. Рассмотрим некоторый довольно случайный набор алгоритмов, среди которых есть наиболее употребительные. Все алгоритмы будем рассматривать применительно к процессу кратковременного планирования.

First-Come, First-Served (FCFS). Самый простой алгоритм - First Come, First Served/первым пришел, первым обслужен. В этом случае все процессы, находящиеся в состоянии "готовность", организованы в виде очереди, которая обычно представляет собой направленный список, в котором поддерживается два конца - начало и конец очереди. Когда процесс переходит в состояние "готовность", то помещается в очередь, он ставится в хвост, то есть ссылка на его PCB помещается в конец очереди. Новый процесс для исполнения планировщик выбирает из начала очереди с удалением оттуда ссылки на его PCB. **Очередь FIFO - First In, First Out**. Для выполнения выбирается процесс, который дольше всего простоял в очереди. Классический алгоритм FIFO работает в невытесняющем режиме, то есть в этом случае процесс, получивший в свое распоряжение процессор, занимает его до конца своего текущего CPU burst - это время, которое он сам хочет провести на процессоре (не откладывается). После этого для

выполнения выбирается новый процесс из начала очереди. Преимуществом алгоритма FCFS является простота реализации: очередь однонаправленная, никаких обратных ссылок, никогда не происходит удаление из середины очереди, вставки в середину очереди, процесс ставится всегда в хвост, выбирается всегда из начала. Но у него имеется и много недостатков, как у все алгоритмов категории FIFO (когда мы дойдем до управления виртуальной памятью, то увидим, как странно себя ведет этот алгоритм по отношению именно к замещению страниц). Рассмотрим пример:

Процесс	P_0	P_1	P_2
Продолжительность очередного CPU burst	13	4	1

Пусть в состоянии "готовность" находятся три процесса P_0, P_1 и P_2 , для которых известны времена их очередных CPU burst. Эти времена приведены в таблице в некоторых условных единицах. Нулевому потребуется 13 единиц времени на процессоре, первому - 4, третьему - 1. Предположим, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не производят операций ввода-вывода, и что время переключения контекста пренебрежимо мало. Это предположение довольно суровое.

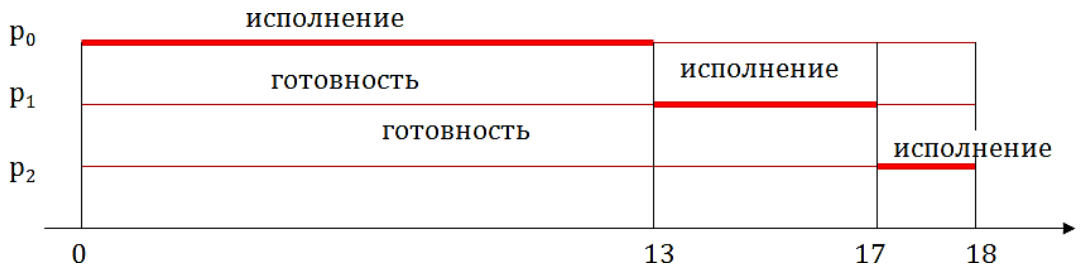

Рис. 3.3. Выполнение процессов при порядке P_0, P_1, P_2

Предположим следующий сценарий: первым выбирается процесс P_0, он получает процессор на все время своего CPU burst (13 единиц времени), процесс заканчивается, после него в состояние "исполнение" переводится процесс P_1, так как в таком порядке они стоят в очереди, он занимает процессор на свои 4 единицы времени и тоже кончается. Наконец, возможность работать получает процесс P_2, который вырабатывает свою единицу. Итого: нулевой процесс не ждал ни сколько, его время ожидания - 0, первый прождал 13 единиц времени, второй - 17, то есть среднее время ожидания в этом случае составляет - (0+13+17)/3 = 10 единиц времени. Среднее время выполнения (то есть от начала до конца) у нулевого процесса - это 13 единиц, он сразу попал на процессор и сразу 13 единиц доработал, у второго выполнения 17 единиц, у третьего - 18 единиц, он закончился в это время. Среднее время ожидания - (13+17+18)/3 = 16 единиц времени.

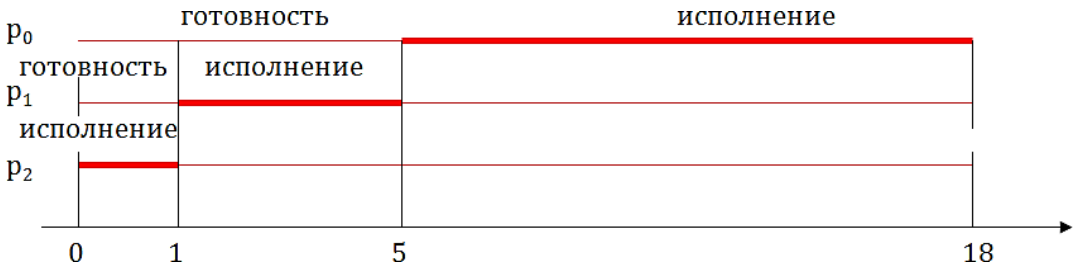

Рис. 3.4. Выполнение процессов при порядке и P_2, P_1, P_0

Если мы переставляем их порядок, то есть сначала запускаем второй процесс, потом первый, потом нулевой, тогда у P_2 время ожидания равно 0, он работает сразу, P_1 ждет 1 единицу времени, P_0 ждет 5 единиц. Среднее время ожидания в этом случае (5+1+0)/3 = 2 единицы времени. Полное время выполнения для нулевого процесса - 18 единиц, для первого процесса - 5, для второго - 1. Среднее полное время выполнения (18+5+1)/3 = 6 единиц времени. Мы видим, что среднее время ожидания и среднее полное время выполнения для алгоритмов First Come, First Serve существенно зависят от порядка расположения процессов в очереди. Если есть процессы с большим CPU burst, с большим запросом на время на процессоре, и мы их запускаем в начале, то процессы с короткими CPU burst, перешедшие в состояние готовность после длительного процесса, будут очень долго ждать начала своего выполнения. Такой алгоритм FCFS практически неприменим для систем разделения времени в невытесняющем режиме, хотя он очень хорош для пакетных задач и пакетного режима, потому что загрузка в таком режиме предельно эффективная.

Вариант алгоритма FIFO, реализованный в режиме вытесняющего планирования, называется **Round Robin** (RR).

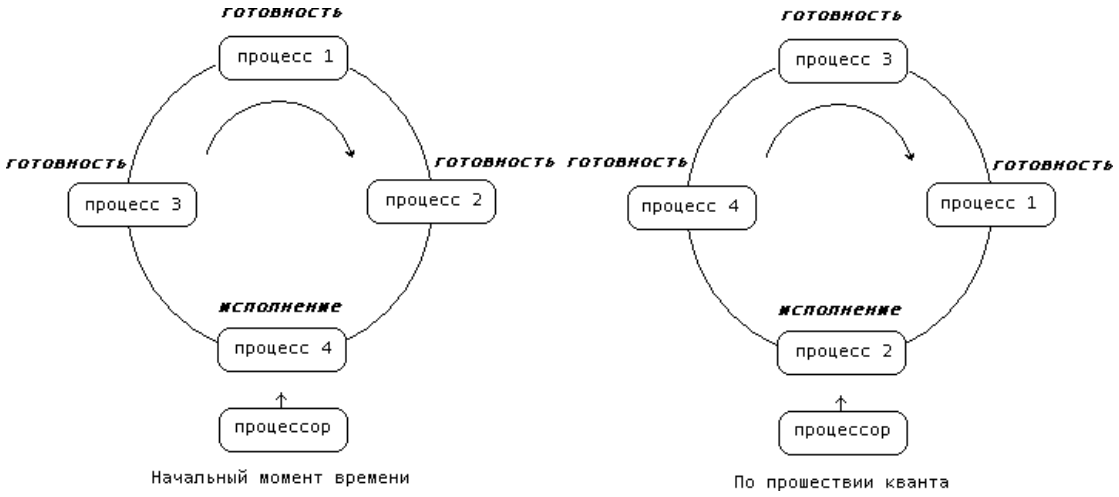

Рис. 3.5. Процессы на карусели

В этом случае все готовые процессы организованы в циклическую очередь - сидят на карусели, начало завязано на конец. Карусель крутится так, что каждый процесс находится на процессоре небольшой фиксированный квант времени, раньше это время составляло 10-100 миллисекунд, что много, на современных процессорах в режиме квантования достаточно 1 миллисекунды, так как необходимо предотвратить накладные расходы, потому что если мы даем поработать немного времени поработать, а потом выбираем другой процесс, то приходится часто восстанавливать контекст. Смысл алгоритма именно такой: поработал четвертый процесс, на его место попадает второй процесс, который работает свой квант времени, он получает процессор в свое распоряжение и может исполняться, и т.д. Round Robin реализуется с помощью организации процессов, находящихся в состоянии готовность, в очередь FIFO. Для очередного исполнения выбирается процесс, расположенный в начале очереди, устанавливается таймер, для того чтобы система была оповещена о том, что он проработал свой квант времени.

При выполнении процесса возможны два варианта:

1. Если время непрерывного использования процессора (остаток текущего CPU burst, то есть он может провести на процессоре какое-то время, и оно у него ещё останется – столько, сколько ему необходимо для откладывания), требующееся процессу, **меньше или равно кванту времени**, который установлен в системе, то процесс откладывается, он сам говорит, что больше работать не может. Тогда он становится неготовым, блокированным процессом. Меньше и равно кванту времени означает: равно - он прекращает выполнение, то есть откладывается ровно тогда, когда закончился квант; меньше - процесс может запустить операцию ввода-вывода в середине своего кванта, откладывается раньше. Он ставится в хвост очереди, на исполнение выбирается новый процесс из начала очереди, и таймер заново запускается, чтобы следующий процесс получил свой квант.

2. Если устроено так, что продолжительность остатка текущего CPU burst процесса, который сейчас работает на процессоре - **больше кванта времени**, то по истечении кванта процесс прерывается таймером и помещается в конец очереди процессов, которые готовы к исполнению, а процессор выделяется для использования процессу, который находится в ее начале.

Round Robin - очень хороший алгоритм, который мы ещё обсудим в контексте использования на практике, в том виде, в котором мы его рассмотрели, этот алгоритм использовался в системах разделения времени, но в специализированных системах.

время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\mathbf{P_0}$	и	и	и	и	г	г	г	г	г	и	и	и	и	и	и	и	и	и
$\boldsymbol{P_1}$	г	г	г	г	и	и	и	и										
$\boldsymbol{P_2}$	г	г	г	г	г	г	г	г	и									

Посмотрим, как работает при алгоритме Round Robin предыдущий пример, когда есть процессы P_0, P_1 и P_2 (13, 4 и 1 единицы времени) с размером кванта – 4 единицы времени. Нулевой процесс начинает работать, отрабатывает свои 4 единицы времени, запускается первый процесс, он дорабатывает 4 единицы времени и кончается, запускается второй процесс, прорабатывает 1 единицу времени и кончается, после чего на карусели остался всего один процесс. Карусель крутится быстро, процесс все время на процессоре и быстро дорабатывает свои 9 единиц времени, не уходя с процессора. Состояния процессов показаны на протяжении соответствующей единицы времени: столбец с номером 1 соответствует промежутку времени от 0 до 1. Среднее время ожидания у нулевого процесса - 5 единиц, у первого - 4, у второго - 9 единиц, то есть среднее время ожидания - (5+4+8)/3 = 5,6 (6) единиц времени. Среднее полное время - (18+8+9)/3 = 11,6 (6) единиц времени. Интересно, что при алгоритме Round Robin, если мы закрутим карусель в другом порядке, то средние времена ожидания и выполнения для обратного порядка процессов не отличаются от случая FCFS (2 и 6 единиц соответственно).

время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P_0	и	г	г	и	г	и	г	и	г	и	и	и	и	и	и	и	и	и
P_1	г	и	г	г	и	г	и	г	и									
P_2	г	г	и															

На производительность алгоритма Round Robin сильно влияет величина кванта времени, если выбрать квант, который равен единице, то тот же пример с порядком процессов P_0, P_1 и P_2 будет проходить следующим образом: нулевой процесс доработал свой квант, первый доработал свой квант, второй доработал и закончился, снова нулевой - квант, первый - квант, нулевой - квант, первый - квант и т.д., заканчивается первый процесс, дальше получается пустая карусель, на которой крутится один процесс P_0. В этом случае среднее время ожидания - (5+5+2)/3 = 4 единицы времени, то есть нулевой процесс ждал 5 единиц времени, первый - 5, второй - 2. Среднее полное время исполнения - (18+9+3)/3 = 10 единиц, то есть нулевой процесс ждал все 18 единиц, пока не закончился, первый - 9, второй - 3 единицы времени.

При очень больших величинах кванта времени, когда каждый процесс успевает проработать свой CPU burst до возникновения прерывания по времени (то есть сам откладывается или завершается), алгоритм RR вырождается в алгоритм FCFS. При очень малых размерах кванта возникает иллюзия того, что каждый из n процессов работает на своем собственном виртуальном процессоре с производительностью ~ 1/n от производительности реального процессора. Это так, если не считать, что существуют накладные расходы на переключение, которые резко снижают производительность системы. В действительности, чем меньше квант - тем больше относительный вклад накладных расходов, которые приходятся на каждое переключение константы.

Shortest-Job-First (SJF). Для алгоритмов First-Come, First-Served и Round Robin существенным является порядок расположения процессов в очереди процессов, которые готовы к выполнению. Если ближе к началу очереди находятся короткие процессы, то общая производительность возрастает. Если знать время следующих CPU burst для готовых процессов, то можно выбрать для исполнения процесс с минимальной длительностью CPU burst. Учитывается, что необходимо выполнять те процессы, у которых маленькие потребности. Если таких процессов несколько, то для выбора можно использовать обычный FCFS без квантования времени. Соответствующий алгоритм планирования называется Shortest Job First (SJF)/кратчайшая работа первой. Это один из первых случаев, когда операционная система должна знать будущее, то есть она должна знать, какой будет следующий запрос у процесса на этот промежуток времени на процессоре. В качестве алгоритма краткосрочного планирования SJF может быть, как вытесняющим, так и невытесняющим.

- Если планирование **невытесняющее**, то процессор предоставляется избранному процессу на все требуемое ему время независимо от событий в вычислительной системе. Если мы знаем его будущий CPU burst, то в действительности это будет небольшое время, мы выбираем процесс, у которого он самый маленький.

- Если планирование **вытесняющее** - SJF-планировании, то в нем учитывается появление новых процессов в очереди процессов, готовых к выполнению во время работы выбранного процесса. Если в очереди появляется процесс, у которого CPU burst нового процесса меньше, чем остаток CPU burst у процесса, который сейчас выполняется, то исполняющийся процесс вытесняется новым.

Это пример приоритетного алгоритма, в котором учитывается некоторая характеристика процесса, которая определяет его приоритетность среди всех других готовых процессов. Этот пример показывает, что система в действительности может не знать его приоритет, она может только предполагать, сейчас мы увидим, как она это может делать. Сначала пример работы невытесняющего алгоритма SJF:

Процесс	**P_0**	**P_1**	**P_2**	**P_3**
Продолжительность очередного CPU burst	5	3	7	1

В состоянии готовность находятся четыре процесса P_0, P_1, P_2 , P_3 , для которых известны времена их очередных CPU burst (соответственно 5,3,7,1 единиц времени). Вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, больше у них нет никаких вопросов, процессы не совершают операций ввода-вывода, время переключения контекста пренебрежимо мало.

время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P_0	г	г	г	г	и	и	и	и	и							
P_1	г	и	и	и												
P_2	г	г	г	г	г	г	г	г	г	и	и	и	и	и	и	и
P_3	и															

Первым начинает выполняться третий процесс (P_3), у него CPU burst - 1, он сразу завершается, следующий - первый процесс с CPU burst - 3, после запуска он отрабатывает 3 единицы времени и заканчивается, следующий - нулевой, у него 5 единиц времени, он их отрабатывает и заканчивается, наконец - самый "жадный" второй процесс, у которого 7 единиц, отрабатывает свой промежуток времени CPU burst. Среднее время ожидания у нулевого процесса - 4, у первого - 1, у второго - 9, у третьего - 0 единиц времени. Среднее время ожидания SJF - (4+1+9+0)/4 = 3,5 единицы. порядке процессов P_0, P_1, P_2 , P_3 , то среднее время ожидание было бы - (0+5+8+15)/4 = 7 единиц времени. То есть он дает очевидный выигрыш. Можно доказать, что для заданного набора процессов, если новые процессы не появляются в очереди готовых, алгоритм SJF является оптимальным с точки зрения минимизации среднего времени ожидания среди класса всех невытесняющих алгоритмов.

Процесс	Время появления в очереди	Продолжительность очередного CPU burst
P_0	0	6
P_1	2	2
P_2	6	7
P_3	0	5

Для примера вытесняющего варианта алгоритма SJF планирования рассмотрим ряд процессов P_0, P_1, P_2 , P_3 с различными временами CPU burst (6,2,3,5, самый маленький у P_1) и различным временем их появления в очереди процессов, готовых к исполнению: P_0 в системе появляется сразу, P_1 через 2 единицы времени, P_2 через 6 единиц, P_3 тоже появляется сразу.

время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
P_0	г	г	г	г	г	г	г	и	и	и	и	и	и							
P_1			и	и																
P_2							г	г	г	г	г	г	г							
P_3	и	и	г	г	и	и	и													

В момент времени 0 у нас в состоянии "готовность" находятся 2 процесса: P_0 и P_3 . Меньшее время очередного CPU burst оказывается у процесса P_3, он выбирается и начинает работать, прорабатывает 2 единицы времени, пока среди готовых не появляется и не поступает в систему первый процесс, у него 2 единицы времени, а это

значит, что его CPU burst меньше, чем остаток CPU burst у третьего процесса, который вытесняется. По прошествии еще 2-х единиц времени - первый процесс заканчивается, так как у него всего 2 единицы. Новых готовых процессов ещё не появилось, поэтому опять выбирается третий процесс, он прорабатывает свои 3 единицы времени и заканчивается. К этому моменту t = 6, то есть через 6 единиц времени в очереди появляется P_2 с 7 единицами, а процессу P_3 осталось работать 2 единицы времени, остается P_3. После завершения P_3 в момент t = 7 в очереди P_0 и P_2, выбирается P_0. Последним получает возможность выполняться процесс P_2.

Алгоритм Shortest-Job-First - хороший, он действительно минимизирует среднее время ожидания, это можно и не трудно доказать. Даже интуитивно понятно, что если мы знаем, сколько какому процессу необходимо непрерывного времени на процессоре, то этот алгоритм будет минимизировать среднее время их выполнения. Основная его неприятность при реализации SJF - это то, что мы должны знать будущее, то есть необходимо знать, сколько процессу потребуется времени на процессоре до тех пор, пока он не задаст какой-то обмен или нечто такое, что его заблокирует. В пакетных системах объем процессорного времени, требующееся заданию для выполнения, указывает пользователь. Если это использовать для долгосрочного SJF - планирования, то при тех условиях, что человек сам говорит - сколько времени необходимо его заданию, тогда это делать возможно, это будет точно, потому что система выкинет этот процесс, как только он отработает столько - сколько запросил пользователь. Укажет больше времени, чем нужно - будет ждать получения результата дольше. Если укажет меньшее время, задача может не досчитаться до конца.

Для краткосрочного планирования, конечно, это не так здорово, предлагается довольно несуразный подход к предсказанию: можно только прогнозировать длительности следующего CPU burst, исходя из предыстории работы процесса. Пусть τ (n) - величина n-го CPU burst, T (n + 1) - предсказываемое значение для (n + 1)-го CPU burst, α - некоторая величина в диапазоне от 0 до 1. Определим рекуррентное соотношение T (n + 1) = $\alpha\tau$ (n) + (1a) T(n). T (0) можно взять произвольной константой, потому что мы про него ничего не знаем. Первое слагаемое в этом рекуррентном соотношении учитывает последнее поведение процесса, тогда как второе слагаемое учитывает его предысторию (как мы его предсказывали). Это рекуррентное значение зависит от того, какое значение выбирается у α:

- если α = 0, то за реальным последним поведением процесса система перестает следить фактически, полагая T(n) = E(n-1) = … = (0). Все CPU burst оцениваются одинаково, исходя из некоторого начального предположения, взяв константу.
- если α = 1, то мы забываем о предыстории процесса, мы считаем, что время очередного CPU burst равно времени последнего CPU burst: T(n+1) = τ (n)

- если выбирать $\alpha = ½$, то обычно получают взвешенный учет последнего поведения и предсказания. Такой выбор удобен и для быстрой организации вычисления оценки T (n + 1)

Почему мы так предсказываем - сказать невозможно, потому что это невозможно доказать, более того, есть уверенность, что в реальной жизни это соотношение будет работать неправильно. Если взять поведение какого-либо произвольного процесса, то у него эти промежутки либо будут всегда одинаковыми (так бывает, когда процессы примерно через одно и тоже время запускают какие-то обмены с внешними устройствами), либо это будет совершенно произвольно, потому что в действительности бывают такие процессы, которые ведут себя абсолютно непредсказуемым образом (что может зависеть, например, от входных параметров). Это один из примеров совершенно несуразное поведения операционных систем для при предсказании будущего. Нормальные предсказания будущего связаны только с управлением виртуальной памятью, это связано с принципом локальности. Здесь о локальности речь не идет, никак нельзя сказать, что если в прошлый раз следующий обмен был задан через 23 миллисекунды, а мы полагали, что это будет 20 миллисекунд, то следующий раз это будет приблизительно 21,5 миллисекунды. Мы в действительности не знаем, как ведут себя процессы, операционная система ничего не знает про процессы: какие выполняются программы, каковы цели процессов, она только наблюдает, как они себя вели в прошлом, и на основании этого делает какие-то предположения о том, как они будут себя вести в будущем.

Гарантированное планирование. Если при интерактивной работе N пользователей в вычислительной системе, то можно применять алгоритм планирования, который гарантирует, что каждый пользователь будет иметь в своем распоряжении ~ 1/N часть процессорного времени. Пронумеруем пользователей от 1 до N. Для пользователя с номером i введем T_i - время нахождения пользователя в системе и τ_i - суммарное процессорное время, которое уже выделено всем его процессам в течение сеанса, то есть сколько он просидел астрономического времени, сколько его процессы получили реального процессорного времени. Вероятно, это относится к категории справедливого деления, для систем разделения времени такая процедура разумна, хотя это зависит от того, что делают в системе операторы, потому что в системах разделенного времени совершенно необязательно все делают одно и тоже. Действия, которые требуются при запросе билета Москва-Ставрополь, существенно проще и быстрее выполняются, чем если запросить билет до Веллингтона в Новой Зеландии, желательно с пересадкой в Сингапуре с промежутком ожидания не более 2-ух часов. Подбор второго варианта требует существенно больше процессорного времени. Справедливым было бы получение T_i/N процессорного времени.

- Если $\tau_i << T_i/N$, то i - й пользователь обделен процессорным временем
- Если $\tau_i >> T_i/N$, то система явно благоволит к пользователю с номером i

Тогда для каждого пользовательского процесса вычислим значение коэффициента справедливости $\tau_i N/T_i$ и будем предоставлять очередной квант времени процессу с наименьшей величиной этого коэффициента, то есть процессу, который наиболее "обижен".

Недостаток алгоритма гарантированного планирования состоит в том, что пользователи интерактивные и невозможно предугадать их поведение. Именно для таких пользователей существует разделение времени, чтобы обеспечивать те самые 4 сек. времени реакции. С точки зрения системы пользователь становится "обиженным", если он долго сидит, а его процессы не работают на процессоре. Если, например, пользователь решит систему обмануть, отойдя от своего рабочего места, что ещё хуже - начнет пить чай и не будет смотреть на монитор, то когда он заново обратиться к системе, продолжит работу своей сессии, то он будет системой считаться несправедливо обиженным, а все его процессы резко заработают, то есть ему будут давать преимущество - неоправданно много процессорного времени.

Если Shortest-Job-First (SJF) имеет смысл в полутеоретическом смысле, потому что показывает, как можно было бы исправить First-Come, First-Served (FCFS) (фактически это исправление, то есть мы подсчитываем - кому меньше сейчас требуется ресурсов и его пропускаем вперед), то алгоритм гарантированного планирования - это игра ума, которая в чистом виде на практике работать не может. Тем не менее алгоритмы SJF и гарантированного планирования представляют собой частные случаи приоритетного планирования, то есть каждому процессу присваивается некоторое числовое значение - приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. В случае SJF в качестве приоритета выступает оценка следующего CPU burst. Фактически, чем меньше оцененное значение CPU burst, тем выше приоритет процесса. Для алгоритма гарантированного планирования приоритетом служит вычисленный коэффициент справедливости, то есть чем процесс более обижен, тем у него выше приоритет, тем больше его необходимо поощрять.

В целом принципы назначения приоритетов могут опираться как на внутренние критерии вычислительных систем, так и на внешние критерии.

- **Внутренние критерии** - это разные количественные и качественные характеристики процесса, то есть это могут быть ограничения на время использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних I/O burst (промежутков ожидания завершения ввода-вывода) к CPU burst и т.д.

- **Внешние критерии** - это важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и другие политические факторы.

Приоритетное планирование может быть, как вытесняющим, так и невытесняющим, при вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет выполняемый процесс с более низким приоритетом.

В случае невытесняющего планирования высокоприоритетный процесс просто становится в начало очереди готовых процессов и, соответственно, получает процессор, когда низкоприоритетный отложатся.

Процесс	Время появления в очереди	Продолжительность очередного CPU burst	Приоритет
P_0	0	6	4
P_1	2	2	3
P_2	6	7	2
P_3	0	5	1

Пример: здесь числовые приоритеты, так принято исторически: чем меньше значение - тем выше приоритет. Пусть в очередь готовых процессов поступают те же процессы, с теми же характеристиками, что и в примере для алгоритма SJF, но им присвоены приоритеты. Нулевой процесс в нулевое время, с запросом 6 единиц времени появляется с приоритетом 4, первый процесс во вторую единицу времени появляется с запросом на 2 единицы времени CPU burst - с более высоким приоритетом 3, у третьего процесса, который появляется через 6 единиц времени, запрос 7 единиц времени с приоритетом 2, у четвертого тоже появляется сразу с запросом 5 единиц времени и приоритетом 1 - самым высоким. Тогда происходит следующее: большее значение соответствует меньшему приоритету, т.е. самый высокоприоритетным является процесс P_3, он начинает работать первым, поэтому он отрабатывает весь свой CPU burst и завершается (мы опять предполагаем, что у процессов больше ничего нет, кроме одного CPU burst). После него из готовых процессов у нас есть только P_0, который самый низкоприоритетный, он образован в момент времени t = 3, остается процесс P_1, который получает процессор и отрабатывает свои 2 единицы времени. Это невытесняющий вариант: хотя в тот же момент времени появился P_2, он не вытесняет P_1, который завершает свое выполнение. После чего идет следующий по приоритету процесс P_2, который отрабатывает все свои 7 единиц времени и заканчивается, наконец работает самый низкоприоритетный процесс P_0, он отрабатывает 6 единиц времени.

В случае вытесняющего приоритетного планирования:

время	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
P_0	г	г	г	г	г	г	г	г	г	г	г	г	г	г	и	и	и	и	и	и
P_1			г	г	г	и	г	г	г	г	г	г	г	и						
P_2							и	и	и	и	и	и	и							
P_3	и	и	и	и	и	и	и													

Ситуация отличается тем, что в момент времени t = 7 процесс P_2 вытесняет с процессора процесс P_1, вытеснил и работает - он самый высокоприоритетный, отрабатывает до конца, потом пропускает процесс P_1, который дорабатывает свой CPU burst, последним опять же работает процесс P_0 - самый низкоприоритетный. Здесь на первый план ставится важность - это самая главная характеристика процесса. В соответствии с этой важностью они обрабатываются в нужном порядке.

В этих примерах приоритеты процессов не изменялись со временем, то есть они были статические. Такой подход легко реализовать, он вызывает небольшие расходы на выбор наиболее приоритетного процесса. Однако статические приоритеты не реагируют на изменения ситуации в вычислительных системах. Более гибкими являются динамические приоритеты процессов, у которых меняются значения по ходу исполнения процессов. В ранних версиях системы Unix, которая не просто так называлась Time-Sharing System, действительно преследовалась цель справедливого разделения времени. Неизвестно, использовался ли когда-нибудь Unix именно как система разделения времени, но цель такая была, с одной стороны. С другой стороны, Unix разрабатывался на компьютере PDP-8 - это была 8 разрядная система с очень маленькой основной памятью. А первые варианты Unix, которые работали на более-менее нормальных компьютерах, работали на PDP-11 - это 16 разрядная система, в которой тоже было немного памяти (это происходило в конце 70-ых годов прошлого века). А процессов хотелось иметь побольше, поэтому они делали свопинг: есть много процессов, но не все образы присутствовали в основной памяти. Это делалось на основе одновременного управления памятью и поддержки приоритетов. Задача была такая: система квантовалась, но квант выставлялся для всех не одинаковый, а в зависимости от того, какой у процесса приоритет. Процесс запускается, в начале у него высокий приоритет, но при этом он получает маленький квант времени, то есть он крутится на карусели и квантуется с другими высокоприоритетными процессами по маленьким квантам. По ходу того, сколько астрономического времени он крутится на карусели, у него начинает падать приоритет. Чем меньше приоритет, тем больше величина кванта, то есть он получает все большие кванты времени, но всё реже, потому что у него не хватает на это приоритета. В конце концов приоритет падает до такого нижнего состояния, что процесс теряет право на присутствие в памяти. Тогда его откачивают на диск, а вместо него с диска (если там есть такой процесс) берут другой, который получает высокий приоритет и маленький размер кванта. У процесса, который был перенесен на внешнюю память, после этого начинает расти приоритет, пока этот процесс не становится кандидатом на возвращение в основную память, то есть он возвращается с высоким приоритетом и с маленьким квантом, потихонечку приоритет у него падает, квант увеличивается, он уходит во внешнюю память, и т.д. Получается необычная карусель.

Могут быть разные подходы к изменению приоритетов: есть начальное состояние динамического приоритета, присвоенное процессу, оно действует в течение короткого времени, после этого назначается новое, более подходящее значение

приоритета. Единственная операция над процессами, которую мы не обсуждали. Изменение приоритета процессов проводится согласованно с совершением других операций: при рождении нового процесса, при разблокировке или блокировании процесса, по истечении кванта времени или по завершении процесса. Это зависит от общей политики построения системы. Примерами алгоритмов с динамическими приоритетами являются алгоритм SJF и алгоритм гарантированного планирования. Потому что приоритеты меняются со временем, хотя правила изменения приоритетов весьма необоснованные. Схемы с динамическими приоритетами гораздо сложнее в реализации и связаны с большими издержками по сравнению со статическими приоритетами и связаны с большими накладными расходами. Однако их использование предполагает, что эти издержки оправдываются улучшением поведения системы.

Лекция 4. Кооперация процессов и алгоритмы синхронизации

Алгоритмы планирование. Заключение

Прошлый раз мы остановились на теме приоритетного планирования. **Приоритет** - это некоторое числовое значение, которое показывает насколько большими правами обладает процесс при выборе его на процессор. Примерами приоритетных алгоритмов являются Shortest-Job-First (SJF) и гарантированное планирование. Кроме того, мы рассмотрели схему Unix, когда приоритет процесса меняется со значением приоритета, в частности процесс (его образ) может быть откачан во внешнюю память.

Проблемой приоритетного планирования является то, что если приоритеты назначены процессом достаточно произвольным образом, и нет никакой политики изменения приоритетов, то при ненадлежащем выборе механизма назначения низкоприоритетные процессы могут никогда не получить процессор, то есть они навсегда останутся незаконченными. Рано или поздно приходится вычислительную систему перезапускать и такие процессы теряются. Это нехорошая ситуация, её стандартное решение: если какой-то процесс не работает, а для каждого процесса есть его локальное время жизни, то система может смотреть, какие процессы работали, а какие не работали, если у процесса не растет локальное время жизни, то это означает, что ему никогда не дают процессор. Если процесс по-прежнему остается не обслуженным, то система может, например, повышать ему приоритет до тех пор, пока он не доберется до процессора. Решением является увеличение со временем значения приоритета процесса, находящегося в состоянии "готовность". Пусть изначально процессам присваиваются приоритеты от 128 до 255, каждый раз, по истечении некоторого промежутка времени, значения приоритетов готовых процессов уменьшаются на 1. Процессу, побывавшему в состоянии "исполнение", восстанавливается первоначальное низкое значение приоритета. Когда мы дойдем до тупиков, особенно до тупиков, которые могут вызываться использованием семафоров, то есть синхронизацией процессов, то познакомимся с тем, какие иногда в приоритетной схеме бываю разновидности тупиков, как с ними можно бороться.

Многоуровневые очереди (Multilevel Queue) – это последний подход, который является семейством алгоритмов управления процессами. В этом случае для каждой группы процессов, находящихся в состоянии "готовность", создается своя очередь. Очередям приписываются фиксированные приоритеты. Внутри очередей для планирования могут применяться разные алгоритмы: для больших счетных процессов, не требующих взаимодействия с пользователем (фоновых процессов), может использоваться алгоритм FCFS, а для интерактивных процессов – алгоритм Round Robin. Подход многоуровневых очередей повышает гибкость планирования: для процессов с различными характеристиками применяется наиболее подходящий им алгоритм. Если мы пойдем снизу-вверх по схеме (рис.4.1.) - то, например, в самом низу

находятся процессы, которые запускаются для студентов, они самые бесправные и запускаются в такой модели на самом низком приоритете, а обслуживаются по алгоритму Round Robin; выше находятся фоновые процессы, которые, например, что-то считают на фоне всего основного, их совершенно ни к чему квантовать и можно обслуживать по алгоритму First-Come, First-Served. Выше опять используется алгоритм Round Robin, но с разными приоритетами: если на уровне системных процессов на нулевом приоритете есть готовый процесс, то они и будут работать; если все они отложились, то крутится карусель процессов ректората, и т.д. Для такой грубой схемы это работает, хотя схема утрирована.

Рис. 4.1. Несколько очередей планирования

Многоуровневые очереди с обратной связью являются более жизненными. В этом случае приоритет очереди определяется не тем, какие процессы в ней находятся, в такой схеме они все должны работать в системе разделения времени. Это зависит не от того, от имени какого пользователя запустился этот процесс, а от того, сколько ему необходимо времени на процессоре. Развитием алгоритма многоуровневых очередей является добавление к нему механизма обратной связи. Процесс не постоянно приписан к определенной очереди и может мигрировать из очереди в очередь, в зависимости от своего поведения. Рассмотрим, как это может работать: на самом высоком приоритете работают процессы с очень маленьким квантом. Примером, когда процессу имеет смысл работать с маленьким квантом, является ситуация, когда все интерактивные пользователи занимаются написанием какого-то текста, например, текста программы. Когда человек пишет текст, то в основном процесс ждет пользователя, а не он процесс. Именно такие процессы прекрасно квантуются с очень маленьким квантом и именно они все время будут недорабатывать свой квант. Представим, что в какой-то момент времени люди, которые одновременно пишут программы, дописали их и решили запустить свою программу на компиляцию, а компилятору уже необходимо процессорное время. Находясь на этой очереди с маленьким квантом, его процесс начинает всё время на ней крутиться и не

откладывается, CPU burst становится большим, ему не хватает этого кванта. Система это видит, срабатывает квант 8, который должен означать конец этого кванта времени, а процесс не отложился, он хочет работать дальше. Далее он 1-2 раза проезжает в очереди на карусели до процессора, система снижает его приоритет, например, до первого приоритета, увеличивая размер кванта. Дальше может быть следующее: либо процессу хватает этого кванта, то есть его CPU burst меньше 16 единиц, тогда он отрабатывает свое CPU burst - откладывается, потом становится готовым, попадает в конец очереди, то есть очередь начинает вести себя нормально. Либо ему не хватает 16 единиц времени (например, это большая программа, которая долго компилируется), тогда ему ещё снижают приоритет, и он попадает на квант - 32 единицы.

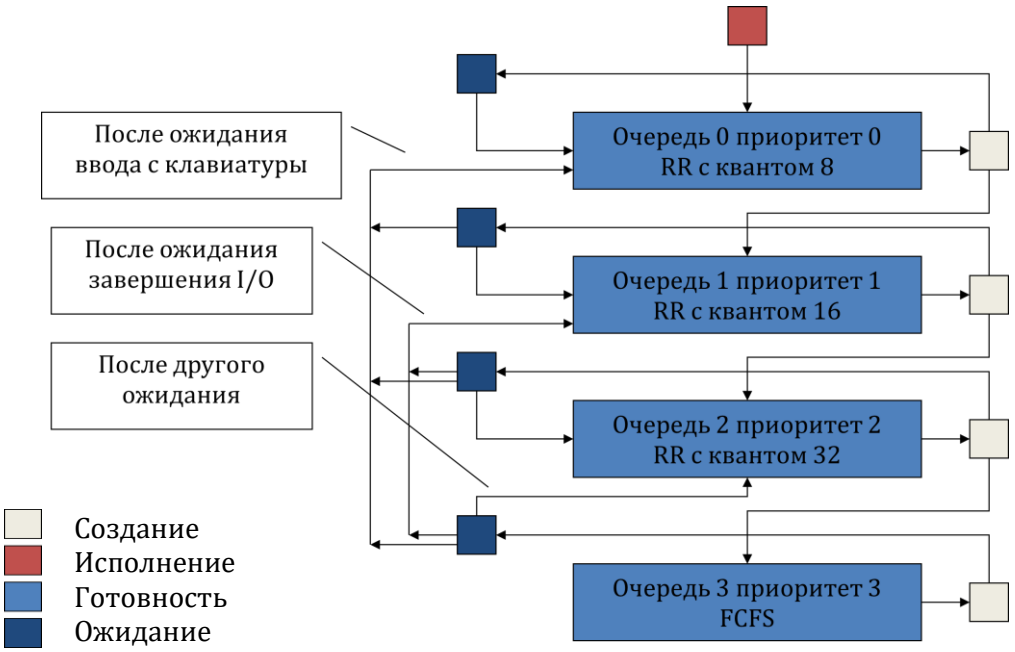

Рис. 4.2. Схема миграции процессов в многоуровневых очередях планирования с обратной связью

Итого: родившийся процесс поступает в очередь 0, при выборе на исполнение получает квант времени размером 8 единиц. Если продолжительность его CPU burst меньше этого кванта времени, то процесс остается в очереди 0, иначе он переходит в очередь 1. Для процессов из очереди 1 квант равен 16, если он в него укладывается - остается в очереди 1, иначе переходит в очередь 2 - квант 32. Если не укладывается - в очередь 3 без квантования времени.

Таких очередей не делают много, в нашей системе их делалось две: очередь с маленьким квантом и очередь с большим квантом. Кроме того, для совсем низкоприоритетных процессов разумно запустить очередь First-Come, First-Served, которая не будет квантоваться, то есть процессы дорабатывают свой CPU burst до тех

пор, пока могут. Но это самая низкоприоритетная очередь, то есть до неё дело доходит только тогда, когда все остальные очереди не содержат готовых процессов.

Миграция процессов в обратном направлении может осуществляться по различным принципам. Когда компиляция закончилась, то пользователь смотрит на распечатку, если компилятор выдает ошибки, он начинает исправлять ошибки, править текст. Это работа, у которой тоже очень маленький CPU burst, для которой необходим маленький квант Процесс начинает все время недорабатывать свой квант в той очереди, в которую он попал, когда шла компиляция. Тогда система начинает ему повышать приоритет с одновременным уменьшением кванта, это происходит то тех пор, пока он не начнет в него укладываться. Это достаточно гибкая схема, она хорошо работает: после завершения ожидания ввода с клавиатуры, процессы из очередей 1, 2 и 3 могут помещаться в очередь 0. После завершения дисковых операций ввода-вывода процессы из очередей 2 и 3 могут помещаться в очередь 1. После завершения ожидания всех других событий - из очереди 3 в очередь 2.

Многоуровневые очереди с обратной связью - это наиболее общий подход к планированию процессов из числа рассмотренных. В операционных системах, чем ближе к ядру, тем, - более ответственные задачи (вроде бы), тем меньше кода они занимают. Они могут быть достаточно тяжелые в том смысле, что их необходимо хорошо продумать, чтобы все было компактно, корректно и т.д. Для полного описания конкретного воплощения многоуровневых очередей необходимо указать: количество очередей для процессов, находящихся в состоянии "готовность"; алгоритм планирования, действующий между очередями; алгоритмы планирования, действующие внутри очередей; правила помещения родившегося процесса в одну из очередей; правила перевода процессов из одной очереди в другую. Для написания операционной системы сначала использовался язык ассемблера, потом использовалось C, на котором и сейчас пишутся системы. Его преимущество заключается в том, что это достаточно прозрачный язык, там видно, что мы делаем, он не слишком поднят над уровнем систем команд, поэтому понятно, какие ресурсы тратятся. Можно писать на разных языках, но языки более высокого уровня уже более оторваны от архитектуры компьютера. Например, если писать программу на языке Ада (очень хороший и мощный язык, на котором можно написать операционную систему), но не имеет смысла писать на Java. Но только потому, что это не компилированный и не интерпретированный язык, даже если он just-in-time compilation, то для операционных систем это не годится. Естественно, не надо писать операционные системы на скриптовых языках.

Заключение:

- Для распределения процессорного времени между процессами в вычислительной системе применяется процедура планирования процессов.
- Она может работать на разных уровнях:

  - краткосрочное планирование процессов - если есть мультипрограммирование;
  - среднесрочное планирование процессов - если есть свопинг;
  - долгосрочное планирование процессов - если присутствует управление заданиями.
- Конкретные алгоритмы планирования процессов зависят от целей, которые стоят перед операционной системой, от класса решаемых задач, они опираются на статические и динамические параметры процессов и компьютерных систем.
- Важное понятие - это вытесняющий и невытесняющий режимы планирования:
  - **вытесняющее планирование** - когда при появлении процесса, который имеет больше прав на процессор, тот процесс, который на нем сейчас выполняется, снимается с процесса и переводится в очередь готовых, а запускается тот, который более приоритетный
  - **невытесняющее планирование** - когда процесс остается на процессоре, пока не кончится его CPU burst, то есть пока он сам не захочет отложиться.
- Простейшим алгоритмом планирования является невытесняющий алгоритм **First-Come, First-Served (FCFS)**, но у него есть свой дефект (у всех алгоритмов FIFO есть свои странности, мы увидим это на примере управления виртуальной памятью): он может существенно задерживать обслуживание коротких процессов, не давая им закончиться. Это процессы, которые не вовремя перешли в состояние **"готовность"** после того, как процесс с длинным CPU burst начал работать.
- Более распространена, особенно в системах разделения времени карусельная вытесняющая версия этого алгоритма - **Round Robin (RR)**.
- Среди невытесняющих алгоритмов оптимальным с точки зрения среднего времени ожидания процессов является алгоритм **Shortest-Job-First (SJF)**. К сожалению, он основан на том, что необходимо знать будущее, то есть - сколько процессу потребуется в будущем процессорного времени. Хороших способов предсказания неизвестно, поэтому алгоритм хороший теоретически, практически им пользоваться нельзя.
- В интерактивных системах используется алгоритм **гарантированного планирования.**
- Алгоритм Shortest-Job-First и алгоритм гарантированного планирования - это частные случаи **планирования с использованием приоритетов**.
- При более общих методах приоритетного планирования применяются **многоуровневые очереди процессов и многоуровневые очереди с обратной**

связью. Это действительно адаптивные алгоритмы, здесь понятно, на что операционная система опирается - на то, как себя ведет процесс. Если она ошибается, то она тут же исправляется. Теоретически может оказаться, что процесс ведет себя так, что его CPU burst все время меняется, тогда он будет перемещаться между очередями с высоким и низким приоритетом.

- Будучи наиболее сложными в реализации, эти способы планирования обеспечивают гибкое поведение вычислительных систем и их адаптивность к решению задач разных классов.

Кооперация процессов

В ходе лекции мы обсудим, что такое взаимодействующие процессы и зачем им вообще взаимодействовать; какие бывают категории средств обмена информации между процессами, так как взаимодействие предполагает обмен информацией; как могут быть устроены на логическом уровне механизмы передачи информации; в заключение поговорим про нити исполнения (threads), почему при программировании к ним необходимо относиться с осторожностью.

В основном процессы находятся в очереди к процессору, потому что - сколько бы не было ядер у процессора, процессов в системе обычно все равно больше, то есть они борются за процессор и конкурируют за другие ресурсы, которые ограничены. Для того, чтобы процессы операционной системы функционировали нормально: надежно, эффективно и т.д. - операционная система старается их обособить друг от друга. Для этого каждому тяжелому процессу дается свое собственное адресное пространство, а в настоящее время - своя собственная виртуальная память. Каждый процесс получает собственные дополнительные ресурсы, которые ему необходимы. Но для решения некоторых задач процессам требуется объединять свои усилия. Обсудим причины, по которым процессам следует взаимодействовать, поговорим и о способах их взаимодействия, рассмотрим какие при этом возникают проблемы.

Для достижения той цели, ради которой они запущены, различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться, взаимодействуя между собой. Это взаимодействие может происходить:

- между процессами на одной вычислительной системе в одном процессоре, которые выполняются с чередованием команд в одном узле, то есть concurrently. Это подразумевает либо реальную параллельность, когда несколько процессов выполняется concurrent режиме, либо это происходит квазипараллельно с чередованием операций под управлением операционной системы.

- параллельно на разных вычислительных системах, взаимодействуя между собой.

Для чего процессам нужно заниматься кооперацией и взаимодействием? Какие существуют причины для их кооперации?

1. **Повышение скорости работы** - это основная причина. Как программы, которая выполняется в нескольких процессах, так может быть и всей системы, чтобы она работала более эффективно.

    - **если система однопроцессорная** и один процесс блокирован, то есть ожидает наступления некоторого события, например, окончания операции ввода-вывода внешнего устройства, другие процессы могут заниматься полезной работой, которая направлена на решение общей задачи. Раньше это называлось совмещением обмена со счетом, то есть внешнее устройство работает, если программе есть что делать, пока не закончится обмен. Это можно делать в параллельных процессах, которые должны взаимодействовать с данным процессом.

    - **если система многопроцессорная**, то программа разделяется на отдельные фрагменты (если это возможно), каждый из которых будет исполняться на своем процессоре. Если это действительно многоядерная система с общей памятью, то единственным механизмом является OpenMP, то есть механизм явного распараллеливания с использованием threads, которые взаимодействуют между собой. Это требует разработки соответствующих алгоритмов с самого начала. Многие эксперты много лет занимаются и продолжают заниматься попытками распараллеливания чисто последовательных программ, которые написаны на Фортран или Паскаль. В них нет адресной арифметики, там можно более-менее следить за тем, какие куски программы можно выполнять независимо. И даже это не получается. Автоматически распараллеливать С никто не пытается, в основном из-за того, что там адресная арифметика, там нельзя распараллелить, потому что мы не можем обнаружить, какие есть скрытые взаимодействия между разными частями программы путем смены значений указателей. Этот тупик касается не только параллелизма в многоядерных системах, аналогичная проблема - это распараллеливание программ в больших кластерах, где кроме явного программирования с использованием MPI - больше ничего не придумали. Эту ситуацию ярко подчеркивает то, что суперкомпьютеры, которые обладают десятками тысяч узлов, фактически не имеют задач. Это происходит, потому что для того, чтобы сразу загрузить все узлы - необходимо вручную распараллелить программы на 10 000 отдельных кусочков, которые будут взаимодействовать между собой. Что практически нереально осуществить, тем не менее такая потребность есть. Конечно, параллельные программы надо поддерживать.

2. **Совместное использование данных.** Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с одним и тем же совместно используемом файлом, совместно изменяя их содержимое. Если в операционной системе есть такие возможности, чтобы было можно как-то

координироваться. СУБД точно координируют параллельное изменение базы данных, за изменением файлов следят сами процессы.

3. **Модульная конструкция системы.** Мы помним архитектуру операционной системы с микроядром, в которой есть микроядро, а все остальные компоненты операционной системы и пользовательские приложения - это отдельные процессы, взаимодействующие путем передачи сообщений через микроядро, то есть системный вызов - это обмен сообщений между процессами.

4. **Удобство работы пользователя**. Если, например, в редакторе есть встроенный отладчик, то вполне разумно в одном процессе поддерживать редактирование текста, а в другом - чтобы выдавал сообщения отладчик, чтобы в этой ситуации процессы редактора и отладчика умели взаимодействовать друг с другом (потому что отладчик может запрашивать у редактора какие-то символьные имена и т.д.).

Для того, чтобы процессы взаимодействовали, требуется, чтобы они могли общаться. Общение процессов обычно приводит к изменению их поведения в зависимости от полученных данных. Если деятельность процессов, их способ выполнения программы остается неизменным при принятии ими дополнительной информации о других процессах, то это означает, что они делают вид, что они общаются, а в действительности никак эту информацию не используют, если убрать их партнера, то ничего не изменится.

Процессы, которые действительно влияют на поведение друг друга путем обмена информацией, принято называть **кооперативными или взаимодействующими** процессами. Однако различные процессы под управлением операционной системы в вычислительных системах являются обособленными. Это делается специально, чтобы было можно точно понимать и рассчитывать, что на работу процессора непредсказуемым образом не влияют другие процессы, то есть они должны быть защищены. Работа одного процесса никогда не должна приводить к нарушению работы другого процесса. Для этого, в частности разделяются, становятся независимыми их адресные пространства и системные ресурсы. Для того, чтобы обеспечить корректное взаимодействие процессов, требуются специальные средства и действия операционной системы. Просто так поместить значение, которое вычислено в программе, которая выполняется в одном процессе, в переменную программы, которая выполняется в другом процессе - нельзя.

Основные аспекты организации совместной работы процессов

Категории средств обмена информацией. Принято классифицировать средства обмена данными между процессами на 3 категории в зависимости от объема передаваемых данных и степени возможного воздействия:

- **Сигнальные средства. Сигнал** - это некий объект операционной системы, который может передавать минимальное количество информации - один бит: "да" или "нет". По своей природе он используются только для того, чтобы было можно известить объект о наступлении какого-либо события, то есть про него ничего нельзя узнать, кроме этого. Можно представить, например, что сигнал - это когда у двух войсковых подразделений имеются сигнальные ракеты только одного цвета. Произошло нечто, ракета запускается, наблюдатель понимает, что у соседа что-то происходит, но что произошло - неизвестно. Если бы ракеты были двух цветов (зеленая и красная), то было бы два сигнала, в этом случае можно окрасить значением: зеленая ракета - всё в порядке, красная - авария. Но бит всё равно один, только сигналы разные. Степень воздействия на поведение процесса, который получает информацию через сигнал, - минимальна. Всё зависит от того, знает ли процесс, что означает полученный сигнал, надо ли на него реагировать и каким образом. Две воинские части запускают зелены ракеты на ночь, чтобы каждая из них знала, что у второй все в порядке, солдатам можно идти спать. Если кто-то запускает красную ракету, то необходимо быть в готовности, если ракета всего одного цвета, то это настолько мало информативно, что предпринимать никаких действий нельзя. Неправильная реакция на сигнал или его игнорирование могут привести к непоправим последствиям, необходимо реагировать на сигнал в соответствии с договоренностями. В конце 90-х годов мне с коллегой пришлось сделать специальную систему для внешней машины front-end processor суперкомпьютера Электроника СС БИС, которая была устроена очень своеобразным образом. Это был голый процессор с памятью, из всех внешних устройств был только очень быстрый внешний канал, к которому можно было присоединять другие машины. Для того, чтобы было можно его загружать, разгружать, смотреть, что происходит и т.д. - делалось программное обеспечение на внешней машине, которая взаимодействовала с операционной системой центрального процессора основной машины. В процессе проектирования в Unix было необходимо делать систему, в которой было около 1,5 десятка процессов с разными функциями, которые между собой взаимодействовали. Можно было быть уверенным только в одном (это было предусловие): что программа всегда будет работать в одном компьютере, что она никогда не будет разнесена по сети. Когда программу начинали разрабатывать, то искали в Unix самые дешевые средства коммуникации, которые позволят такому количеству процессов общаться данными так, чтобы этого было достаточно. Начиналось с того, что пытались обойтись сигналами, но оказалось, что в Unix того времени только два сигнала остаются в распоряжении людей, которые разрабатывают свои программы, все остальные заняты операционной системой, то есть имеется множество сигналов, которые она генерирует для процесса. Он может на них реагировать, на некоторые процесс обязан реагировать, а на некоторые не может реагировать, например, kill.

Оказалось, что сигналы - это самое дешевое, что есть в Unix, но, когда необходимо общаться нескольким процессам, этого не хватает категорически.

- **Канальные средства.** Общение процессов происходит через линии связи, предоставляемые операционной системой. Это совсем необязательно аппаратные линии связи - это абстракция, которая обеспечивается операционной системой, позволяющая передавать данные от одного процесса другому. Объем передаваемых данных в единицу времени ограничен пропускной способностью линий связи. Чем больше можно передать, тем больше возрастает возможность влияния на поведение другого процесса.

- **Разделяемая память.** Это тоже понятие Unix, которое появилось после того, как в Unix возникла виртуальная память, совместно используемые сегменты. **Сегмент** - это некоторая область памяти, которую можно подключить к виртуальной памяти нескольких процессов. Один процесс (из этого числа взаимодействующих процессов) создает такой сегмент, а остальные его к себе подключают, заранее договорившись о том, как он будет называться. Поскольку он входит в виртуальную память всех процессов, которые должны взаимодействовать, то возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, потому что ему можно писать, а другим - читать. Но для этого требуется очень большая осторожность, потому что если это делать без координации, то ничего хорошего не произойдет. Самое главное, это настолько эффективно, что если бы было можно обойтись без координации, то процессы могли бы общаться без привлечения операционной системы: сегмент подключен, одни программы пишут, другие читают. Теоретически разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

Логическая организация механизма обмена данными

При рассмотрении любого из общения процессов, мы будем иметь ввиду не их физическую реализацию (то есть наличие, например, общей шины данных, линий прерываний, аппаратно разделяемой памяти и т. д.), а логическую организацию, которая определяет механизм их использования. Некоторые важные аспекты логической реализации являются общими для всех категорий средств связи, некоторые относятся к отдельным категориям. Кратко охарактеризуем основные аспекты, которые необходимо иметь ввиду при освоении того или иного способа обмена данными:

Как устанавливается связь? У нас есть какое-то средство обмена данными, можно ли использовать его для обмена информацией сразу после создания процесса или сначала необходимо каким-либо образом связь установить, инициализировать механизм обмена? Например, для использования общей памяти, общих сегментов различными процессами вначале потребуется обращение к операционной системе, для того чтобы эту память образовать, чтобы этот сегмент создать. Если, например, мы

хотим обмениваться данными через сигнал от одного процесса к другому, то никакая инициализация не нужна - это встроенные механизмы операционной системы, все два сигнала, которые доступны пользователям, - пронумерованы. Их можно объявлять и ждать. Если данные передаются по линиям связи, то сначала может потребоваться что-то похожее на резервирование линии.

К этой же линии примыкает вопрос о том, как адресовать партнера при использовании средства связи. При передаче данных их передает тот процесс, который хочет их передать, но ему необходимо знать - кому их передать, а для получения данных нужно знать - откуда он хочет их получить. В первом случае это похоже на адресата в электронной почте, во втором - это некая фильтрация, иногда для программирования это требуется.

Различают два способа адресации: прямую и косвенную (непрямую).

- В случае **прямой адресации** взаимодействующие процессы непосредственно общаются друг с другом, явно указывая при каждой операции обмена данными имя или номер процесса, которому информация предназначена, или от которого она должна быть получена. Например, передать данные процессу 5 или процессу сервера баз данных, получить данные от 7-го процесса.

  - Если оба процесса (процесс-передатчик и процесс-приемник) указывают имена своих партнеров по взаимодействию, то такая схема адресации называется **симметричной прямой адресацией**. В этом случае ни один другой процесс не может перехватить посланные или подменить ожидаемые данные, то есть между процессами прямая линия.

  - При **асимметричной прямой адресации** только один из взаимодействующих процессов, например, передающий (так бывает чаще всего), указывает имя своего партнера по кооперации, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например, ожидает получения информации от произвольного источника. Если, как часто бывает в операционной системе, какой-то дежурный процесс собирает некую статистику, то его все знают, у него предопределенное имя, все процессы могут к нему обращаться. Ему при этом все равно, кто к нему обращается, ему важно, чтобы передавали данные для сбора статистики.

- При **непрямой адресации** (косвенной, которая представляет собой что-то вроде промежуточного почтового ящика) данные помещаются передающим процессом в некоторый промежуточный объект, из которого данные могут быть затем взяты каким-либо другим процессом. В этом случае передающий процесс не знает, как именно называется процесс, который получит информацию, а принимающий процесс не знает идентификатора процесса, от которого он должен ее получить. Пример: мы хотим с каким-то процессом установить

симметричную прямую связь, но мы знаем только - что мы хотим получить, и не знаем - как он точно называется. Мы хотим запросить какой-то виртуальный принтер, который, как мы знаем, выглядит как процесс, тогда мы обращаемся к менеджеру виртуальных принтеров с запросом на принтер, процесс (как бы промежуточный почтовый ящик) создает виртуальный принтер и сообщает в ответ, что у него такой-то идентификатор процесса. В первый раз он может передать, что происходит подключение сразу при создании, тогда оба процесса (который хочет печатать и который может печатать) начинают быть прямо связанными. Это вполне нормальная косвенная адресация, которая часто бывает очень полезной.

При использовании прямой адресации связь между процессами в классической операционной системе устанавливается автоматически, без дополнительных инициализирующих действий. Если известно, как идентифицируются процессы, участвующие в обмене данными, то больше ничего не нужно. При косвенной адресации инициализация средства связи может требоваться, а может и не требоваться. Необходимо знать идентификатор промежуточного объекта для хранения данных, если он, конечно, не является в вычислительной системе уникальным. Например, есть некая услуга, которая обеспечивается n процессами, при этом пользователей устраивает любой из них, тогда перед этими n процессами делается "почтовый ящик", мы обращаемся к менеджеру услуг за конкретной услугой, идентификатор "почтового ящика" посылает туда сообщение, его принимает тот из процессов, который готов нам эту услугу сейчас оказать и оказывает её. Назад он может обратиться напрямую, а может через "почтовый ящик".

Валентность процессов и средств связи. Слово "валентность" здесь использовано по аналогии с химией: сколько процессов может быть одновременно ассоциировано с конкретным средством связи? Сколько таких средств связи может быть задействовано между двумя процессами?

При прямой адресации (точка-точка, процесс-процесс) только одно фиксированное средство связи может быть задействовано для обмена данными между двумя процессами, и только эти два процесса с этим средством связи могут быть ассоциированы. Если адресация косвенная, то может существовать более двух процессов, которые используют один и тот же объект для обмена данными, и более одного объекта может быть использовано двумя процессами.

Однонаправленные или двунаправленные связи, что не связано прямо с адресацией, так как может быть при любой адресации:

- однонаправленная связь (симплексная) - означает, что процесс может использовать средство связи либо только для приема информации, либо только для ее передачи, то есть через одно средство связи нельзя и передать, и принять данные.

- двунаправленная связь (дуплексная) - означает, что каждый процесс, участвующий в общении, может использовать связь и для приема, и для передачи данных.

Дуплексной принято называть двунаправленную связь, когда есть возможность одновременной передачи информации в обе стороны по инициативе любого из процессов, который в этой связи участвует, а **полудуплексной,** которая является самой распространенной разновидностью связи, когда данные можно передать в одну сторону и получить ответ в другую, то есть процесс, который принял данные через полудуплексную связь, получает право послать их назад. Но это не тоже самое, что послать какое-либо другое сообщение, это именно ответ. Полудуплексная связь самая распространенная, потому что это чрезвычайно похоже на вызов подпрограммы, то есть когда процесс посылает сообщение какому-то другому процессу в этом режиме, то фактически он запрашивает у него некую услугу, семантика которой известна только этим двум процессам. Главное, что может случиться так, что процесс не сможет продолжаться до тех пор, пока не получит ответ, пока эта процедура не будет выполнена. Это чрезвычайно полезный режим, потому что когда-то именно на полудуплексной связи был реализован механизм удаленных вызовов процедур в Unix (что было крайне полезно), на этом же режиме были основаны все системы, в которых взаимодействовали распределенные в сети объекты с вызовом методов. Всеми любимые, замечательные сокеты (механизм обмена сообщениями) тоже используются в полудуплексном режиме.

Передача информации с помощью линий связи

Передача информации между процессами с помощью линий связи безопаснее разделяемой памяти и информативней сигнальных средств, потому что никакой процесс ничего не может получить до тех пор, пока не примет сообщение от своего партнера, и ни один процесс не может записать в чужую память, он может только послать данные через канал связи. Кроме того, разделяемая память не может быть использована для связи процессов, которые разделены в локальной сети на различных вычислительных системах. Поэтому каналы связи получили наибольшее распространение среди всех средств коммуникации процессов. Обсудим некоторые вопросы, которые связаны с логической реализацией канальных средств коммуникации.

Буферизация. Это вопрос о том, может ли линия связи сохранять данные, переданные одним процессом, до тех пор, пока ее не получит другой процесс или эта информация не будет перемещена в какой-то промежуточный объект. Каков объем этого промежуточного буфера? Иными словами, речь идет о том, обладает ли канал связи буфером (буферной памятью) и какой размер этого буфера. Здесь можно выделить три принципиальных варианта:

1. **Буфер нулевой емкости или отсутствует**. Если процесс хочет послать другому процессу данные, то он должен ожидать, пока процесс, который ожидается как приемник этих данных, не скажет, что он готов их получить. Только после этого передатчик сможет продолжать выполнение, наоборот так же: если какой-то процесс хочет получить данные и говорит системе receive, то он замирает, пока его напарник не пошлет туда данные. Никакая информация не может сохраняться на линии связи. Это схема фактически синхронного обмена сообщениями, то есть ни один процесс не может продолжать свое выполнение до тех пор, пока у него не возьмут посылаемое им сообщение. Кажется, что это очень грубо, однако, например, в MINIX 3 у Эндрю Таненбаума в микроядерной операционной системе все построено на сообщениях. Они как раз выбрали вариант без буферизации. Вроде бы много задержек, но это очень дешевый способ. Понятно, что это нельзя сделать без помощи операционной системы, потому что данными обмениваются тяжелые процессы, каждый со своей виртуальной памятью. То есть необходимо, чтобы, когда какой-то процесс говорит системе receive, а есть процесс, который к этому моменту уже сказал sent и замер, то операционная система должна полезть в виртуальную память процесса-отправителя, забрать оттуда данные и переписать их в буфер, который указал для получения сообщения процесс-получатель. Мы видим, что происходит всего одна перепись, а это очень дешево (и в смысле накладных расходов, и для производительности), потому что переписи данных - это "бич божий" операционных систем, которым довольно часто приходится переписывать большие объемы данных из одного места в другое. Чем больше удается с этим побороться, тем эффективнее используются ресурсы и быстрее все работает.

2. **Буфер ограниченной емкости.** Если размер буфера равен n, то есть линия связи не может хранить до момента получения более чем n единиц информации, а какой-то процесс выполняет операцию sent и посылает сообщение через эту линию связи другому процессу, тогда если напарник к этому времени уже ждет сообщение, оно ему переписывается (из одной памяти в другую, как и при отсутствии буфера). Если партнер не ждет и буфер не полностью заполнен, то сообщение буферизуется, а процесс-отправитель может продолжать свою работу, система взяла сообщение на отправку. Если процесс хочет послать сообщение, а буфер уже заполнен или места недостаточно, то процесс блокируется до тех пор, пока в буфере не появится свободное место. Теперь процесс ждет, не пока сделают receive, а пока не появится место в буфере, потому что получатель работает следующим образом: если получатель говорит receive, а в буфере ничего нет, то он блокируется и ждет пока его напарник не сделает sent. Если в буфере что-то есть и процесс делает receive, то в его память переписывается именно то, что в буфере находится первым в очереди, потому что все единицы информации хронологически связываются в очередь, и

процессу отдается самая старая часть. При этом, когда ему переписывается кусочек, буфер освобождается на единицу информации. Получатель блокируется в случае, если буфер пуст. Когда мы дойдем до средств синхронизации, то посмотрим, как можно реализовать буфер ограниченной емкости в Unix. Это распространенная схема, потому что процессы могут работать асинхронно, по крайней мере, отправитель, а получатель, как правило, уже имеет, что взять, когда делает receive, потому что для него в буфере что-то находится.

3. **Буфер неограниченной емкости** - это теоретическая возможность. Прекрасная схема, которая практически недостижимая. Если бы было можно рассчитывать, что в буфере всегда есть свободное место, то процесс-отправитель никогда бы не блокировался. Это было бы замечательно, но так сделать невозможно, потому что память все-таки ограниченного размера. Если используется канальное средство связи с косвенной адресацией, то под емкостью буфера обычно понимается тот объем данных, который можно поместить в промежуточный объект для хранения данных.

Поток ввода/вывода и сообщения. Существует две модели передачи данных по каналам связи: поток ввода-вывода и сообщения. При передаче данных с помощью потоковой модели операции передачи/приема информации вообще не знают ни структуры данных, ни содержимого данных. Если какой--то получатель просит прочитать 100 байт из потоковой линии связи, то он не знает и не может знать, были ли они переданы туда одновременно (то есть одной операцией) от пяти процессов порциями по 20 байт, пришли они от одного процесса или от разных. Такие данные представляют собой простой поток байтов без их интерпретации со стороны системы. Одним из наиболее простых способов передачи информации между процессами по линиям связи служит, прежде всего **pipe** (программные каналы, pipe от английского трубка) и **FIFO**.

Pipe – это некоторый канал в вычислительной системе, в один из концов которого процессы могут "лить" данные, а из другого конца принимать полученный поток. Если мы представим себе странную, обрезанную водопроводную трубу, к которой по очереди подходят люди и выливают туда по ведру воды, то на другой стороне трубы, где стоит умывальник - никак не разобраться, какой водой мы сейчас умываемся, из какого ведра. В Unix pipe исторически были первыми средствами обмена данными между процессами, то есть в системе был специальный вызов pipe, который создавал такую трубу. Только процесс, который её создал, обладал возможностью к ней обратиться, то есть только он фактически знал идентификатор этой трубы. Но процессы, которые порождаются в Unix (очень странная схема порождения процессов) системным вызовом fork - являются точной копией процесса-родителя, то есть клоном. Он обладает ровно теми же ресурсами, в его виртуальной памяти находятся те же самые коды программ. Второй системный вызов - это exec: программа, которая

работает в некотором процессе, может попросить операционную систему выкинуть её из этого процесса и запустить в этой же виртуальной памяти новую программу, подготовленный к выполнению файл, который её полностью заменит. Все они наследуют данные о pipe. В действительности, в исходном Unix связываться через pipe могли только процессы, которые находятся в одной иерархии родства, имеющие общего предка, создавшего данный канал связи. Другие процессы, которые не входят в это же дерево, этот pipe не знают, не видят и никак не могут к нему обратиться.

Зачем такая схема была сделана в Unix? Поначалу эта схема казалась странной, казалось, что её создатели перемудрили, но потом Деннис Ритчи (это была его идея) объяснил, почему он предложил сделать такую схему процессов и именно такой способ взаимодействия. Память в то время была 16 разрядная, маленькая, а программы хотелось писать большие, и для того, чтобы на маленькой памяти было можно выполнять большие программы, не было никаких средств, кроме Overlay. **Overlay** - это когда на одно и то же адресное пространство мы настраиваем несколько кусков программы, сама программа или, может быть, операционная система, если в ней поддерживались соответствующие средства - в каждый момент времени обеспечивает присутствие в основной памяти только один из этих кусочков. То есть они меняются по мере того, какой сейчас из них необходимо сейчас выполнять. Overlay - очень неплохой механизм, особенно те, которые выполняются вручную, то есть когда есть президентный монитор в основной памяти, который сам решает, какую из веточек overlay сейчас нужно держать в памяти. Но почему-то в 70-е годы к overlay относились пренебрежительно. Для того, чтобы сделать такую разбивку, мало того, что было необходимо специальным образом скомпоновать программу, ещё было необходимо написать специальную программу при построении overlay, описать дерево. То есть это было ещё одно программирование, причем для сложных программ оно не очень простое. Поэтому идея виртуальной памяти для 16 разрядных машин, конечно, хороша, но толку от нее немного, все равно она остается маленькой.

На маленьких машинах PDP-11 был ещё один интересный вид overlay, который назывался **Memory Resident Overlay**. В этом случае 16 разрядов - это виртуальная память процесса. Можно было заранее настроить несколько кусочков программ на одни и те же виртуальные адреса и заставить систему держать созданный массив в основной оперативной памяти. Тогда смена ветки overlay (это можно было делать только через операционную систему) означала, что мы хотим изменить приписку виртуальной памяти к основной памяти с тем, чтобы у нас сейчас в виртуальной памяти находилась именно та часть программы, которая нам необходима. Memory Resident Overlay - это очень хороший и быстрый механизм. За счет виртуальной памяти добиться того, чего стали добиваться потом, стало возможным только тогда, когда появились 32 разрядные машины. Деннис Ритчи говорил, что он предлагает вместо того, чтобы делать эти хитроумные overlay, дать людям простое средство: fork и exec. В этом случае человек при программировании может сам разбить свою программу на модули. Если он хочет выполнять какой-то второй модуль - делай fork и exec, тогда в

параллельном процессе будет выполняться этот модуль, а взаимодействие будет осуществляется через pipe, если необходимо. Если посмотреть из сегодня, то кажется чудным использовать понятие процессов для структуризации программ, но так придумал в свое время Деннис Ритчи.

Pipe играют ещё одну, совершенно потрясающую роль при программировании на языке **Shell**, на скриптовом языке. В Shell есть такая замечательная вещь, как перенаправление ввода-вывода, то есть у каждого процесса есть стандартные файлы: ввода - stdin, выхода - stdout и stderr - файл, куда выводятся сообщения об ошибках. На языке Shell можно собирать конвейеры команд, перенаправляя вывод одной команды на вход другой. В результате получается очень интересное и красивое программирование Shell, которое унаследовано во множестве скриптовых языков. Все это делается через pipe, потому что у pipe файловый интерфейс. Во-первых, даже обычный pipe имеет точно такие же операции, как у файлов Unix: read, write. У него нет lseek, потому что у него не по чему позиционироваться - это труба, а не файл. Семантика такая: read - всегда из конца, write - всегда в начало (как в трубе). Поэтому и у файлов так, если не делать никаких хитростей. Поэтому ровно на этом держится перенаправление ввода и вывода. При наличии этого механизма ввода-вывода Shell - это мощнейший язык программирования, его даже не с чем сравнить, настолько он мощный. Это происходит за счет того, что можно определять новые команды просто как цепочки существующих команд с перенаправлением ввода-вывода. И все это делается за счет такой простой трубы - совершенно примитивной конструкции.

По мере того, как Unix рос и развивался, к середине 80-х годов стало понятно, что иногда необходимо взаимодействовать не только тем процессам, которые находятся в одной иерархии родства. Если разрешить процессу, создавшему трубу, сообщать о ее местонахождении в системе другим процессам, сделав вход и выход трубы каким-либо образом видимыми для всех остальных. Например, зарегистрировав ее в операционной системе под определенным именем, мы получим объект, который принято называть FIFO или именованный pipe **(named pipe)**. **FIFO** - это объект файловой системы, имеющий свое имя, у него есть, как у обычных файлов, набор прав доступа. То есть известно - какие процессы могут открывать этот файл, какие могут читать, какие могут писать. Такой именованный канал можно использовать для организации связи между любыми процессами в системе, но он, опять же, потоковый, это труба по подписке. Интересно, что, используя pipe, можно прекрасно поить самого себя, мы можем записать в pipe, а потом оттуда и прочитать, а можем прочитать совсем не то, что записали, потому что если сначала прочитает, а потом запишет что-либо какой-то другой процесс, то после мы прочитаем что-то другое, что осталось от него.

Ещё одна вещь, которую полезно сказать про именованные и неименованные pipe: как и для файлов в операционной системе Unix - нельзя думать, что если операционная система не знает структуры данных, которые хранятся в pipe, то их обязательно не знают и процессы, которые pipe используют. Они могут заранее

договориться, что мы обмениваемся через pipe всегда кусками, размер которых написан в первых 10 байтах этого куска, так что читай сначала 10 байт, узнавай размер и дочитывай все остальное, а потом понимай, что это структура. В действительности можно делать нормальный обмен структурированными сообщениями, используя на уровне операционной системы такой потоковый канал связи.

В модели сообщений процессы говорят операционной системе, какая будет иметься структура передаваемых сообщений. В этом случае весь поток информации они разделяют на отдельные сообщения, между данными вводятся границы сообщений. Разница в том, что в этом случае размер сообщения не нужно узнавать - он предопределен, сколько полагается, столько и передадут. К служебной части сообщений могут быть присоединены указания на то, кем конкретное сообщение было послано и для кого оно предназначено. Примерно, как заголовок сообщения электронной почты, которое обычно сжимают, там написано много полезного. Сообщения могут быть одинакового фиксированного размера или могут быть переменной длины. По крайней мере, система знает, какой длины каждое сообщение, соответственно процесс-получатель тоже может узнать - какой длины сообщение он получил.

Для передачи сообщений в Unix имеются разнообразные средства: **очереди сообщений, sockets** (гнезда) и т.д. Реализация очереди сообщений мы рассмотрим с использованием семафоров. Это делается очень легко: очередь сообщений - это некоторый объект ядра операционной системы, есть операция создать MessageQ, в ответ дается соответствующий идентификатор (как в файлах, только это не файл). Как получатель узнает идентификатор сообщений - мы не знаем, это дело этих двух процессов, которые обмениваются сообщениями через очередь сообщений. Есть две основные операции: send и receive. Send - это операция, которая помещает передаваемое сообщение в очередь сообщений, а вся очередь сообщений держится в памяти ядра. Максимальный размер очереди - это размер буфера, он говорится ядру во время создания очереди сообщений. На другой стороне, на стороне получателя - операция receive, она выдает самое последнее по времени сообщение из очереди, либо есть модификация - можно покрасить сообщение, то есть присвоить ему некоторую метку. Можно на стороне получателя говорить, что хотим получить сообщение с такой меткой, тогда не приходится разгребать всю очередь, а ищется сообщение с такой меткой, процесс блокируется, если её нет.

Sockets (гнезда) - очень красивый механизм, это объекты, которые тоже имеют файловый интерфейс со стороны получателя. Они отличаются тем, что для каждого socket можно использовать режим внутри компьютера, то есть через общую память. В этом случае фактически это почти тоже самое, что очереди сообщений, только с другим интерфейсом. А можно тот же socket перестроить на то, чтобы он выходил на TCP/IP, чтобы он выходил в сеть. Тогда, не меняя прикладных программ, мы начинаем взаимодействовать с сетевыми процессами.

И потоковые линии связи, и каналы сообщений могут обладать и не обладать буфером. Если мы будем говорить о емкости буфера для потоковых линий связи данных, то емкость будет меряться в байтах. Если мы будем говорить о емкости буфера для сообщений, то будем измерять ее в сообщениях.

Надежность средств связи

Мы будем называть способ коммуникации надежным, если при обмене данными выполняются четыре условия:

1. Данные не теряются.
2. Данные не повреждаются.
3. Среди данных не появляются лишние.
4. Не нарушается порядок данных в процессе обмена.

Легко увидеть, что если для передачи данных используются разделяемые сегменты, разделяемая память, то это будет надежный способ связи (что туда запишет процесс, что мы сохранили в разделяемой памяти, то и будет записано, ниоткуда ничего другого не появится). Однако для других средств коммуникации это не всегда верно. Каким образом в вычислительных системах можно бороться с ненадежностью коммуникаций?

Давайте рассмотрим возможные варианты на примере обмена данными через линию связи, которые основаны на передаче **сообщений**. Тогда, например, можно каждое передаваемое сообщение снабжать некоторой контрольной суммой, которая вычисляется при посылке сообщения, то есть значение контрольной суммы является служебной частью сообщений, оно передается на сторону получателя. При приеме для неслужебной части сообщения контрольная сумма считается заново и сравнивается - соответствует или не соответствует ли она пришедшему значению. Если данные не повреждены, то есть контрольные суммы совпадают, то подтверждается правильность их получения. Если иначе, то просто игнорируется то, что сообщение было послано. Как правило, это будет означать, что отправитель рано или поздно повторит сообщение. Вместо контрольной суммы можно использовать специальное кодирование передаваемых данных с помощью данных, исправляющих ошибки. Обычно это тоже некоторый циклический код, но он содержит большее количество бит, чем просто контрольная сумма, которая содержит избыточные данные. Тогда при повторном подсчете этого кода можно обнаружить и где было искажение данных, и как его исправить. Длина кода, возможно, растет линейно, в зависимости от того, сколько мы хотим ошибок контролировать и исправлять, как правило, рассчитывают на одиночные ошибки. Из практики известно, что это обнаружение двойных ошибок (то есть данные попорчены в двух местах, а контрольная сумма, может быть, даже сохранилась - такое кодирование это распознает), а также исправление одиночных ошибок, которые находятся и исправляются. Для большего количества необходимы более длинные коды - это, прежде всего, конечно, код Хэмминга, который является сугубо популярной

вещью. Есть некая смычка повышения надежности и криптографии, потому что тоже кодирование с исправлением ошибок используется и в криптографии. И там, и там достаточно часто используют хеширование.

Если по прошествии некоторого интервала времени со стороны получателя не приходит подтверждение принятия информации, то отправитель считает информацию утерянной и посылает её повторно. Вопрос состоит в том, как избежать повторного получения одних и тех же данных на стороне получателя. Для этого на приемном конце линии связи должен производиться соответствующий контроль. Например, для того, чтобы гарантировать правильный порядок получения сообщений - можно их нумеровать. Получатель знает, какое следующее по номеру сообщение должно к нему прийти, если он получает сообщение с номером, которое не соответствует ожидаемому, то он считает, что это сообщение ложное, то есть обращается с ним как с утерянным и продолжает ждать сообщение с правильным номером. Подобные действия могут быть возложены:

- **На операционную систему**, например, в протоколе TCP, где могут быть длинные сообщения. Они разбиваются на маленькие дейтаграммы, все дейтаграммы нумеруются по порядку, на каждую дейтаграмму на передачу приходит подтверждение со стороны получателя. Если на какую-то дейтаграмму получение не приходит, то отправитель, соответственно, повторяет её посылку. Если получатель второй раз получает дейтаграмму с одним и тем же номером, то просто игнорирует её. Это значит, что они где-то рассинхронизовались и ему пришла лишняя дейтаграмма.
- **На процессы, обменивающиеся данными**. Если они обмениваются данными через протокол UDP (User Datagram Protocol), гле пользовательская единица - это дейтограммы, то в этом случае вполне разумно, чтобы сами процессы, которые обмениваются данными, считали кусочки передаваемых длинных сообщений, чтобы они сами делали ту же самую схему.
- **Совместно на систему и процессы**, разделяя их ответственность. Например, операционная система может обнаруживать ошибки при передаче данных, а процесс принимать решение о том, что делать дальше

Как завершается связь?

Здесь можно выделить два аспекта:

- требуются ли от процесса какие-либо специальные действия по прекращению использования соответствующего средства коммуникации;
- влияет ли такое прекращение на поведение других процессов.

Для способов связи без инициализирующих действий - ничего специального для окончания взаимодействия делать не надо. Для сигналов в этом нет необходимости, для

линии связи с прямой адресацией тоже ничего делать не надо. Если же установление связи требовало некоторой инициализации, то, как правило, при ее завершении бывает необходимо выполнить ряд операций, например, сообщить операционной системе об освобождении выделенного связного ресурса.

Если кооперативные/взаимодействующие процессы перестают действовать согласованно, то это никак не влияет на их дальнейшее поведение. Если же какой-либо из этих процессов, которые ещё не завершили общение, находится в этот момент в состоянии ожидания получения данных, либо попадает в такое состояние позже, то операционная система должна предпринять некоторые действия, для того чтобы исключить вечное блокирование этого процесса. Обычно это извещение о том, что связи больше нет, например, с помощью сигнала. Это большая проблема, с которой приходится бороться, а это очень непросто. Мы установили связь между двумя процессами, которые работают по одной прямой линии, никто не вмешивается. Такая картинка бывает, например, когда есть связь между процессом, который хочет писать на диск, и процессом, который обслуживает драйвер диска: драйвер ждет, когда ему поступит следующее задание от этого процесса, а процесс умер (ему сделали kill, что сделать легко, потому что он не посылает никаких сообщений), остается вопрос - что делать с оставшимся драйвером, которому необходимо каким-то образом узнать, что ему никто больше ничего не пришлет по этой линии связи? Это очень непросто, потому что легко сказать - с помощью сигнала, но мы помним, что сигнала всего два. Необходимо, чтобы операционная система каким-то образом отследила, что этот процесс не просто так умирает, что при этом его, возможно, ждут. Необходимо, чтобы система каким-то образом (желательно до убиения процесса) послала по этой линии вместо него какое-то сообщение о том, что ждать больше не нужно. Вероятно, с аварийными завершениями это самое сложное.

Нити исполнения/ threads

В заключение поговорим про нити исполнения (threads), почему при программировании к ним необходимо относиться с осторожностью. Возможность продолжать выполнять программу в другом процессе, если в основном процессе задан обмен с внешним устройством - позволяет уменьшить среднее время ожидания результатов работы процессов.

Любой, отдельно взятый процесс в мультипрограммной системе никогда не может быть выполнен быстрее, чем при выполнении в однопрограммном режиме в той же вычислительной системе. Но, если режим решения задачи обладает определенным внутренним параллелизмом, то его можно ускорить, организуя взаимодействие нескольких процессов. Рассмотрим пример, пусть у нас есть следующая программа:

Ввести массив a
Ввести массив b
Ввести массив c

Выполнить векторную операцию a = a + b
Прибавить массив c = a + c
Вывести результирующий массив c

Легко увидеть, что если такая программа выполняется в рамках одного процесса, то он будет 4 раза блокироваться, ожидая окончания операций ввода-вывода:

Ввести массив a	
Ожидание окончания операции ввода	
Ввести массив b	
Ожидание окончания операции ввода	
Ввести массив c	
Ожидание окончания операции ввода	a = a + b
c = a + c	
Вывести результирующий массив c	
Ожидание окончания операции ввода	

Но алгоритм обладает внутренним параллелизмом, то есть вычисление суммы массивов a + b можно было бы выполнять параллельно с ожиданием завершения операции ввода массива c. Такое совмещение операций по времени можно было бы реализовать, используя два взаимодействующих процесса. Для простоты будем полагать, что самым дешевым средством коммуникации между ними служит разделяемая память. Тогда наши процессы могут выглядеть следующим образом:

Процесс 1	Процесс 2
Ввести массив a	Ожидание ввода
Ожидание окончания операции ввода	массивов a и b
Ввести массив b	
Ожидание окончания операции ввода	
Ввести массив c	
Ожидание окончания операции ввода	a = a + b
c = a + c	
Вывести результирующий массив c	
Ожидание окончания операции ввода	

Казалось бы, мы предложили конкретный способ ускорения решения задачи, но в действительности всё не так просто, потому что второй процесс необходимо создан, оба процесса должны сообщить операционной системе, что им необходим общий разделяемый сегмент памяти. И, наконец, нельзя забывать о переключении контекста, то есть реальное поведение процессов будет примерно таким:

Процесс 1		Процесс 2
Создать процесс 2		
Переключение контекста		

		Выделение общей памяти
		Ожидание ввода a и b
Переключение контекста		
Выделение общей памяти		
Ввести массив a		
Ожидание окончания операции ввода		
Ввести массив b		
Ожидание окончания операции ввода		
Ввести массив c		
Ожидание окончания операции ввода		
Переключение контекста		
		a = a + b
Переключение контекста		
c = a + c		
Вывести результирующий массив c		
Ожидание окончания операции ввода		

При таком выполнении программы можно не только не выиграть во времени при решении задачи, но и проиграть, поскольку накладные расходы на создание процесса, выделение общей памяти и переключение контекста могут превысить выигрыш, который получен за счет совмещения операций.

Для того, чтобы это можно было сделать поэкономнее, введем новую абстракцию внутри понятия "процесс" - **нить исполнения (thread)**. Сам термин thread появился в самом начале 90-х годов прошлого века. До этого соответствующие понятия назывались легковесными процессами/ light-weight process.

- Heavy process - это процесс со своей виртуальной памятью.
- Light-weight process - это процесс, который работает на общей виртуальной памяти с другими легковесными процессами.

Threads или потоки управлений, или нити исполнения процесса совместно используют его программный код, глобальные переменные (те, которые существуют в сегменте данных) и системные ресурсы. Но у каждой нити имеется свой собственный счетчик команд, свое содержимое регистров и свой стек. Теперь процесс представляется как совокупность взаимодействующих нитей и выделенных ему ресурсов. Тогда процесс - это совокупность взаимодействующих нитей и выделенных ему ресурсов. Процесс с одной нитью выполнения в точности совпадает с понятием процесса, которое мы использовали ранее. В этом курсе мы будем использовать термин **"традиционный процесс"**, хотя временами будем употреблять термин "тяжелый процесс". Иногда нити называют легковесными или облегченными процессами, так как во многих отношениях они подобны традиционным процессам. Нити, как и процессы могут порождать нити-потомки (правда, только внутри своего процесса) и переходить из одного состояния в другое. Состояния нитей аналогичны состояниям традиционных

процессов: нити исполняются, могут быть готовыми к исполнению, могут быть блокированными, могут порождаться и заканчиваться, и т.д.

- Из состояния "рождение" процесс приходит содержащим всего одну нить исполнения, все другие нити процесса будут являться потомками этой нити.
- Процесс находится в состоянии "готовность", если хотя бы одна из его нитей находится в состоянии "готовность" (есть кому занять процессор), и ни одна из нитей не находится в состоянии "исполнение".
- Процесс находится в состоянии "исполнение", когда одна или больше из его нитей находится в состоянии "исполнение".
- Процесс находится в состоянии "ожидание", если все его нити находятся в состоянии "ожидание".
- Процесс находится в состоянии "закончил исполнение", если все его нити находятся в состоянии "завершили исполнение".

Поскольку они работают асинхронно, если одна нить процесса блокируется, то другая нить того же процесса может прекрасно в это время выполняться на процессоре. Нити разделяют процессор при совместном его использовании точно так же, как это делали традиционные процессы в соответствии с теми алгоритмами планирования, о которых мы говорили ранее. Эти алгоритмы работают и на multithread систему тоже. Поскольку нити одного процесса разделяют существенно больше ресурсов, чем различные процессы, то операции создания новой нити и переключения контекста между нитями одного процесса - занимают существенно меньше времени, чем аналогичные операции для процессов в целом. Немного порассуждаем, за счет чего это происходит: во-первых, конечно же, не надо менять виртуальную память - это тяжелая смена контекста, регистры переключать надо, счетчик команд переключать надо. Очень плохо, когда один тяжелый процесс меняется на процессоре на другой: необходимо сбрасывать кэш, потому что он недействителен, там поиск идет по виртуальным адресам. То есть нам необходимо его полностью сбросить и заново набрать для процесса, у него возникает довольно тяжелое время раскрутки. Что происходит с threads кэша, который является небольшим? Высока вероятность того, что при смене внутри одной виртуальной памяти одного thread на другой, аппаратура в действительности будет заново выгружать кэш, потому что там будут другие адреса команд и другие адреса по обращению к данным. Насколько будут большими накладные расходы на переключение контекста - ещё необходимо посмотреть, так как это сильно зависит от того, что за программа и насколько там близко адреса в разных threads. Все не так просто, все требуется считать.

Предложенная схема совмещения работы в терминах нитей одного процесса получает право на существование. Но в данной схеме не хватает синхронизации, потому что, предположим, при "ожидание ввода a и b" можно без нее обойтись, а в "a =

a + b" - никак не обойтись, этот процесс должен каким-то образом сказать первому процессу, что a уже посчитано, иначе они будут работать неправильно. В этой схеме необходимо кое-что добавить, но мы пока этого не делаем, потому что мы пока ещё не говорили про синхронизацию.

Нить 1		Нить 2
Создать нить 2		
Переключение контекста нитей		
		Ожидание ввода a и b
Переключение контекста нитей		
Ввести массив a		
Ожидание окончания операции ввода		
Ввести массив b		
Ожидание окончания операции ввода		
Ввести массив c		
Ожидание окончания операции ввода		
Переключение контекста нитей		
		a = a + b
Переключение контекста нитей		
c = a + c		
Вывести результирующий массив c		
Ожидание окончания операции ввода		

Когда я участвовал в разработке своей первой операционной системы, то это была операционная система для нового компьютера **АС-6** - вычислительного комплекса, который делался специально для поддержки управления полета космической экспедиции "Союз-Аполлон" в 1978 году. Эта операционная система была, с одной стороны, очень интересной, с другой стороны - мы тогда слишком доверяли операционной системе **Multics** (очень советую обратить внимание на сайт https://www.multicians.org/). Это великая операционная система, она делалась с 1969 года, а окончательно её забросили только в 2010 году. Multics делалась тогда, когда операционные системы такого уровня были миру совершенно не нужны, в ней было множество интересных идей. Команда, которая делала Unix в AT&T, до этого участвовала в проекте Multics. Multics был проектом, который делался консорциумом, то есть кроме AT&T ещё участвовало ряд университетов и пр. Поскольку проект оказался очень тяжелым (он выполнялся 5 лет) и было понятно, что в таком объеме операционную систему сейчас не сделать, то его перестали финансировать. Молодые специалисты, которые работали в AT&T, через некоторое время уговорили начальство, что они не будут больше делать Multics, не будут делать такие сложные вещи, а сделают простенькую систему.

Unix - это UNiplexed система, Multics - multiplexed, она изначально была задумана как многопользовательская, многофункциональная. Unix не просто так

назвали Uniplexed Information and Computing Service - это несколько уничижительное название. Интересно было потом наблюдать за Unix, в который много лет понемногу затаскивались идеи Multics. Он становился все больше и больше на него похожим, наверное, только в последние годы Линус Торвальдс внедряет свои идеи в ядро. Долгое время и Linux тоже развивался по тому пути, что был у Multics. В 70-е годы мы делали операционные системы практически целиком на идеях Multics, одна из которых, в частности - использование сегментированной виртуальной памяти (это хорошо, когда она сегментирована). Причем таким образом, что операционная система находилась в отдельных сегментах каждого из процессов, то есть она как бы виртуально дублировалась, но в действительности была единая. В этом случае те же системные вызовы, например, были не прерываниями, а просто специальными переходами, специальными защищенными вызовами подпрограмм. В этой операционной системе программирование на уровне легковесных процессов было совершенно естественным, его было множество, соответственно, синхронизации тоже, потому что, когда много легковесных процессов обращается к одним и тем же данным - необходимо все время делать синхронизацию, чтобы они друг другу не навредили. В то время поиски ошибок в программах, которые были написаны в расчете на использование легковесных процессов, заключались в том, что проблемы этих ошибок (которые, конечно, объективно существуют) - это получить ситуацию, когда эта ошибка проявляется. А это чрезвычайно трудно, они возникают, потом пропадают и могут не проявляться в течении года. Отлаживаться и находить такие ошибки очень проблематично - их невозможно найти с помощью дампов, с помощью отладчика. Можно найти только путем чтения кода и, может быть, распечатки стеков (как программы себя ввели в этих процессах). В действительности произвольное программирование с использованием общей памяти и явной синхронизацией - это вещь, которая чрезвычайно опасная для программистов, поэтому, если кого-нибудь привлекает программирование с помощью threads - необходимо думать, как это сделать более безопасным.

Threads и их поддержка в операционных системах появились в начале 90-х годов, первой была операционная система **Solaris** компании Sun Microsystems. Компании потребовалось перелопатить внутри ядра множество кода, они сделали две вещи: это threads на уровне ядра и threads на уровне библиотек. Threads на уровне библиотеки означает почти тоже самое, что и overlays, то есть в этом случае в одном процессе есть монитор, который выполняет операции "образовать новый поток", никакого потока заново не создается, у него есть веточки кода, которые соответствуют разным потокам, он сам переключается с одной веточки на другую по своим собственным критериям. То есть - это реально одна последовательная программа с точки зрения операционной системы, а внешне это выглядит как multithread программа, которая себе подобным образом. Отладить программу на библиотеке threads ничего не стоит, там все детерминировано, все всегда повторяется, когда мы переходим на уровень ядра, то появляется множество ошибок, которые мы раньше просто не могли найти. Treads - это, к сожалению, вещь неизбежная, потому что мы не знаем, как по

другому использовать многоядерные процессоры. Использовать их необходимо аккуратно, один из аккуратных способ использования threads сделан в **Oracle** в сервере, который рассчитан на симметричные мультипроцессорные архитектуры. Там ровно по одному thread на процессор (не больше и не меньше), каждый thread обрабатывает кусочки запросов SQL и работает только с теми частями буферной памяти, кеша, который поддерживается в основной памяти, которые необходимы этому кусочку. Никакой общей памяти они не использую, но она аппаратно доступна. Это можно сделать в Oracle, потому что там один код СУБД и планы запроса генерируются самой СУБД.

Все сказанное выше справедливо для операционных систем, поддерживающих нити на уровне библиотек пользователей, и планирование процессора, и управление системными ресурсами осуществляются в терминах процессов. Распределение использования процессора по нитям в рамках выделенного процессу временного интервала осуществляется средствами библиотеки. В подобных системах блокирование одной нити приводит к блокированию всего процесса, ибо ядро операционной системы не имеет представления о существовании нитей. По сути дела, в таких вычислительных системах просто имитируется наличие нитей исполнения.

Заключение:

- Для достижения поставленной цели различные процессы могут исполняться **псевдопараллельно** на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой.
- Причинами для совместной деятельности процессов обычно являются: **необходимость ускорения** решения задачи, **совместное использование** обновляемых данных, **удобство работы** или **модульный принцип** построения программных комплексов.
- Процессы, которые влияют на поведение друг друга путем обмена информацией, называют **кооперативными или взаимодействующими** процессами.
- Для обеспечения корректного обмена информацией операционная система должна предоставить процессам специальные **средства связи**.
- По объему передаваемой информации и степени возможного воздействия на поведение процесса, получившего информацию, средства операционной системы можно разделить на три категории: **сигнальные, канальные и разделяемую память.**
- Через канальные средства коммуникации информация может передаваться в виде **потока данных** или в виде **сообщений** и накапливаться в **буфере** определенного размера.

- Для **инициализации** общения процессов и его прекращения могут потребоваться специальные действия со стороны операционной системы.
- Процессы, связываясь друг с другом, могут использовать прямую **симметричную, прямую асимметричную и непрямую** схемы адресации.
- Существуют **одно- и двунаправленные** средства передачи информации.
- Средства коммуникации обеспечивают надежную связь, если при общении процессов не происходит **потеря или повреждение** информации, не появляется **лишняя** информация, не нарушается **порядок** данных.
- Усилия, направленные на ускорение решения задач в рамках классических операционных систем, привели к появлению новой абстракции внутри понятия "процесс" – **нити исполнения (treads).**
- Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но **каждая нить** имеет собственный **программный счетчик**, свое **содержимое регистров** и свой собственный **стек**.
- Процесс представляется как **совокупность взаимодействующих нитей** и выделенных ему ресурсов.
- Нити могут порождать **новые нити** внутри своего процесса
- Нити имеют состояния, аналогичные состояниям процесса, и могут переводиться операционной системой из одного состояния в другое.
- В системах, поддерживающих **нити на уровне ядра**, **планирование** использования процессора осуществляется в **терминах нитей** исполнения, а **управление остальными системными ресурсами – в терминах процессов.**
- **Накладные расходы** на создание новой нити и на переключение контекста между нитями одного процесса **существенно меньше**, чем на те же самые действия для процессов
- Это **позволяет** на однопроцессорной вычислительной системе **ускорять решение задач** с помощью организации работы нескольких взаимодействующих нитей.

Алгоритмы синхронизации

Мы будем рассматривать синхронизацию и координацию процессов, рассмотрим основные понятия и программные алгоритмы организации взаимодействия процессов: как можно обеспечить синхронизацию процессов без использования операционной системы при некоторых предположениях о том, как работает аппаратура. Рассмотрим чередование, состязания, взаимное исключение, критическую секцию, аппаратную поддержку взаимных исключений. Ранее мы говорили о внешних

проблемах кооперации, связанных с организацией взаимодействия процессов со стороны операционной системы, предположим, что надежная связь процессов организована с помощью какого-то механизма, и процессы умеют обмениваться информацией. Необходимо ли что-то еще делать для того, чтобы процессы взаимодействовали правильно? Нужно ли менять их внутреннее поведение? Разъяснению этих вопросов и посвящена данная лекция.

Чередование, состязания, взаимное исключение

Давайте временно отвлечемся от операционных систем, процессов и нитей исполнения и поговорим о некоторых "активностях". Под активностями мы будем понимать последовательное выполнение некоторых действий, которые направлены на достижение некоторой цели. Активности могут иметь место в программном и аппаратном обеспечении/software и hardware, в обычной деятельности людей и животных. Мы будем разбивать активности на некоторые неделимые или атомарные операции. Например, активность "приготовление бутерброда" можно разбить на следующие последовательные атомарные операции:

- Нужно отрезать ломтик хлеба.
- Нужно отрезать ломтик колбасы.
- Нужно намазать ломтик хлеба маслом.
- Положить ломтик колбасы на подготовленный ломтик хлеба.

Конечно, эти действия, хотя мы их можем считать атомарными, могут иметь некоторые внутренние действия, от которых мы абстрагируемся. Мы называем **операцию неделимой или атомарной** потому, что считаем её одним целым, которое выполняется за один раз, без прерывания деятельности. Может быть, например, так: мы приготовились к тому, чтобы отрезать кусок хлеба, взяли батон, взяли нож, а оказалось, что нож совершенно тупой. Тогда в этом действии придется ввести некоторое под-действие, например, поиск острого ножа.

Пусть имеется две активности:

P: a b c и Q: d e f
(где a, b, c, d, e, f - атомарные операции)

Если эти активности выполняются чисто последовательно, то мы получаем следующую последовательность атомарных действий:

PQ: a b c d e f

Что будет происходить, если эти действия выполняются concurrently/ псевдопараллельно в режиме разделения времени? Активности могут расслоиться на неделимые операции с различным их чередованием, если действительно каким-то образом происходит разделение времени, то есть может произойти то, что на английском языке принято называть словом **interleaving/чередование операций.**

Если у нас есть действительно атомарные операции: a b c d e f, то возможны следующие варианты их чередования:

a b c d e f
a b d c e f
a b d e c f
a b d e f c
a d b c e f
.....
d e f a b c

Мы сохраняем порядок внутри активностей атомарных операций, а между собой они могут чередоваться. Поскольку псевдопараллельное выполнение активностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения.

Рассмотрим пример: пусть у нас имеется две активности P и Q, состоящие из двух атомарных операций каждая:

P: (x=2) (y=x-1) Q: (x=3) (y=x+1)

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются общими для активностей? Можно убедиться, что возможны четыре разных набора значений для пары x, y:

(x, y): (3, 4), (2, 1), (2, 3) и (3, 2).

Набор активностей (например, программ) называется **детерминированным,** если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он **не детерминирован**. Предыдущий пример - это пример недетерминированного набора программ из двух таких программ. Детерминированный набор активностей можно спокойно, безопасно выполнять в режиме разделения времени. Для недетерминированного набора такое исполнение нежелательно, потому что непонятно, что мы будем получать. Можно ли заранее определить, является ли набор активностей детерминированным или, может быть, необязательно? Для этого существуют **достаточные условия Бернстайна**. Это тот самый Филипп Бернстайн, который написал книгу "Операционные системы" вместе с Д. Цикритзисом.

Мы рассмотрим эти условия применительно к программам с разделяемыми переменными. Соответственно, мы введем понятия **"наборы входных и выходных переменных программы"**. Для каждой атомарной операции наборы входных и выходных программ - это наборы переменных, которые атомарная операция считывает и записывает, которые находятся в правой части операции "присваивание", а также те, которые находятся в левой части операции "присваивание".

- Набор входных переменных программы R(P) (P - это programm, R - от слова read) - это объединение наборов входных переменных для всех ее неделимых действий.

- Набор выходных переменных программы W(P) (W от слова write) - это объединение наборов выходных переменных для всех ее неделимых действий.

Например, для программы: P: (x=u+v) (y=x*w)
получаем R(P) = {u, v, x, w}, W(P) = {x, y}

Заметим, что переменная x присутствует как в R(P), так и в W(P).

Условия Бернстайна звучат так: если для двух данных активностей P и Q:

пересечение W(P) и W(Q) пусто,
пересечение W(P) с R(Q) пусто,
пересечение R(P) и W(Q) пусто,
тогда выполнение P и Q детерминировано.

Тоже самое условие требуется в том случае, когда в системе управления базами данных с таким чередованием происходит выполнение двух транзакций, и мы хотим гарантировать, что эти транзакции никогда не дадут нехороших эффектов. Тогда требуется, чтобы эти транзакции не конфликтовали по записи, то есть чтобы они не писали в один и тот же элемент баз данных; чтобы при транзакции, которая изменяет какой-то объект базы данных, другая транзакция одновременно не считала этот объект. И наоборот - если первая транзакция его считает, то вторая не должна его изменять. То есть условия Бернстайна - это общие условия для детерминированности, которые по-разному называются в разных случаях. Если эти условия не соблюдены, то легко доказать, что квазипараллельное выполнение P и Q будет детерминировано, а если не соблюдены, то - как получится, это достаточное условие, то есть оно не является необходимым - они могут быть детерминированными, а могут и не быть. Это условие обобщается на большее число активностей, что очень легко доказывается от противного: предположим, что выполнение недетерминировано, это означает, что при разном выполнении мы получим разные результаты выходных переменных.

Условия Бернстайна информативны, но очень жестки для того, чтобы процессы выполнялись - практически требуются, чтобы не взаимодействовали процессов. Действительно, если они ничего другого не меняют, то непонятно, как они могут взаимодействовать. Мы хотим, чтобы детерминированный набор образовывали активности, которые совместно используют данные и обмениваются ими. Для этого необходимо уметь ограничить число возможных чередований атомарных операций, исключая некоторые чередования с помощью механизмов синхронизации выполнения программ и обеспечивая упорядоченный доступ программ к некоторым данным.

Про недетерминированный набор программ (и активностей вообще) говорят, что он находится в состоянии состязания - **race condition**, иногда это состояние называют гонки. В примере, который мы рассматривали, где имеется недетерминированный набор из двух программ, эти процессы состязаются за то, чтобы вычислить выходные значения переменных x и y, кто последний присвоит - тот и победил.

Задачу упорядоченного доступа к разделяемым данным, если не важна его очередность к ним, можно решить, если обеспечить каждому процессу монопольное право доступа к этим данным (когда это разрешается делать только одному процессу). Каждый процесс, который обращается к совместно используемым ресурсам, в этом случае исключает для всех других процессов возможность одновременного с ним общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется взаимоисключением - **mutual exclusion.** Это работает, если очередность доступа к разделяемым ресурсам не важна, если она требуется, чтобы эти доступы выполнялись именно в каком-то порядке, то одними взаимоисключениями уже не обойтись.

Критическая секция

Ещё одно важное понятие - это критический участок или критическая секция/critical section. **Критическая секция** - это часть программы, выполнение которой может привести к возникновению состояния состязания/race condition. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, нужно уметь организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом. То есть необходимо обеспечить реализацию взаимного исключения для критических участков программ. Реализация взаимного исключения для критических участков программ с практической точки зрения означает, что по отношению к другим процессам, которые участвуют во взаимодействии, критический участок начинает выполняться как атомарная операция, они просто не могут вклиниться в середину. Давайте рассмотрим пример, в котором псевдопараллельные взаимодействующие процессы представлены действиями различных студентов, которые живут в одной комнате в общежитии:

Время	Студент 1	Студент 2	Студент 3
17-05	Приходит в комнату		
17-07	Обнаруживает, что хлеба нет		
17-09	Уходит в магазин		
17-11		Приходит в комнату	
17-13		Обнаруживает, что хлеба нет	
17-15		Уходит в магазин	
17-17			Приходит в комнату
17-19			Обнаруживает, что хлеба

			нет
17-21			Уходит в магазин
17-23	Приходит в магазин		
17-25	Покупает 2 батона на всех		
17-27	Уходит из магазина		
17-29		Приходит в магазин	
17-31		Покупает 2 батона на всех	
17-33		Уходит из магазина	
17-35			Приходит в магазин
17-37			Покупает 2 батона на всех
17-39			Уходит из магазина
17-41	Возвращается в комнату		
17-47		Возвращается в комнату	
17-53			Возвращается в комнату

Здесь критический участок для каждого процесса от операции "обнаруживает, что хлеба нет" до операции "возвращается в комнату" включительно. В результате у студентов 6 батонов хлеба, то есть в результате отсутствия взаимоисключения мы из ситуации "хлеба нет" мы попадаем в ситуацию "хлеба слишком много". Если бы этот критический участок выполнялся как атомарная операция "добывает два батона хлеба", то проблема образования избытка хлеба была бы решена.

Время	Студент 1	Студент 2	Студент 3
17-05	Приходит в комнату		
17-07	Достает два батона хлеба		
17-43		Приходит в комнату	
17-47			Приходит в комнату

Первый студент приходит в комнату и атомарным образом где-то добывает 2 батона хлеба, потом приходит второй студент, после него – третий, и хлеба всем достаточно. Если говорить про общежитие, то самое разумное для первого студента – это перед тем, как пойти в магазин, закрыть дверь изнутри на ключ и уйти добывать хлеб через окно, тогда второй и третий студенты будут стоять у двери снаружи. Когда первый вернется с хлебом, то влезет в комнату через окно и откроет дверь.

Для решения задачи требуется, чтобы в том случае, когда процесс находится в своем критическом участке, другие процессы в свои критические участки войти не могли. То есть критический участок должен сопровождаться прологом **entry section** - "закрыть дверь изнутри на засов" (с этого момента начинается критический участок) и эпилогом **exit section** - "отодвинуть засов", эти действия не имеют отношения к самой активности одиночного процесса, они ограждающие. В прологе процесс должен получить разрешение на вход в критический участок, а в эпилоге сообщить другим

процессам, что процесс критический участок покинул. То есть в общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

```
while (some condition) {
      entry section
            critical section
       exit section
             remainder section }
```

Здесь под remainder section понимаются все атомарные операции, не входящие в критическую секцию. Дальше мы будем рассматривать разные способы программной организации пролога и эпилога критического участка в случае, когда очередность доступа к критическому участку не имеет значения.

Лекция 5. Алгоритмы и механизмы синхронизации

Прошлый раз мы начали рассматривать тему синхронизации и сегодня продолжим с повторения того, что наиболее важно.

- Первое наиболее важное понятие - это **чередование/ interleaving**, способ выполнения активностей с чередованием атомарных операций. Это то, что называют concurrently или псевдопараллельный режим, если речь идет про один процессор. В действительности понятие "чередование" обобщается на понятие **"реальное параллельное выполнение"**.

- Следующее важное понятие - это **детерминированность.** Если параллельно или псевдопараллельно выполняется некоторый набор активностей (например, программ), то выполнение называется детерминированным, если при любом способе чередования или псевдопараллельного выполнения результат выполнения один и тот же. В противном случае он **недетерминирован**.

- Очень важная вещь - **достаточные условия Бернстайна**, когда две активности выполняются всегда детерминированным образом, как бы не чередовались их атомарные операции. В действительности - это фактически **требования отсутствия конфликтов** между этими активностями. Конфликты бывают, когда они пытаются писать в одни и те же переменные, когда одна активность читает, а другая пишет в ту же переменную. Или наоборот - когда одна пишет, другая читает. Это важно с точки зрения обеспечения разумного выполнения взаимодействующих активностей. Видно, что они детерминированным образом выполняются без всяких дополнительных усилий, когда они практически не взаимодействуют, потому что если никто из них не может писать туда, откуда другая активность читает, то у них фактически нет никакого способа взаимодействия. Это важно, потому что в действительности аналогичные проблемы возникают и в других областях, в частности в областях управления базами данных. Мы можем в чистом виде почти то же самое там прочитать, только почему-то это никто не называет условиями Бернстайна. Это называется отсутствие конфликтов между транзакциями. Фактически задача планирования транзакций в системах управления базами данных - как раз добиться устранения такого рода конфликтов. Нам необходимо, чтобы все это не так жестко требовалось со стороны взаимодействующих активностей, а для этого нужно иметь ограниченное число возможных чередований атомарных операций с тем, чтобы остались такие возможные чередования, которые обладают свойствами детерминированности. Как раз для этого и существуют механизмы синхронизации, они обеспечивают упорядоченный доступ программ к некоторым данным. Фактически приходится бороться с тем, что называют состоянием состязания или гонки. Это и есть тот случай, когда два процесса или

две активности пытаются состязаться - кто из них последним запишет какую-то переменную. Кто последним запишет - то состояние и останется у программы.

- Необходимо обеспечить какой-то порядок доступа к разделяемым данным, если не важна его очередность к ним. Потому что если очередность важна, то просто синхронизацией не обойдешься, тогда необходимо координировать выполнение программ специальным образом. Для того, чтобы было можно решить задачу доступа к совместно используемым участкам памяти - необходимо обеспечить процессам монопольное право доступа к этим данным. Вокруг этого важное понятие - **взаимное исключение/mutual exclusion**. Если можно обеспечить такой механизм, который не дает процессам одновременно находиться в том участке, который приводит к конфликтам, который фактически нарушает условия Бернстайна, тогда надо этим воспользоваться.

- Последнее важное понятие - **это критический участок**. Он устроен следующим образом: фактически необходим кусок специального кода, который принято называть прологом от критического участка - entry section, который как раз и обеспечивает взаимные исключения на входе. Необходим и конечный участок - эпилог/exit section, который дает возможность войти в критический участок другому процессу. Тое есть в общем случае структура процесса, который участвует в такого рода взаимодействии, выглядит следующим образом:

```
while (some condition) {
      entry section
            critical section
       exit section
             remainder section }
```

То есть критический участок ограждается прологом и эпилогом, вне него выполняется remainder section - атомарные операции, которые не входят в критический участок. Процесс крутится по некоторому условию, пока оно придерживается.

Дальше мы в этой лекции рассмотрим, как можно организовать взаимные исключения в программах, не прибегая к поддержке со стороны операционных систем. Оказывается, это можно сделать чисто программным образом. То, что мы будем рассматривать, ни в коем случае не означает, что так делать необходимо. В действительности, как правило, делать так как раз не нужно, а лучше пользоваться средствами операционных систем, про которые мы будем говорить в следующих лекциях. Но вот такие, чисто программные средства, во-первых, помогают понять, как все-таки реально можно это сделать, помогают понять - что же делает операционная система, во-вторых - есть некоторые подозрения, что не исключено, что не в очень далеком будущем, когда мы будем иметь дело с многоядерными процессорами (в которых число ядер будет доходить до 1 000), похоже, что это будет даже в обозримом будущем. Там накладные расходы, которые вызывают обращения к операционным

системам для синхронизации выполняемых процессов - слишком дорого стоят, то есть не исключено, что именно там придется делать что-нибудь в подобном роде. В лучшем случае, конечно, не исключено, что кому-нибудь удастся придумать какую-нибудь дополнительную аппаратную поддержку, но пока этого не видно.

Программные алгоритмы организации взаимодействия процессов

Требования, предъявляемые к алгоритмам

Организация взаимного исключения для образования критических участков - позволяет избегать возникновения состязаний, но не является достаточной для правильной и эффективной параллельной работы взаимодействующих процессов. Есть **пять условий**, принято считать, что они должны выполняться для разумного программного алгоритма организации взаимодействия процессов с критическими участками, если они могут их проходить в произвольном порядке, то есть если порядок здесь не существенен. Мы увидим далее на примерах, что в действительности имеется много практических приложений, когда порядок действительно не важен, то есть не важно, пройдет ли первый процесс первым (или второй, или третий) - главное, чтобы не одновременно.

- **Требование 1:** Задача организации взаимного исключения должна решаться чисто программным способом на обычном компьютере, у которого нет специальных команд для поддержки взаимного исключения. Предполагается, что основные операции языка программирования (**load** - чтение из памяти на регистр, **store** - запись из памяти в регистр, **test** - проверка условия), для того чтобы выработать условия для условного перехода - **являются атомарными операциями**. Это, наверное, в компьютерах соблюдается, если их разрешить прерывать, то семантика программ будет не очень понятна.

- **Требование 2:** В алгоритмах не должны использоваться какие-либо предположения об относительных скоростях выполняющихся процессов или о числе процессоров, на которых они выполняются. Это не должно влиять на работу алгоритмов.

- **Требование 3:** Это **условие взаимного исключения**, если процесс P_i выполняется в своем критическом участке, то не должны существовать не одновременно какие-либо другие процессы, которые выполняются в своих соответствующих критических участках, то есть должно обеспечиваться условие взаимного исключения (**mutual exclusion)**.

- **Требование 4:** Процессы, которые находятся вне своих критических участков и не собираются входить в них, то есть никаких действий по этому поводу не предпринимают - не должны мешать другим процессам входить в их собственные критические участки. Если нет процессов в критических участка, там вообще никто не работает, и имеются процессы, которые туда хотят войти,

то только те процессы, которые не выполняются в **remainder section (**там, где уже не нужно взаимное исключение), должны принимать решение о том, какой процесс войдет в свой критический участок. Такое условие должно приниматься за конечное время. Это условие получило название **"условие прогресса" - progress**. Процессы, которые не конкурируют, - не должны никому мешать, а конкурирующие процессы, естественно, друг другу мешают, на то они и взаимоисключают друг друга.

- **Требование 5:** Ни у какого процесса не должно возникать бесконечное ожидание входа в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критический участок, и до того момента, как он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие называется **условие ограниченного ожидания - bound waiting**. Есть такое заклинание: если в области Computer Sciences кто-нибудь пытается нам предложить какое-то решение, рассчитанное на бесконечность (на любую бесконечность, включая счетное множество - самое простое, самое маленькое множество), то его не нужно слушать, потому что в действительности компьютерные науки имеют дело только с конечными множествами. Из-за того, что компьютеры сами по себе конечны, конечны адресации, конечна память, в конце концов, конечно и время работы, потому что ни один компьютер не может работать бесконечно. Кроме того, конечна и наша жизнь, а мы живем в реальном мире компьютерных наук - это не абстрактная математика, здесь всё должно быть предельно конкретно. Поэтому условие ограниченного ожидания несколько искусственное, потому что если мы это ограниченное число раз сделаем очень большим, то это будет означать, что какой-то процесс реально никогда не войдет в критический участок, потому что он просто не проживет столько времени. Это дань математичности этих алгоритмов, но мы видим, что они не очень математические. Есть математики серьезные, которые, как правило, не имеют отношения к Computer Sciences, они занимаются фундаментальной математикой. Может быть, когда-то их результаты для компьютеров пригождаются, но они не делаются специально для компьютерных наук, это приложение. Есть математики, которые находятся около компьютеров, их довольно много, их больше, чем настоящих чистых математиков. Эта математика довольно часто выглядит искусственной, потому что её усложняют для того, чтобы она была похожа на математику. С другой стороны, какие-то вещи бывают явно полезными, но они не имеют отношения к бесконечности. Например, в компьютерных науках очень помогает теория вероятности, хотя она тоже абстракция. В мире нет ничего, кроме математической статистики. Теория вероятности - это абстрактная наука, но поскольку формально от нее до математической статистики не очень далеко, то можно получать результаты, которые получаются за счет применения методов теории вероятности, а

использовать их применительно к тому, что мы работаем уже не с вероятностями, а с мерами достоверности, которые оцениваются статистически.

Запрет прерываний. Самый простой подход - это организация критических участков путем запрета прерывания. В этом случае пролог состоит в том, что в этом компьютере запрещаются все прерывания. Очевидно внешние прерывания, потому что внутренние прерывания запретить невозможно, если возникает условие исключительной ситуации, то оно возникает. Работает критический участок, после чего прерывания разрешаются и другие процессы могут сами закрыть прерывание и войти туда.

```
while (some condition) {
  запретить все прерывания
    critical section
  разрешить все прерывания
    remainder section }
```

Поскольку для любого процесса выйти из состояния исполнение, уйти с процессора без наступления прерывания - невозможно, то понятно, что внутри критического участка никто не сможет вмешаться в работу этого процесса. Но с предположениями, что там не виртуальной памяти, нет прерываний по отсутствию страницы виртуальной памяти, что можно работать без таймера, что его тоже можно заблокировать и т.д. Однако такое решение - опасно, последствия у него могут быть непредсказуемы, потому что фактически процессу пользователя в этом случае разрешается запрещать и разрешать прерывания во всем компьютере. Например, в этой программе, которая внутри критического участка зациклится, тогда никто не снимет этот процесс, потому что прерывания запрещены. В этом случае требуется перезагрузка всей машины, ещё их необходимо обнаружить. На БЭСМ - 6 всю работу можно было наблюдать на неонках, которые светились на передних панелях компьютеров. Там было видно всё, если какая-то программа зациклилась, то видно, что машина в странном состоянии. В нормальном состоянии у нее неонки дрожали, потому что там всё время были какие-то прерывания, всё время были обмены с внешними устройствами, ни один регистр не находился в спокойном состоянии, чтобы светиться всеми лампочками постоянно. Если вдруг возникала такая статическая картинка, то уже через 3 секунды было понятно, что какая-то программа зациклилась и, видимо, с блокировкой прерываний (и, скорее всего, это зацикливалась сама операционная система).

Для пользовательских процессов этого допускать нельзя, вообще **блокировка прерывания - это команда привилегированная**, её нельзя выполнять в современных компьютерах в пользовательском режиме. Однако внутри операционной системы, внутри ядра блокировка прерываний используется именно для организации некоторых критических участков на самом нижнем уровне. Например, в тех случаях, когда применение каких-либо высокоуровневых примитивов, функций операционной системы было бы слишком дорого. Как это можно сделать в наращиваемом режиме -

мы рассмотрим, когда будем говорить про средства, которые поддерживаются операционными системами для синхронизации процессов. То есть это - негодный подход для организации взаимного исключения на уровне пользовательских программ.

Переменная-замок (блок). Пускай есть некоторая переменная, которая доступна всем процессам, у нее начальное состояние - 0. Процесс может войти в критический участок только тогда, когда значение этой переменной-замка равно 0, одновременно он меняет ее состояние на 1, закрывая замок. При выходе из критического участка процесс просто сбрасывает значение в 0, тем самым он замок открывает. Shared означает, что переменная является разделяемой.

```
shared int lock = 0;
while (some condition) {
  while(lock); lock = 1;
    critical section
  lock = 0;
    remainder section }
```

На вид - это очень симпатичное решение, если, конечно же, не обращать внимание, что такое решение не удовлетворяет условию взаимоисключения, так как действие while(lock) и lock = 1; не является атомарным, то есть между ними может возникнуть прерывание и переключение процесса.

Пусть есть два процесса: P_0 и P_1, процесс P_0 вышел из цикла, где он крутился по состоянию 1 в переменной lock, тем самым он принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1 - пришло прерывание, которое вызвало то, что планировщик снял с процессора процесс P_0 и запустил там процесс P_1. Он тоже вошел в пролог, то есть из цикла while(lock) тут же вывалился, потому что переменная всё ещё равна 0, тем самым они оба находятся в точке lock = 1 (один выполняется, другой - готов). Кто из них будет выполняться первым - неизвестно, главное, что они точно не будут взаимно исключаться при входе в критический участок. То есть они именно одновременно выполняют свой критический участок, а если это двуядерная машина, то они и реально параллельно там будут работать. Не годится.

Строгое чередование. Теперь посмотрим, как можно это делать со строгим чередованием. Идея заключается в следующем: всегда проходит только один процесс, а следующий может пройти только за ним следом. Будем использовать общую переменную с начальным состоянием 0 для двух процессов P_0 и P_i. Она явно указывает, какой процесс может войти в критический участок. То есть, если есть всего два процесса - нулевой и первый, до для i-го процесса это выглядит так:

```
shared int turn = 0;
while (some condition) {
  while(turn != i);
```

```
    critical section
  turn = 1-i;
    remainder section }
```

Есть ещё переменная turn, которая и принимает значение 0 или 1. Процесс смотрит, чему равен turn. До тех пор, пока turn не равно 1 - он циклится, как только равно i - входит в критический участок, а потом пропускает другой процесс, присваивая 1-i. В turn != i процесс циклится, если он - первый процесс, то циклится пока там 0, когда там 1, то он оттуда вываливается, выполняет критический участок и пропускает первый процесс. Всё замечательно, то есть взаимное исключение гарантируется, процессы входят в критический участок по очереди: P_0 P_i, P_0 P_i, P_0 P_i, P_0 P_i,... Но условие прогресса не выполняется. Например, если значение turn равно 1, а процесс P_0 готов войти в критический участок, то он не может это сделать, даже если процесс P_i находится за пределами своего критического участка в remainder section. То есть тоже не годится. Это происходит из-за того, что процессы при строгом чередовании ничего не знают о том, какое состояние у партнера в текущий момент времени.

Флаги готовности. Давайте попробуем исправить эту ситуацию с незнанием процесса о состоянии партнера. Пусть есть два процесса P_0 и P_i, есть массив из двух флагов (вероятно, целочисленных) готовности входа процессов в критический участок:

```
shared int ready[2] = {0, 0};
```

Когда i-й процесс готов войти в критический участок, он присваивает своему элементу массива ready[i] значение 1. После выхода из критического участка он записывает значение в 0, потому что он готов, когда доходит до пролога. Ни один процесс не входит в критический участок, если другой процесс уже готов ко входу в него или находится в ней. Программа будет выглядеть следующим образом:

```
while (some condition) {
  ready[i] = 1;
  while(ready[1-i]);
    critical section
  ready[i] = 0;
    remainder section}
```

ready[i] = 1 и while(ready[1-i]) - это пролог, то есть i-й процесс - i 0 или 1, он говорит: я готов войти в критический участок, смотрит, что происходит с другим процессом 1-i, если он тоже готов войти в критический участок, то он крутится в while(ready[1-i]) до тех пор, пока 1-i процесс не скажет, что он уже больше не хочет туда входить.

Этот алгоритм обеспечивает взаимное исключение и позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога

в другом процессе, но условие прогресса тоже нарушается, к сожалению. Предположим, что оба процесса одновременно подходят к выполнению пролога. Тогда, после выполнения присваивания ready[0]=1, нулевой процесс говорит: я готов. По причине неатомарности этих операций планировщик запускает на процессоре процесс 1, который также выполнил присваивание ready[1]=1, он говорит: я готов. В результате на этом цикле оба процесса застревают навсегда, потому что случилось так, что они оба готовы войти. Они оба готовы из-за того, что ready[i] = 1 - здесь может быть переключение процессора с одного процесса на другой. То есть возникает ситуация, когда оба процесса циклятся, они бесконечно ждут один другого на входе в критический участок.

Это пример ситуации, которая называется **тупиком - deadlock**. Deadlock я всегда любил называть синхронизационным тупиком, 50 лет назад довольно часто для этого пытались использовать название клинч, по аналогии с боксерами, на которых это очень похоже: они сцепились друг с другом, но ни один из них не может стронуться, ни один не может ударить. Этот клинч - странный, потому что процессы не совсем сцепились - скорее оба стоят и ничего не решаются сделать, хотя ни один из них ничего не держит, они сами себя сдерживают (но так они друг друга ведут). Настоящие тупики устроены не так. По эффекту - это, конечно, тупик, так как фактически ни один процесс не может войти в критический участок. Забавно, что процессор при этом будет полностью занят, а оба процесса будут усиленно циклится. Подробнее о тупиковых ситуациях мы поговорим в лекции 7. Далее мы познакомимся с разными программными алгоритмами.

Алгоритм Петерсона. Это самый простой алгоритм и это первое решение проблемы, которое было вроде бы без ошибок (в действительности можно доказать, что ошибок нет), удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов. Оно было предложено датским математиком Деккером (Dekker). В 1981 году датчанин Петерсон (Peterson) предложил более изящное решение. Интересно, что в 70-80-е годы было подозрительно много скандинавов среди специалистов в операционных системах: Вибе Дейкстра, Петерсон, Деккер и прочие, не менее пяти очень известных экспертов.

Алгоритм выглядит следующим образом: есть и то, и другое - и флаги готовности и переменная, которая говорит, кому дальше можно входить. Пусть оба процесса имеют доступ и к массиву, и к переменной очередности. Рассмотри, как тогда устроен пролог:

```
shared int ready[2] = {0, 0};
shared int turn;
while (some condition) {
  ready[i] = 1;
  turn = 1-i;
  while(ready[1-i] && turn = 1-i);
```

```
    critical section
  ready[i] = 0;
    remainder section }
```

P_i процесс говорит, что он хочет войти и переворачивает значение переменной 0 или 1, показывая, что он разрешает войти другому процессу. Что проверяется при входе? Это обычная конъюнкция, то есть в (ready[1-i] && turn = 1-i), чтобы он вышел из while - ready должно быть 1-i. Процесс не хочет войти в критический участок, это будет 0, чтобы он предлагал войти партнеру войти в критический участок. Его партнер не должен быть готов, чтобы войти туда, либо его партнер предлагает войти данному процессу. Когда мы выходим, то P_i процесс говорит, что ему больше не нужно быть в критическом участке. Давайте посмотрим, почему это работает правильно.

При выполнении пролога критического участка P_i процесс здесь - ready[i] = 1 говорит, что он готов выполнить критический участок и предлагает другому процессу выполнить свой критический участок. Если два процесса одновременно подошли к while, когда всё проверятся, то они оба установят ready[i] = 1, то есть они оба готовы, оба предложат друг другу выполнять критический участок. Но при этом одно из предложений всегда следует после другого, поэтому работу в критическом участке продолжит тот процесс, которому последним было сделано предложение. Давайте докажем, что все пять наших требований к алгоритму Петерсона действительно удовлетворяются.

- **Удовлетворение требований 1 и 2 происходит автоматом**, то есть мы не использовали никаких специальных команд процессора, нет никаких предположений относительных скоростей процессов или аппаратуры.

- Докажем выполнение **условия взаимоисключения** методом от противного, что два процесса никогда не могут одновременно попасть в критический участок. Пусть оба процесса одновременно оказались внутри своих критических участков. Процесс P_i может войти в критический участок тогда и только тогда, когда ready[1-i] = 0 или turn = i (именно этого процесса). Так как здесь while(ready[1-i] && turn = 1-i) стоит конъюнкция, значит необходимо, чтобы либо ready[1-i] turn = 1-i было в 0, чтобы выйти из while. Заметим, что если бы два процесса одновременно выполняли свои критические участки, то значения флагов готовности у них совпадали бы и были равны 1, то есть и ready от 0 и ready от 1 - были бы равны 1. Могут ли эти два процесса войти в критические участки из состояния, когда они оба находятся в процессе выполнения цикла while? Нет, чтобы оба процесса вышли while переменная turn должна была бы одновременно иметь значения 0 и 1, чтобы 1-i стала равно 0. Если процесс P_0 первым вошел в критический участок, то процесс P_i должен был выполнить перед входом в цикл while хотя бы один предваряющий оператор (turn = 0;). Но после этого он уже не может выйти из цикла до окончания критического участка процесса P_0, так как при входе в цикл ready[0] = 1 и turn = 0, эти значения не

могут измениться до тех пор, пока процесс P_0 не выйдет из своего критического участка. То есть мы получаем противоречие, следовательно, здесь имеет место взаимное исключение.

- Докажем теперь, что удовлетворяется **условие прогресса**. Что за конечное число проходов другим процессом критического участка, процесс P_1 туда тоже войдет. Пускай это будет процесс P_0. Он не может войти в свой критический участок только тогда, когда одновременно совместно выполняются условия ready[1] = 1 и turn = 1, когда у него там две единицы - он циклится. Если процесс P_1 не готов к выполнению критического участка, то ready[1] = 0, и процесс P_0 может войти туда, соответственно, он выйдет из этого цикла while. Если процесс P_1 готов к выполнению критического участка, то ready[1] = 1, а переменная turn имеет значение 0 либо 1, позволяя либо процессу P_0, либо процессу P_1 начать выполнение критического участка. Если процесс P_1 вышел из критического участка, то он сбросит свой флаг готовности ready[1] = 0, тем самым он разрешит процессу P_0 выйти из своего цикла и выполнить критический участок. То есть в действительности не более, чем за один проход другим процессом критического участка, другой процесс получит возможность в него войти. Таким образом, условие прогресса выполняется.

- Отсюда же вытекает выполнение **условия ограниченного ожидания**. В действительности не просто конечное число раз, а ровно не более одного раза будет достаточно, чтобы войти в критический участок. Поскольку в процессе ожидания разрешения на вход процесс P_0 не изменяет значения переменных, он сможет начать исполнение своего критического участка после не более, чем одного прохода по критическому участку процесса P_1.

Алгоритм Петерсона - очень простой. Кажется странным, что он при этом работает правильно, но это как раз и есть заслуга Петерсона, потому что один из моих любимых героев в области баз данных - Джеймс Николас **"Джим" Грей** когда-то написал совершенно замечательную фразу: не применяйте сложные решения, сложные решения хорошо подходят для учебников и разнообразных, хитрых документаций, а на практике работают только простые решения. Это действительно так, важно уметь находить именно простые решения, так как они очень необходимы, потому что именно они и работают. Алгоритм Петерсона обеспечивает решение задачи корректной организации взаимодействия двух процессов. Давайте теперь рассмотрим соответствующий алгоритм взаимного исключения для n взаимодействующих процессов.

Алгоритм булочной (Bakery algorithm). Алгоритм получил название "алгоритм булочной" и это довольно странно, потому что он был придуман не в голодные послевоенное время. Алгоритм родом из Скандинавии, где в действительности уже в 50-е годы никаких проблем с этим не было. Это скорее похоже на алгоритм работы регистратуры в поликлинике, потому что поликлиника, врачи - это

ограниченный ресурс. Это понятно, потому что в поликлинике как раз необходимы взаимное исключение и точность.

Основная идея алгоритма выглядит следующим образом: каждый вновь прибывающий клиент (он в компьютере соответствует процессу) получает талончик с номером на обслуживание. Клиент, который имеет наименьший номер на талончике - обслуживается следующим. Из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов - первым обслуживается клиент с меньшим значением имени, а имена сравниваются в лексикографическом порядке (Петров больше, чем Кузнецов). Для этого алгоритма приходится поддерживать два массива совместно используемых данных:

shared enum {false, true} choosing[n]; - выбор очередного клиента
shared int number[n]; - номер талончика

Элементы этих массивов инициируются значениями false и 0 соответственно.

Введем следующие обозначения:

(a,b) < (c,d), если a < c
или если a = c и b < d
max(a0, a1, ..., an) – это число k такое, что
k >= a_i для всех i = 0,... ,n

Тогда структура процесса P_1 (процесс, который получил талончик с i номером) для алгоритма булочной ведет себя следующим образом:

```
while (some condition) {
  choosing[i] = true;
  number[i] = max(number[0],... , number[n-1]) + 1;
  choosing[i] = false;
  for(j = 0; j < n; j++){
  while(choosing[j]);
   while(number[j] != 0 && (number[j],j) < (number[i],i)); }
    critical section
  number[i] = 0;
    remainder section }
```

Процесс P_1 просит пропустить его, он говорит, что готов войти - choosing[i] = true, одновременно он выбирает максимальный номер из номеров от 0 до n-1 и прибавляет к нему ещё 1, после чего сбрасывает choosing[i] на false и просматривает все процессы от j = 0 до j < n, смотрит до choosing[j], то есть выходит, когда j-й процесс не хочет входить в критический участок, после чего смотрит для j-го процесса false, то есть нужно: либо - чтобы он хотел войти в критический участок, либо - чтобы

выбранные номера j и i были меньше, чем [i],i, то есть для данного процесса. Если это условие дает ложь, то j-й процесс входит в критический участок, иначе он говорит, что больше не хочет, и выполняет свою оставшуюся часть секции. Это доказательство, конечно же, не такое простое, как для алгоритма Петерсона, но построено примерно на тех же самых принципах: взаимное исключение доказывается от противного; то, что есть продвижение - легко увидеть, потому что оно будет достаточно быстрым, то есть за несколько проходов других процессов по своим критическим участкам. Скажем прямо, алгоритм Петерсона для двух процессов - более симпатичен и понятен, чем алгоритм булочной для n - процессов.

Аппаратная поддержка взаимоисключений

В действительности с аппаратной поддержкой дело обстоит не очень хорошо. Мы рассмотрим некоторые команды, которые поддерживаются аппаратурой, но в чистом виде они не обеспечивают взаимного исключения, к сожалению. Но иногда, используя эти команды, можно обойтись без взаимного исключения, они как раз хороши для случая, когда реально есть параллельные процессоры.

Самое простое аппаратное средство для поддержки взаимного исключения - это блокировка прерываний. Её нельзя использовать на пользовательском уровне, то есть нельзя заменить услуги операционной системы для синхронизации, но сами операционные системы активно используют блокировки прерываний как раз для взаимного исключения. Кроме того, во многих вычислительных системах, особенно многоядерных, многопроцессорных имеются специальные атомарные команды процессора, которые позволяют проверить и изменить значение машинного слова (как одно действие) или поменять местами значения двух машинных слов в памяти. Посмотрим, как такие команды можно использовать для реализации взаимных исключений. Но, к сожалению, полностью они не позволяют отказаться от программной поддержки

Первая команда исторически называется **Test-and-Set**. Можно считать, что она выполняет следующую функцию, то есть у нее есть один параметр - указатель на целое. Она сначала считывает значение этого целого и присваивает туда 1.

```
int Test_and_Set (int *target){
  int tmp = *target;
  *target = 1;
  return tmp; }
```

Можно с использованием этой атомарной команды попробовать модифицировать алгоритм для переменной-замка, чтобы там обеспечивалась атомарность, чтобы он обеспечивал взаимоисключения, тогда это будет выглядеть следующим образом:

```
shared int lock = 0;
```

```
while (some condition) {
  while(Test_and_Set(&lock));
    critical section
  lock = 0;
    remainder section }
```

Есть lock, который устанавливается командой Test_and_Set, после выполнения которой в lock будет находиться 1, чтобы следующий процесс циклился, то есть он выйдет из while(Test_and_Set(&lock)), если там 0, а войдет, если там 1. К сожалению, эта конструкция не обеспечивает условие ограниченного ожидания для алгоритмов. После того, как процесс дошел до эпилога критического участка и сбросил lock на 0, он спокойно может войти туда в следующий раз. Это никак не ограничивается, то есть в действительности может так сложиться, так может планировать планировщик, что другой процесс никогда не попадет в критический участок.

Команда Swap. Команда обменивает значения двух переменных a и b, находящихся в памяти, это можно проиллюстрировать следующей функцией:

```
void Swap (int *a, int *b){
  int tmp = *a;
  *a = *b;
  *b = tmp; }
```

Параметры - это два указателя на переменные a и b. Тогда сначала считывается значение переменной a, туда присваивается значение переменной b, соответственно - b присваивается бывшее значение a. Тогда, применяя атомарную команду Swap , можно реализовать предыдущий алгоритм, введя дополнительную логическую переменную **key**, локальную для каждого процесса:

```
shared int lock = 0;
int key;
while (some condition) {key = 1;
  do Swap(&lock,&key);
  while (key);
    critical section
  lock = 0;
    remainder section }
```

В этом случае key присваивается 0, выполняется обмен между замком и ключом до тех пор, пока ключ не стане равным 0, после чего процесс входит в критический участок. Легко увидеть, что это тоже не совсем полное решение, потому что здесь тоже не гарантируется отсутствие бесконечного ожидания.

Алгоритмы, которые опираются на атомарные команды, - не полные, и так ими пользоваться нельзя. Интересно, что в современных многоядерных процессорах, когда

число процессоров настолько велико, что операционные системы дают слишком дорогую синхронизацию, то обходятся этими командами. Что происходит для того, чтобы не делать полноценные критические участки. Это очень хитроумное программирование, где в каждом случае необходимо делать по-своему, но тем не менее приходится это делать именно так.

Обратим внимание, что в многоядерных системах просто блокировать прерывание - мало, потому что нет такой возможности - заблокировать прерывание во всех процессорах одновременно, то есть нет такой команды. Существует сейчас или нет, но когда-то на многопроцессорных системах была специальная команда, которая приводила к тому, что процессор зависал на чтении какой-то переменой, какого-то машинного слова до тех пор, пока там не будет 1. Чтобы все остальные процессоры когда-то ему сказали, что теперь можно дальше двигаться. В действительности это ужасная команда, потому что она приводит к тому, что процесс не работает, он просто стоит и жужжит, пока идет такой внутренний цикл. Полноценной синхронизации в многопроцессорных системах без этой команды сделать невозможно.

Заключение:

- Последовательное выполнение действий, направленных на достижение определенной цели, называется активностью. Они состоят из **атомарных операций**, которые выполняются неразрывно, как единое целое.
- При исполнении нескольких активностей в квазипараллельном режиме атомарные операции могут перемешиваться между собой с соблюдением порядка следования внутри активностей. Это явление получило название **interleaving/чередование**.
- Если результаты выполнения нескольких активностей не зависят от варианта чередования, то такой набор называется **детерминированным**, в противном случае он называется **недетерминированным.**
- Существует достаточное **условие Бернстайна** для определения детерминированности набора активностей, оно накладывает жесткие ограничения.
- Недетерминированный набор активностей имеет **race condition/условие гонки, состязания,** устранение race condition возможно при ограничении допустимых вариантов чередований атомарных операций с помощью синхронизации поведения активностей.
- Участки активностей, выполнение которых может привести к race condition, называют **критическими участками**.
- Необходимым условием для устранения race condition является **организация взаимоисключения** на критических участках: внутри соответствующих

критических участков не может одновременно находиться более одной активности.

- Для эффективных программных алгоритмов устранения race condition помимо условия взаимоисключения требуется выполнение следующих условий: алгоритмы не используют специальных команд процессора для организации взаимоисключений, алгоритмы ничего не знают о скоростях выполнения процессов, алгоритмы удовлетворяют условиям прогресса и ограниченного ожидания. Все эти условия выполняются в **алгоритме Петерсона** для двух процессов и в **алгоритме булочной** – для нескольких процессов.
- Если применять специальные команды процессора - **test-and-set, swap** (что означает повышение уровня операций, которые выполняются атомарно), то, не достигая всех пяти условий, алгоритмы синхронизации процессов можно сильно упростить.

Механизмы синхронизации

Мы рассмотрим три разновидности механизмов синхронизации: **семафоры, мониторы и сообщения**. Средств синхронизации, конечно, гораздо больше. Сложно точно сказать - сколько их в действительности было придумано. Когда-то отечественные специалисты использовали другие средства синхронизации для обеспечения взаимодействия процессов. Семафоры использовались, а мониторы и сообщения - нет. В другой операционной системе использовались только сообщения, но не использовались семафоры и мониторы. Мы докажем, что эти три группы механизмов эквивалентны, то есть один можно реализовать через любой из других. Главное, что они важны практически.

Алгоритмы Петерсона и булочной, которые мы рассмотрели - корректны, мы доказали, что алгоритм Петерсона работает корректно. Но они громоздки и не элегантны. Элегантность не так важна. Но в прологе процессу приходится циклится - это серьезный недостаток, потому что это означает, что мы обеспечиваем взаимное исключение за счет того, что процессор достаточно много времени простаивает. Это даже хуже, чем то, что предлагают в Unix, в котором, например, есть такой прием: если какое-то условие для процесса не выполнено, то ему предлагают не ждать, чтобы не было тупиков, а отложиться по таймеру, поспать, а потом ещё раз проверить, если условия возникнут, то продолжить работать. Это тоже довольно несуразное решение, но решение с вращением в пустом цикле - совсем несуразное, потому что мы теряем ресурсы, а никакого толка от этого нет. Есть и другие недостатки, которые свойственны алгоритмам, которые реализуются на уровне пользовательских программ.

Предположим, что в вычислительной системе есть два взаимодействующих процесса: H – с высоким приоритетом (от hi), L – с низким приоритетом (от low). Предположим, что алгоритм планирования процессов вытесняющий, то есть процесс с высоким приоритетом вытесняет низкоприоритетный процесс всякий раз, когда он

готов к выполнению. Он занимает процессор на все время своего CPU burst, если не появится процесс с еще большим приоритетом. Предположим, что так сложилось, что процесс L вошел в свой критический участок, а процесс H в это время активизировался (стал готовым), ему дали процессор, он дошел до входа в критический участок и зациклился. Тогда он не может выти из этого цикла, потому что некому открыть код из-за того, что процесс L не работает. Процесс L не может работать потому, что он имеет низкий приоритет, и его не пускают на процессор. Чтобы не допустить возникновения подобных проблем, были разработаны механизмы синхронизации более высокого уровня: семафоры, мониторы и сообщения. Ровно такая же ситуация может возникнуть, когда семафоры реализованы в операционной системе, только необходимо иметь не два, а три семафора. Тогда может быть так, что один процесс семафор закрыл, ему не дают работать, чтобы он его открыл. Один процесс ждет семафор, а третий процесс работает и не дает процессу L закончить, чтобы открыть свой семафор. В действительности такие ситуации бывают, никакие семафоры на уровне операционных систем от них не спасают.

Семафоры

Одним из первых механизмов для синхронизации поведения процессов стали семафоры, которые предложил голландский профессор **Дейкстра** в 1965 году. Его алгоритм, его понятие "семафор" и правила работы с семафорами были описаны в статье, которая была опубликована на русском языке. Поразительно, что она вышла, в то время велась настоящая холодная война, а конференция проходила под эгидой НАТО. В Советском Союзе выпустили сборник трудов конференции НАТО на русском языке (это была одна из первых книг про операционные системы).

Концепция семафоров: Семафор - это целочисленная переменная, принимающая неотрицательные значения. По крайней мере, семафор Дейкстры, в Unix семафоры могут принимать отрицательные значения. За исключением момента инициализации переменной - доступ к семафору может производиться только через две атомарные операции: **P** (от датского слова **proberen** – проверять) и **V** (от **verhogen** – увеличивать). Классическое определение операций P и V выглядит следующим образом:

```
P(S): пока S = 0 процесс блокируется;
        S = S – 1;
V(S):  S = S + 1;
```

Эта запись означает следующее: при выполнении операции P над семафором S сначала проверяется значение семафора. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над семафором S к его значению просто прибавляется 1.

Семафоры могут успешно применяться для решения различных задач организации взаимодействия процессов. Существовали языки программирования, где понятие "семафор" было введено в синтаксис языка (например, в ALGOL-68). В других случаях семафоры применяются на основе специальных системных вызовов. В этом случае сама переменная-семафор располагается внутри адресного пространства ядра операционной системы. Ядро обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказываются несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

Решение проблемы производителя-потребителя. Типичная задача, совершенно стандартный пример, который приводится во всех учебниках про операционные системы, - это взаимодействие процессов **producer-consumer**: есть два процесса, один из которых информацию производит, а другой потребляет. Производитель информации кладет её в буфер, а потребитель оттуда её извлекает. Их деятельность можно описать следующим образом:

Producer: while(1) {	**Consumer: while(1) {**
produce_item;	get_item;
put_item; }	consume_item; }

Процессы бесконечно циклятся, производитель производит и производит, кладет и кладет в буфер, а потребитель берет из буфера, потребляет и потребляет до бесконечности. Очевидно, что буфер конечный, мы предполагаем, что он бесконечным быть не может. Если буфер забит, то производитель должен ждать, пока в нем появится место, чтобы туда было можно положить новую порцию информации. Если буфер пуст, то потребитель должен дожидаться нового сообщения, когда там появится то, что можно прочитать.

Как можно реализовать эти условия с помощью семафоров? Возьмем три семафора: **empty, full и mutex**. Семафор full будет использоваться для того, чтобы потребитель мог ждать, пока в буфере появится информация. Семафор empty - это организация ожидания производителя при заполненном буфере (когда освободится место), а семафор mutex - организация взаимного исключения процессов на критических участках, к которым относятся действия **put_item** и **get_item**. Потому что они оба работают с буфером, один туда пишет - другой читает. Тогда решение задачи выглядит следующим образом:

```
Semaphore mutex = 1;
Semaphore empty = N, где N – емкость буфера
Semaphore full = 0;
```

Producer: while(1) {	**Consumer: while(1) {**
produce_item; P(empty); P(mutex); put_item; V(mutex); V(full); }	P(full); P(mutex); get_item; V(mutex); V(empty); consume_item; }

Тогда Producer формирует соответствующий элемент информации, закрывает семафор empty, если empty не нулевой, то мы проваливаемся и, соответственно, уменьшаем там на 1 число свободных элементов. После чего закрывается mutex (это взаимное исключение), в буфер пишется этот элемент, открывается mutex, и, наконец, прибавляется 1 к full. Full поначалу у нас был 0, сейчас там появился первый элемент. Производитель первым действием смотрит - есть ли что-нибудь в буфере, то есть поначалу там ничего нет, так что если он запустился первым, то он здесь застрянет до тех пор, пока там не появится элемент. Далее он закрывает семафор взаимного исключения mutex, берет оттуда элемент, открывает семафор и увеличивает значение семафора empty на 1, говоря, что он освободил в буфере один элемент, потребляет соответствующий буфер. Легко увидеть, что это решение корректное, семафоры использовались для организации взаимного исключения на критическом участке и для синхронизации скорости работы процессов при использовании ограниченного буфера.

Двоичные семафоры. Предположим, что мы решили обойтись двоичными семафорами, то есть такими, которые сохраняют всего два состояния - 0 и 1. Все остается в силе: если значение семафора 0, то выполнение операции P приводит к блокировке процесса, если там не 0, то вычитается единица, и значение становится равным 0. Как можно переписать это решение? Представим, что внутри ядра операционной системы реализован двоичный семафор, мы хотим написать отдельные пользовательские функции, которые будут работать с переменными N j-ми семафорами, нам разрешается использовать каждую функцию для одного семафора. Мы привязываем к семафору эту функцию. Вопрос следующий: сколько потребуется двоичных семафоров, чтобы реализовать операции P и V над пользовательскими N j-ми семафорами? Эта задача очень похожа на задачу "производитель и потребитель". Если выполняется операция P, то необходимо проверить состояние семафора и блокировать процесс, если там 0, иначе - вычесть оттуда 1. Операция V: посмотреть - не ждет ли кто этот семафор. Если никто не ждет, то прибавить 1. Сколько двоичных семафоров для этого необходимо? Интересно, что бывают совершенно разные ответы, но в действительности необходим один семафор для взаимного исключения и один семафор для того, чтобы подталкивать процесс, который ждет. Для producer-consumer двумя семафорами обойтись нельзя, необходимо больше. Важно учесть, что необходимо понимать, как их использовать.

Лекция 6. Механизмы синхронизации

Семафоры. Продолжение

Первая операционная система, с которой мне довелось работать, была целиком основана на использовании общей памяти, легковесных процессах и т.д.**,** поэтому двоичные семафоры были там основным средством синхронизации. Их приходилось использовать очень много (в 1 000 мест в ядре), приходилось делать очень много критических участков и т.д. Причем было две разновидности семафоров: семафоры, которые использовались только внутри операционной системы для организации критических участков, и семафоры, которые разрешалось использовать пользователям, то есть там было больше контроля. Но была проблема: когда очень много семафоров, то очень трудно разобраться. Одна из распространённых ошибок в системе, когда неправильно сделан порядок закрытия семафоров, то рано или поздно процессы подают в deadlock/тупик. Картинка, с одной стороны, понятна: у нас тупик, потому что закрыты такие-то семафоры, процессы ждут открытия семафоров, которых они никогда не дождутся. Но зачем они их ждут? Понять это, когда мы смотрим на дамп, разобраться, что же там произошло - очень трудно, потому что **семафоры** – это механизм формальный, это всего лишь переменная, она абсолютно оторвана от семантики, для которой собственно семафоры используются, то есть зачем он был закрыт в этот конкретный момент. С этим в действительности очень непросто разбираться.

Решение задачи producer-consumer с помощью семафоров выглядит достаточно изящно, но программирование с их использованием требует осторожности и внимания, чем напоминает программирование на языке Ассемблера. На языке Ассемблер программировать трудно не потому, что там какие-то особые сложности, а из-за того, что это очень низкий уровень программирования. Там приходится писать очень много команд, за набором команд скрывается смысл программирования, а здесь смысл программирования скрывается из-за того, что от того, зачем необходима синхронизация - оторвано средство синхронизации. Пример нехорошей ситуации: в коде ниже обязательно сначала (и в том, и в другом случае) идет P от семафора, который управляет прохождением процессов, к семафору, который делает критический участок. Предположим, что мы поменяли местами операции P: сначала закрываем семафор mutex, а потом семафоры full и empty. Тогда предположим, что какой-то потребитель входит в критический участок - значит он закрыл mutex, то есть там 0. Он начинает ждать семафор full. Очевидно, что поначалу буфер пуст. Значит, процесс блокируется, а программист считает, что процесс ждет, пока в буфере появятся какие-то новые элементы данных. Но производитель в свой критический участок войти не сможет, поскольку семафор mutex закрыт, поэтому он никогда не сможет туда ничего положить. То есть мы получаем тупиковую ситуацию, которая возникает только из-за того, что существует неправильный порядок закрытия семафора.

Producer: while(1) {	**Consumer: while(1) {**
produce_item; P(empty); P(mutex); put_item; V(mutex); V(full); }	P(full); P(mutex); get_item; V(mutex); V(empty); consume_item; }

Ситуация может быть более забавной. Предположим, что mutex сделан так, что один и тот семафор закрывается в разных точках программы, то есть по смыслу – это (и то, и другое) критические участки, которые мы должны ограждать семафором, но программист забыл его открыть и второй раз попадает на закрытие семафора mutex. Тогда он затыкается на своем собственном семафоре. Это ошибка, которую в действительности может ловить сама операционная система, потому что нехорошо, когда семафор закрывается повторно в том же самом процессе, не открывая его. Но контролировать это внутри ядра - довольно накладно. Это происходит из-за того, что использование семафора оторвано от того, для чего оно используется. В сложных программах проанализировать правильность использования семафоров вручную очень непросто, здесь не работают обычные способы отладки программ. Не работают практически никакие способы: с отладчиком это просто нельзя сделать, встраиваемые печати тоже не работают. Самое печальное, что возникновение ошибок зависит от того, как чередуются атомарные операции или как они параллельно выполняются. Их может быть очень трудно воспроизвести. В действительности бывали такие случаи, когда известно, что в программе есть ошибка, известно, что она уже была получена, воспроизведена, но повторно её произвести не удается в течении года, потому что подобрать имена, именные соотношения - крайне трудно.

Мониторы

Известный **Тони Хор** (Hoare) в 1974 предложил механизмы еще более высокого уровня, чем семафоры, которые получили название "монитор". Они назывались мониторы Хора, хотя в последствии их стали называть **мониторами Хансона**. Это ещё один датчанин, который смог усовершенствовать то, что предложил Хор.

Монитор можно представлять, как некий тип данных (в духе Барбары Лисков), который можно внедрить в языки программирования. Были такие языки программирования, в которых мониторы присутствовали как синтаксические единицы. Монитор обладает своими собственными переменными, которые определяют его состояние. Как любой класс, как любой абстрактный тип данных, то есть у него есть внутреннее состояние, есть набор функций-методов, которые позволяют обращаться к функциям монитора, к его переменным состояния. Функции-методы инкапсулированы,

то есть они могут использовать в своей работе только данные, которые находятся внутри монитора, и свои параметры, которые они получают на вызове.

```
monitor monitor_name {
  описание внутренних переменных;
  void m1(...){...
  }
  void m2(...){...
  }
  ...
  void mn(...){...
  }
  {
   блок инициализации внутренних переменных;
  }
}
```

На абстрактном уровне структуру монитора можно описать следующим образом: заголовок, описание монитора и набор методов, которые вводятся условно (в действительности они могут быть любого типа, потому что это функции), а также блок инициализации внутренних переменных, который работает при образовании монитора (как в классах объектно-ориентированных языков). Здесь m1, … , mn - это функции, которые представляют собой функции-методы монитора, а блок инициализации внутренних переменных содержит операции, которые выполняются один и только один раз: при создании монитора или при самом первом вызове какой-либо функции-метода до ее исполнения.

Самая важная особенность мониторов состоит в том, что в любой момент времени только один процесс может быть находиться в состоянии **"готовность"** или **"исполнение"** внутри данного монитора. То есть не разрешается нескольким процессам попасть в монитор через разные входы одновременно. Как их можно реализовать? Поскольку это конструкция высокого уровня, то если в язык встроены мониторы, то компилятор может отличить вызов функции, которая принадлежит монитору, от вызовов других функций и обработать его специальным образом, добавляя к нему пролог и эпилог, реализующий взаимное исключение. В этом случае взаимное исключение привязано к тому месту, где оно собственно требуется. Если мы видим, что какой-то процесс блокирован при попытке доступа к монитору, то мы понимаем, зачем он это делает - он пытается обратиться к какой-то функции монитора. То есть обязанность конструирования механизма взаимного исключения в этом случае возлагается на **компилятор**, а не на программиста. Работа программиста при использовании мониторов существенно упрощается, а вероятность возникновения ошибок становится существенно меньше. Когда-то мониторы во многом способствовали тому, что мы сделали в первой операционной системе с

настраиваемыми модулями: тогда не стали делать взаимные исключения на входе, но это можно было сделать достаточно легко.

Кроме того, одних только взаимоисключений, для того чтобы поддерживать взаимодействие процессов - недостаточно. Нужны еще и средства организации очередности процессов, типа семафоров **full** и **empty**, как в предыдущем примере, где mutex - это взаимное исключение, а full и empty - для того, чтобы регулировать заполненность буфера, чтобы производителю было можно дождаться - пока там есть место, а потребителю дождаться - когда там появится новый элемент данных. Для координации процессов в мониторах было дополнительно введено **понятие условных переменных - condition variables.** Это отдельное средство синхронизации, которое Хор просто встроил в мониторы. Над условными переменными можно совершать операции **wait** и **signal**, которые несколько похожи на операции P и V над семафорами. Только они скорее похожи на двоичные семафоры.

Если функция монитора с какой-то точки - до наступления некоторого события не может продолжать выполняться дальше, то она выполняет операцию wait над некоторой условной переменной. Тогда процесс, выполнивший операцию wait, блокируется, и считается, что он освободил монитор, то есть другой процесс может войти в монитор, так как открылся ограждающий семафор. Когда событие происходит, другой процесс внутри функции-метода выполняет операцию signal над той же самой условной переменной. Ранее заблокированный процесс активизируется. Если несколько процессов выполнили wait для этой переменной, то активизируется только один из них. Когда-то по этому поводу были сомнения: если процессы приоритетны, то есть мы используем приоритетное планирование процессов, и есть некоторый семафор или сигнал, к которому образовалась очередь процессов. Необходимо ли в этом случае упорядочивать эту очередь по приоритетам? Необходимо ли, если сигнал объявляется, чтобы первым активизировался тот процесс, который имеет более высокий приоритет? Если эту очередь упорядочивать по приоритетам, то операция становится намного более сложной. Если система поддерживает приоритеты, то очень велика вероятность того, что процессы будут там находиться в порядке уменьшения приоритетов, то есть тот, кто имеет больше шансов, тот прорвется первым к семафору или сигналу, первым на нем застрянет. Активизация только одного процесса - это нормально, это не противоречит тому, чтобы приоритетность поддерживалась.

Как добиться того, чтобы, когда внутри монитора объявляется сигнал - там не оказалось двух процессов? Чтобы это не привело к тому, что мы нарушаем взаимное исключение? Хор предлагал, чтобы процесс, который пробуждается подавлял выполнение разбудившего процесса, то есть, чтобы он выполнялся, а тот, который его разбудит - откладывался, пока сам не покинет монитор. Хансен предложил другой механизм, который оказался более правильным (как тогда казалось): процесс, который выполнил объявление сигнала сам покидает монитор сразу после выполнения операции signal. Так устроено, что он после этого оказывается вне монитора, ему необходимо

заново в него входить. Дальше мы будем придерживаться **подхода Хансена**. У которого в свое время был замечательный проект - датская микропрограммная машина с очень интересной архитектурой и очень интересной операционной системой. Хансен выпускает замечательные хрестоматии про операционные системы.

Решим задачу производитель-потребитель, используя мониторы:

```
monitor ProducerConsumer {
    condition full, empty;
    int count;
    void put() {
        if(count = N) full.wait;
        put_item;
        count += 1;
        if(count =1) empty.signal; }
    void get() {
        if (count =0) empty.wait;
        get_item();
        count -= 1;
        if(count = N-1) full.signal;
```

Producer:	Consumer:

while(1) {
produce_item;
ProducerConsumer.put();
}
	while(1) {
	producerConsumer.get();
	consume_item;
	}

Есть монитор ProducerConsumer, у него две условные переменные full и empty, кроме того целый count (тут содержится намек, как это делать с двоичными семафорами производителю, потому что там тоже необходимо вводить counter, никуда от этого не денешься). Обратим внимание, что здесь не надо mutex, потому что взаимное исключение обеспечивается на входе. Есть два метода: put без параметров и get без параметров. Put работает следующим образом: если count = N (это означает, что буфер полностью заполнен), то мы ждем сигнала full. В противном случае (значит, он не заполнен полностью), помещаем в буфер элемент и увеличиваем счетчик. После увеличения счетчика count = 1 (то есть ранее он был равен 0), объявляется сигнал "буфер пуст". Get работает следующим образом: если count = 0, то это означает, что там пусто, тогда необходимо ждать, пока в буфере что-нибудь появится. Иначе потребляется элемент, от счетчика вычитается 1. Если после вычитания 1 count стал равен N - 1, значит был равен N, то объявляется сигнал, что буфер полный. Это инициализация count = 0, то есть в буфере ничего нет, туда можно писать. Тогда производитель легко работает в цикле, то есть он просто обращается к операции

монитора ProducerConsumer.put(); Сначала, конечно, он формирует этот элемент. Consumer сначала берет элемент оттуда, а потом каким-то образом потребляет. Как мы видим, вся сложность ушла в программирование мониторов, причем исчезло взаимное исключение. То есть все обеспечивает общий механизм мониторов. Видно, что приведенный пример решает поставленную задачу "производитель-потребитель" корректно, int count - это ровно то, что необходимо будет сделать, если мы будем аналогичным способом реализовывать двоичные семафоры.

Для реализации мониторов требуется разработка специальных языков программирования и компиляторов для них. Мониторы поддерживаются в **параллельном Евклиде** и **параллельном Паскале**. Эмулировать мониторы с помощью системных вызовов не так просто, как семафоры. Это слишком близко к программированию. Можно пользоваться еще одним механизмом со скрытыми взаимоисключениями - механизмом передачи сообщений. Мониторы - красивый механизм, программировать с их помощью хорошо, но их хорошо делать для пользователей, если мы хотим обеспечить удобную среду программирования для пользователей. Хотя люди, которые реально программируют с использованием threads, с параллельными потоками в одной памяти, откажутся от мониторов. Они предпочтут пользоваться mutex, потому что это дорого. Понятно, что простенький двоичный семафор - это дешевле, чем надстройки, более удобные для программирования.

Сообщения

Сообщения - это механизм передачи информации, но его, конечно, можно использовать и для синхронизации. В случае прямой и косвенной адресации, для того чтобы описать передачу сообщений, требуется всего два примитива - **send** и **receive**.

- В случае **прямой адресации**: у send есть параметр, идентификатор или имя процесса, которому адресуется сообщение, и есть буфер, в котором находится подготовленное сообщение; receive - это от кого мы хотим получить сообщение, буфер куда мы хотим это сообщение в виртуальной памяти процесса записать.
    - send(P, message) – послать сообщение message процессу P
    - receive(Q, message) – получить сообщение message от процесса Q
- В случае **косвенной адресации**: вместо имени процессов - имя почтового ящика, то есть в какой почтовый ящик сообщение нужно направить, из какого ящика сообщение получить.
    - send(A, message) – послать сообщение message в почтовый ящик A
    - receive(A, message) – получить сообщение message из почтового ящика A

Сами примитивы send и receive уже имеют скрытый механизм взаимного исключения. Понятно, что они работают с общим буфером, и ядро должно само обеспечивать, чтобы там одновременно не выполнялось send и receive - образных

процессов. В большинстве систем они имеют скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер. Поэтому реализация решения задачи producer-consumer для таких примитивов становится тривиальной. То есть producer просто шлет сообщения, а consumer их принимает.

Передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами - это утверждение спорно. В начале 90-х годов вышла статья, где как раз исследовался механизм обмена данными процессами в одном компьютере. Там было сделано предположение, что самый дешевый способ - это завести для процессов разделяемый сегмент общей памяти и передавать через него сообщения, а для синхронизации предлагалось использовать семафоры (то есть - та же задача "производитель-потребитель" через общую память). **Очередь сообщений** - это отдельный объект операционной системы, то есть они создаются точно так же, как сегменты разделяемой памяти, семафоры и т.д. Там две операции send и receive: send - берет сообщение из виртуальной памяти пользовательского процесса и ставит его в очередь. **Очередь** - это некоторая списочная структура, которая поддерживается памятью ядра, это похоже на буфер, в котором есть порядок сообщений, которые туда поступают. Receive в самом простом случае просто берет из очереди первое сообщение и переписывает его в память получателя.

Оказалось, что в действительности, по крайней мере в Unix, механизм очередей сообщений работал быстрее, чем разделяемая память вместе с семафорами. Потому что при использовании очередей сообщений действительно нет взаимного исключения, там не нужен семафор. Правда, необходимо выйти в память ядра, то есть нужно, чтобы был обработан системный вызов со стороны пользовательского процесса, но и там их придется обрабатывать, потому что там есть семафоры (всё равно системные вызовы). Дорогим получается не то - сколько времени обрабатывается системный вызов, а то - сколько системных вызовов необходимо. Для отправителя и получателя необходим всего один системный вызов, а через общую память получается больше, чем один вызов, потому что один необходим для взаимного исключения, плюс несколько для того, чтобы обрабатывать не бесконечный объем буфера.

Совершенно точно, если для передачи сообщений использовать сокеты - это будет дороже, чем общая память, потому что **сокеты** - это универсальный механизм, они работают и через память, и через сеть, через каналы связи. Поэтому у них больше накладных расходов, чем у очередей сообщений Unix. В результате анализа было установлено, что если принципиально никогда не нужно будет выходить в сеть, то не нужно пользоваться сокетами, а необходимо пользоваться очередями сообщений. Вообще, любой механизм, если он рассчитан на какой-то один способ использования - дешевле, чем универсальный механизм. То есть использовать универсальный механизм имеет смысл только тогда, когда нам действительно необходима универсальность.

Эквивалентность механизмов

Давайте покажем, что если операционная система поддерживает возможность использовать разделяемую память, то все три механизма (семафоры, мониторы и сообщения) эквивалентны между собой. То есть любые два можно реализовать на основе третьего, оставшегося механизма.

Реализация мониторов и передачи сообщений с помощью семафоров

Сначала разберем, как можно реализовать мониторы с помощью семафоров. Для этого нужно уметь реализовывать взаимные исключения и условные переменные при входе в монитор. Для реализации взаимного исключения при входе в монитор заведем семафор mutex с начальным значением 1 (поскольку вначале разрешается вход в монитор любому процессу) и по одному семафору c_i (это семафоры общего вида для каждой условной переменной этого монитора). Когда процесс входит в монитор, компилятор генерирует вызов функции monitor_enter, которая выполняет операцию P над семафором mutex для данного монитора. При нормальном выходе из монитора (то есть при выходе без вызова операции signal для условной переменной), компилятор будет генерировать вызов функции monitor_exit, которая выполняет операцию V над этим семафором. Двоичные семафоры симпатичны тем, что там P и V действительно очень похожи на семафоры. Железнодорожный семафор, который принимает два состояния - красный и зеленый - понятнее, чем семафоры Дейкстры с n-состояниями. Почему они вообще семафоры? Если представить на железной дороге семафоры с n-состояниями, то это будет означать, что n-поездов пропустить можно, а n+1 должны встать. То есть фактически получатся, что там должно быть n-путей после этого семафора, по которым могут проехать разные поезда. Это совсем не та картинка, которую мы видим на семафорах Дейкстры.

- Когда внутри монитора выполняется **операция wait** над условной переменной - компилятор генерирует вызов функции wait, которая сначала выполняет операцию V для семафора mutex, то есть мы открываем вход в монитор. После чего выполняется операция P над соответствующим семафором c_i, блокируя вызвавший процесс. Обратим внимание на тот факт, что здесь нет атомарности, то есть мы сначала разрешили какому-то процессу войти, выполнив V от mutex, потом текущий процесс застревает на семафре. В этом нет ничего страшного, потому что нет никакой содержательной работы, в действительности здесь не требуется никакого взаимного исключения.

- Когда выполняется **операция signal** над условной переменной внутри монитора, компилятор генерирует вызов функции signal_exit, которая выполняет операцию V над соответствующим семафором c_i, и выход из монитора без открытия семафора mutex. Чтобы условный процесс, который связан с условной переменной c_i, работал с условием взаимного исключения внутри монитора. Это

и есть идея Хансона - что процесс выходит из монитора, не открывая семафор mutex.

Semaphore mutex = 1;	**Semaphore c_i = 0;**

produce_item;
void monitor_enter(){
P(mutex);
}
 	
void monitor_exit(){
V(mutex);
}
	P(full);
	void wait(){
	V(mutex);
	P(c_i);
	}
	
	void signal_exit(){
	V(c_i);
	}

Semaphore mutex = 1 - вход в монитор, есть две функции: monitor_enter, в котором семафор mutex закрывается и monitor_exit, в котором открывается. Это, что касается условных переменных. В действительности - это тоже двоичный семафор. Начальное состояние условной переменной - 0, при ожидании сигнала открывается mutex и закрывается семафор c_i; при объявлении сигнала открывается семафор c_i и процесс выходит из монитора. В результате после объявления сигнала семафор mutex откроет семафор, который пробуждается при объявлении сигнала c_i, когда он покинет монитор обычным способом или, когда выполнит новую операцию wait над какой-либо условной переменной. То есть всё совершенно корректно.

Как реализовать передачу сообщений, используя семафоры? Сейчас мы рассмотрим очереди сообщений, только это будут очереди сообщений, которые сделаны вручную, а не так, как в ядре. В ядре - проще, потому что в нем нет общей памяти, в ядре вся память ядра, так что там взаимные исключения не требуются. Для простоты опишем реализацию только одной очереди сообщений: есть общий сегмент между процессами, там выделяется память достаточно большого размера под хранение сообщений, в этой же памяти будет фиксироваться, сколько пустых и заполненных элементов данных находится в буфере, будут храниться ссылки на списки процессов, которые ожидают чтения и записи. Взаимное исключение при работе с разделяемой памятью будет опять обеспечиваться семафором mutex. Кроме того, для каждого процесса, который участвует во взаимодействии - заводится по одному семафору c_i , для того чтобы обеспечивать блокирование процесса при попытке чтения из пустого буфера или при попытке записи в переполненный буфер. Посмотрим, как такой механизм будет работать. Начнем с процесса, который выполняет операцию receive, то есть он получатель сообщения:

- **Процесс-получатель** с номером i сначала всего выполняет операцию P(mutex), то есть он входит в критический участок, когда вся разделяемая память у него в

монопольном владении. После этого он проверяет, есть ли в буфере сообщения. Если сообщений нет, то он себя сам заносит в список процессов, которые ожидают сообщения, выполняет открытие семафора V(mutex) и закрытие семафора P(c_i), которое уже говорит системе, что есть процесс, который ожидает семафор для получения сообщения. Если сообщение в буфере есть, то процесс-получатель читает его, но семафор mutex закрыт, изменяет счетчики буфера. То есть число непрочитанных сообщений становится меньше, а число свободных элементов буфера становится больше. Проверяет, есть ли процессы в списке процессов, ожидающих сообщение. Если такой процесс есть (например, с номером j), то он удаляется из списка, выполняется операция V для его семафора c_j, происходит выход из критического участка. Процесс j начинает выполняться в критическом участке, так как мы семафор mutex не открывали и у mutex значение 0. Никакой другой процесс не может попасть в критический участок. Процесс j смотрит, есть ли ещё сообщения, если они есть, то он делает тоже самое - выбирает сообщение и "будит" ещё один процесс, который ждет сообщений. Если больше процессов, которые ожидают сообщений, - нет, то он выходит из критического участка и производит вызов V(mutex). Процессы, которые попадают на чтение не из пустого буфера, как будто друг друга подстегивают.

- Как работает **процесс-отправитель** с номером i? Сначала опять происходит взаимное исключение для доступа к буферу, то есть процесс, посылающий сообщение, ждет пока он не сможет иметь монополию на использование разделяемой памяти, выполнив операцию P(mutex). Потом он проверяет, есть ли место в буфере, и если есть, то помещает сообщение в буфер, изменяет счетчики и смотрит, есть ли процессы, которые ожидают сообщения. Если таких процессов нет, то он выполняет открытие семафора V(mutex) и выходит из критического участка; если имеется хотя бы один процесс, который ждет сообщения (например, процесс с номером j), то он вызывает операцию V(c_j), а до этого удаляет этот процесса из списка процессов, ожидающих сообщений, и выходит из критического участка без вызова операции открыть семафор V(mutex), давая тем самым возможность разбуженному процессу прочитать сообщение. Если места в буфере нет, то процесс-отправитель заносит себя в очередь процессов, ждущих возможности записи, и вызывает открытие семафора V(mutex) и закрытия семафора P(c_j).

То есть процесс, который читает, освобождая буфер, - пробуждает процессы, которые ожидают места в буфере. Процесс, который посылает сообщение, - пробуждает процессы, которые ждут получение сообщений, и дает им возможность сразу там прочитать все, что необходимо.

Реализация семафоров и передачи сообщений с помощью мониторов

Достаточно показать, что с помощью мониторов можно реализовать семафоры, поскольку на основе семафоров мы уже умеем реализовывать передачу сообщений. Заведем внутри монитора переменную-счетчик, соответствующую числу возможных состояний семафора, и список блокируемых процессов, связанных с эмулируемым семафором, по одной условной переменной на каждый процесс. При выполнении над семафором операции P вызывающий процесс проверяет значение счетчика: если значение счетчика больше нуля, то он уменьшает его на 1, процесс выходит из монитора; если оно равно 0 после вычитания единицы, то процесс добавляет себя в очередь процессов, ожидающих событие, и выполняет операцию wait над своей условной переменной. То есть при выполнении операции V над семафором процесс увеличивает значение счетчика, проверяет, имеются ли процессы, ожидающие открытия семафора. Если такие процессы есть, то один удаляется из списка и выполняется операция signal для условной переменной, соответствующей выбранному процессу. Легко увидеть, что это почти то же самое, что и передача сообщений через общую память, только нет сообщений. Остальное то же самое: очереди процессов и активизация процессов. Не нужно также взаимного исключения, потому что монитор сам его делает.

Реализация семафоров и мониторов с помощью очередей сообщений

Наконец, покажем - как реализовать семафоры с помощью очередей сообщений. Для этого заведем отдельный синхронизирующий процесс. В этом процессе ведется счетчик и очередь процессов, которые ожидают открытия семафора. Чтобы выполнить операции P и V, процессы посылают синхронизирующему процессу сообщения, в которых говорится, чего они желают, после чего ожидают получения от него подтверждения.

Когда синхронизирующий процесс получает сообщения, он проверяет значение счетчика, чтобы выяснить: можно ли выполнить требуемую операцию. Операцию V можно выполнить всегда, а операция P может привести к блокированию обратившегося процесса. Он просто ему не отвечает. Если операция может быть совершена, то она выполняется, и синхронизирующий процесс посылает подтверждающее сообщение. Если процесс должен быть блокирован, то его идентификатор заносится в список блокированных процессов, и подтверждение не посылается. Позже, когда какой-нибудь из других процессов выполнит операцию V, один из блокированных процессов из очереди ожидания удаляется и получает соответствующее подтверждение.

В действительности это можно сделать, но необходимо столько синхронизирующих процессов, сколько семафоров, то есть это дорого. Можно, конечно, сделать так, чтобы один синхронизирующий процесс обслуживал несколько семафоров, но тогда будет сложнее. Тогда будет необходимо следить за разными семафорами и "красить" сообщения (обозначая, какой семафор имеется ввиду).

Поскольку мы показали ранее, как из на основе семафоров можно строить мониторы, то тем самым полностью доказана эквивалентность мониторов, семафоров и очередей сообщений.

Заключение:

Для организации синхронизации процессов могут применяться специальные механизмы высокого уровня, которые могут блокировать процесс, ожидающий входа в критический участок или наступления своей очереди для использования совместного ресурса. К таким механизмам относятся, например, семафоры, мониторы и сообщения. Все эти механизмы являются эквивалентными, то есть, используя любой из них, можно реализовать два оставшихся.

В действительности сообщения не используются для того, чтобы эмулировать мониторы. С использованием сообщений необходимо вести себя по-другому: необходимо, чтобы не было общей памяти, чтобы не было потребности в критических участках, должны быть сервисы, к которым обращаются путем передачи сообщений. В этом процессе инкапсулируется все, что необходимо. Он сам выполняет всю необходимую синхронизацию (если у него один поток, то синхронизация не нужна, сообщения обрабатываются только по очереди) и возвращает подтверждение обратившемуся процессу. Если мы действительно хотим работать с общей памятью, то применимы семафоры и мониторы. Причем мониторы, конечно, предпочтительнее, чем семафоры, потому что они позволяют проще программировать всё, кроме мониторов, которые тоже необходимо программировать (а там нет взаимного исключения, но есть всё остальное, потому что условные переменные не на много приятнее, чем семафоры).

Лекция 7. Синхронизационные тупики

Условия возникновения и основные направления борьбы с тупиками

Это последняя лекция, в которой мы будем говорить об управлении процессами. Она несколько специальная, потому что мы будем говорить о нехороших ситуациях, которые возникают при управлении процессами, а конкретно - про синхронизационные тупики. Сначала мы поговорим о том, что мы понимаем под термином "ресурс", после этого обсудим условия возникновения синхронизационных тупиков/deadlock, поговорим про основные направления борьбы с тупиками, после чего обсудим, что такое алгоритм страуса, какие есть способы обнаружения тупиковых ситуаций, что такое восстановление после возникновения тупика, каковы способы восстановления тупиков путем тщательного распределения ресурсов, как можно бороться с тупиковыми ситуациями за счет нарушения условий возникновения тупиков и, наконец, какие есть родственные проблемы.

В предыдущих лекциях мы рассмотрели возможные способы синхронизации процессов, которые позволяют процессам успешно взаимодействовать. Было видно, что если средствами синхронизации пользоваться не слишком аккуратно, то могут возникнуть непредвиденные трудности. Давайте обсудим, что может случиться: предположим, что несколько процессов конкурируют за обладание неким конечным числом ресурсов. Тогда, если процесс запрашивает ресурс, который в настоящее время недоступен, то процесс, соответственно, должен его ждать, то есть переходит в состояние ожидания. Если требуемый ресурс удерживается каким-то другим ожидающим процессом, то первый процесс не сможет сменить свое состояние. В мультипрограммной системе процесс по определению находится в состоянии тупика, дедлока /deadlock или клинча (это одно из принятых названий дедлоков, которое было распространено в Советском Союзе), если он ожидает события, которое никогда не произойдет. **Системной тупиковой ситуацией** или зависанием системы - является следствие того, что один или более процессов находятся в состоянии синхронизационного тупика.

Другой пример: предположим, что система работает в системах спулинга. Спулинг - это буферизация печати. В этом случае программа, которая производит вывод на печать, должна полностью сформировать в промежуточном файле свои выходные данные, после чего начинается реальная выдача на печать, реальный доступ к принтеру уже спулером - специальным процессом, который разгружает очередь заданий на печать. Задания могут оказаться в тупиковой ситуации, если буфер промежуточных файлов (куда делается спулинг) будет заполнен до того, как одно из заданий закончит свою работу, то есть они оба ещё не сформировали свои файлы для печати, обоим заданиям не хватает места в буфере для того, чтобы это формирование можно было закончить. Возможные решения: самое очевидное - это увеличить размер буфера, если не помещается, или, например, считать, что при достижении какого-то

порогового значения свободной памяти в этом буфере дополнительные задания не принимаются.

Множество процессов находится в тупиковой ситуации, если каждый процесс из множества ожидает события, которое может вызвать только другой процесс данного множества. Поскольку все процессы чего-то ждут, и ни один из них не может возбудить событие, которое активизировало бы другой процесс из этого множества - поэтому все процессы будут вместе чего-то ожидать. Понятно, что обычно то событие, которого ждет процесс в тупиковой ситуации - это освобождение некоторого ресурса. Мы будем обсуждать вопросы обнаружения тупиковых ситуаций, предотвращения тупиковых ситуаций и восстановления после того, как тупиковая ситуация каким-то образом преодолена. Кроме того, мы также рассмотрим тесно связанную с этим проблему бесконечного откладывания, которая может происходить из-за того, что планировщик ресурсов выполняет какою-то дискриминационную политику, то есть какие-то процессы просто не обслуживаются. Во многих случаях цена борьбы с тупиками, которую приходится платить - велика, тем не менее для ряда систем (например, для систем реального времени) другого выхода нет, там не должно быть тупиков.

Концепция ресурса. В предыдущих лекциях мы говорили, что одна из основных функций операционных систем состоит в том, чтобы распределять ресурсы между процессами. Ресурсами могут являться как **устройства**, так и **данные**. Некоторые виды ресурсов могут использоваться процессами совместно, то есть быть так называемыми **разделяемыми** устройствами, например, основная память, процессор, диски и т.д. Другие ресурсы являются **выделенными**, то есть ресурсами, которые запрашиваются в монопольное использование, например – такие, как лентопротяжное устройство (совместно использовать магнитную ленту невозможно). Чаще всего тупики возникают, когда процессы получают монопольный доступ к устройствам, файлам и к другим ресурсам. Мы помним, что если два процесса, например, только читают данные, то они выполняются детерминированным образом. Если два процесса запрашивают ресурсы, которые используются в разделяемом режиме, то между ними никогда не может быть тупика, они не конфликтуют за ресурсы.

Обычная последовательность событий, которая требуется для использования ресурса состоит в том, чтобы:

- Запросить ресурс - **request**
- Использовать ресурс - **use**
- Освободить ресурс **release**

Если ресурса нет в наличии, он недоступен, то процессу приходится ждать. В некоторых операционных системах (практически во всех) такой режим поддерживается, когда при получении отказа на запрос ресурса процесс автоматически

блокируется, то есть прекращает свое выполнение. Он активизируется или просыпается, когда ресурс оказывается в наличии. В других операционных системах, при невозможности выделить запрашиваемый ресурс, процесс получает соответствующий код возврата из ядра операционной системы, а вызывающий процесс должен либо завершиться по собственной инициативе, либо подождать немного и снова попытаться этот запрос осуществить. Например, такой режим есть практически для всех системных вызовов, которые могут блокировать процесс в операционной системе Unix - **no wait**. Это такой флаг, который есть практически везде, который поддерживается во всех системных вызовах, которые могут привести к блокировке процесса.

В этой лекции мы будем предполагать, что при невозможности удовлетворения запроса процесс блокируется, то есть переходит в состояние ожидания. Что такое запрос ресурса - сильно зависит от операционной системы. В некоторых операционных системах имеется системный вызов request для прямого запроса ресурса, в других операционных системах (примером является Unixx) - ресурсами, о которых знает операционная система, являются только специальные файлы. Специальные файлы - это особая разновидность файлов, которая ассоциирована с местными устройствами. Они имеют право открывать только один процесс. В этом случае запрос ресурса делается обычным вызовом **open**. Если специальный файл уже используется, то вызывающий процесс блокируется, пока ресурс не освободится (если нет флага no wait, как мы сейчас предполагаем).

Условия возникновения тупиков

В 1971 году известные специалисты в области компьютерных наук - Коффман, Элфик и Шошани (Coffman E. G., Jr., Elphick M., Shoshani A.) сформулировали следующие четыре условия для возникновения тупиков:

1. **Условие взаимного исключения - Mutual exclusion**. Каждый ресурс выделяется в точности одному процессу или доступен. Процессы требуют предоставления им монопольного управления ресурсами, которые им выделяются.

2. **Условие ожидания ресурсов - Hold and wait**. Процессы, если они ожидают выделения дополнительных ресурсов, - удерживают ресурсы, которые им были ранее выделы.

3. **Условие неперераспределяемости - No preemtion**. Ресурс никогда нельзя принудительно отнять у процесса, его может освободить только тот процесс, который его удерживает, который ранее его получил.

4. **Условие кругового ожидания - Circular wait.** Существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более

ресурсов, требующихся другим процессам цепи. Стрелки, которые идут от процесса к ресурсу, означают, что процесс ресурс ожидает.

Для тупика необходимо выполнение всех четырех условий. Тупик моделируется прямым графом, состоящим из узлов двух видов: прямоугольников процессов и эллипсов ресурсов. Сразу заметим, что такие графы в теории графов называются двудольными. Двудольный граф по определению - это ориентированный граф, в котором множество вершин состоит из двух непересекающихся подмножеств, причем дуги могут вести только из вершин одного подмножества, к вершинам другого подмножества. То есть дуги никогда не связывают вершины одного и того же подмножества. Двудольный граф - две доли вершин. Стрелки, направленные от ресурса к процессу, показывают, что ресурс выделен данному процессу

В связи с проблемой дедлоков в свое время были выполнено много интересных исследований в области компьютерных наук и операционных систем. В 70-е годы одной из областей деятельностей, в которых я и мои коллеги участвовали, было реферирование разнообразных компьютерных изданий для реферативного журнала ВИНИТИ РАН. Из Института точной механики и вычислительной техники поступало множество статей из области операционных систем. Очень много статей было посвящено методам борьбы с тупиковыми ситуациями.

Основные направления борьбы с дедлоками:

1. **Игнорировать** данную проблему - самое простое решение, но, к сожалению, дедлоки - это ситуация плохая, если её полностью игнорировать. Когда в системе возникает тупик, то его можно распознать, глядя на систему только снаружи. Единственный способ борьбы - это перевызвать систему, то есть потерять всю работу, которую она выполняла до сих пор.
2. **Обнаружение** тупиковых ситуаций - соответственно, и их разрушение.
3. **Восстановление после** тупиков - что делать дальше, если мы обнаружили тупик и устранили его, то есть необходимо после этого в каком-то варианте восстановить нормальную работу системы.
4. **Предотвращение** тупиковых ситуаций за счет тщательного выделения ресурсов или нарушения одного из условий возникновения тупиковых ситуаций.

Алгоритм страуса. Самый постой способ – это не обращать внимание на проблему тупиков, игнорировать её. Конечно, это не совсем борьба с тупиковыми ситуациями. Необходимо отметить, что разные люди на эту проблему реагируют по-разному. Для математиков, которые привыкли ко всему относится формально и строго, тупики - это нонсенс, то есть тупики - это в действительности всегда следствие каких-либо ошибок, поэтому необходимо добиваться того, чтобы их не было, любой ценой их надо предотвращать. Если люди обладают более инженерным складом ума, то

возникает вопрос: насколько часто возникает тупиковая проблема и как часто система зависает (и её приходится перевызывать) по другим причинам? Если, например, тупик встречается раз в пять лет, а аварийный останов системы по некоторым другим причинам происходит раз в месяц, то можно не захотеть жертвовать производительностью или удобствами пользователей, чтобы тупики предотвращать или ликвидировать.

Здесь можно привести некоторую аналогию из области баз данных. Мы знаем про системы категории **NewSQL**, которые обычно работают в территориально-распределенном режиме, используя распределенную файловую систему. Одна из основных целей систем - это поддерживать доступность данных. Для глобальных компаний очень важно, чтобы данные были доступны в любом месте мира, чтобы можно было обратиться к данным, которые хранятся поблизости. Для этого они реплицируются, то есть в действительности поддерживаются реплики данных в разных узлах сети.

В начале 2000-х годов известный специалист в области компьютерных наук **Эрик Брюер** сформулировал такую теорему, которую стали называть **Теорема CAP.** Она заключается в следующем: нельзя одновременно обеспечить для распределенной системы управления данными три свойства - **согласованность, доступность данных и устойчивость к разделению сети**. То есть нельзя обеспечить:

- чтобы реплики были идентичны для одного и того же элемента данных;
- чтобы информация была доступна в любом узле где хранится реплика;
- чтобы система устойчиво работала, если рвется связанность сети.

Три этих свойства обеспечить одновременно нельзя. Поскольку на первом месте стоит доступность данных, разработчики системы NewSQL, как правило, жертвуют как раз согласованностью реплик, чтобы система была устойчива к разрыву связанности сети, чтобы при этом данные оставались доступны. По этому поводу несколько лет назад достаточно справедливо заметил **Майкл Стоунбрейкер**. Он сказал примерно следующее: а не надо ли посчитать, насколько вероятна сейчас ситуация разрыва связанности сети? При том, что сети становятся все более надежными, сетевые устройства совершенствуются и т.д. Во всех системах есть программные ошибки. В СУБД они наверняка есть, так как СУБД - это достаточно сложные и большие системы. Не надо ли посчитать, насколько более вероятно, что доступность данных будет потеряна из-за того, что та копия системы, которая обеспечивает доступность к некоторому узлу, остановится из-за программной ошибки? Что в действительности мы потеряем связанность сети. По его мнению - сейчас вероятность программных ошибок гораздо больше. В действительности - это инженерный подход, который очень разумный по отношению к системе NewSQL.

Unix - это система, в которой есть много решений, которые были приняты сгоряча. Например, в ядре операционной системы Unix имеется ряд массивов фиксированного размера, в этом случае операционная система потенциально страдает от дедлоков, даже если они не обнаружены. Например, суммарное число процессов в системе определяется числом элементов таблицы процессов **fork**. Если таблица заполнена, то для программы, которая делает вызов fork, было бы волне разумно подождать какое-то время и снова попробовать осуществить этот вызов. Необходимо ли отказываться от вызова fork, чтобы решить проблему потенциальных тупиков? Аналогично - максимальное число открытых файлов в той же самой операционной системе Unix ограничено размером таблицы дескрипторов. С ними может произойти такая же ситуация. Ещё одним примером является ограниченность пространства свопинга (пространство выгрузки на диски) - опять может возникнуть ситуация, когда процессам не хватает памяти, поэтому возникает потенциальный тупик. То есть фактически любая таблица в операционной системе - это конечный ресурс. В Unix подход состоит в том, чтобы как раз игнорировать данную проблему, то есть пользователи предпочтут случайно возникающую тупиковую ситуацию каким-то правилам, заставляющих их иметь ограниченное число процессов, ограниченное число открытых файлов и т.д. Причем невозможность сказать - сколько этих открытых файлов можно иметь, потому что это зависит от того, как сконфигурирована конкретная версия операционной системы. То есть, в действительности здесь мы сталкиваемся с выбором между строгостью и удобством. Временами удобство, конечно, побеждает, хотя тупики, которые возникают внутри операционной системы - сильно раздражают. Их возникновение означает, что что-то неправильно. В этих случаях при образовании процесса лучше давать код ответа, что его преобразовать нельзя, потому что ресурс исчерпан. Лучше пусть будет иметь место такое нелепое сообщение, чем тупик в системе.

Обнаружение тупиков. Следующий подход - обнаружение тупиков. Он состоит в том, что система автоматически пытается установить тот факт, что возникла тупиковая ситуация, и определить те процессы и ресурсы, которые в эту ситуацию вовлечены. Как правило, алгоритмы обнаружения применяются, когда выполнены первые три необходимых условия возникновения тупиковой ситуации: no preemtion - нельзя ни у кого ничего отнять, hold and wait - если что-то запросил, то все это остается, пока ожидается ресурс, mutual exclusion - монопольное запрашивание ресурса. Если эти условия соблюдаются, то есть система так работает, и возникает подозрение о том, что она попала в тупиковую ситуацию, то проверяется режим кругового ожидания. Поверяется, что возник цикл, возникла цепочка процессов, которые друг друга ждут и не могут освободить ресурс. Для этого активно используются графы распределения ресурсов.

Давайте рассмотрим модельную ситуацию:

- Процесс **A** удерживает ресурс **R** и ожидает ресурс **S**

- Процесс **B** запрашивает ресурс **T**
- Процесс **C** запрашивает на ресурс **S**
- Процесс **D** обладает ресурсом **U** и ожидает ресурсы **S** и **T**
- Процесс **E** удерживает ресурс **T** и ожидает ресурс **V**
- Процесс **F** удерживает ресурс **W** и ожидает ресурс **S**
- Процесс **G** удерживает ресурс **V** и ожидает ресурс **U**

Является ли данная ситуация тупиковой, и если да, то какие процессы в тупик вовлечены?

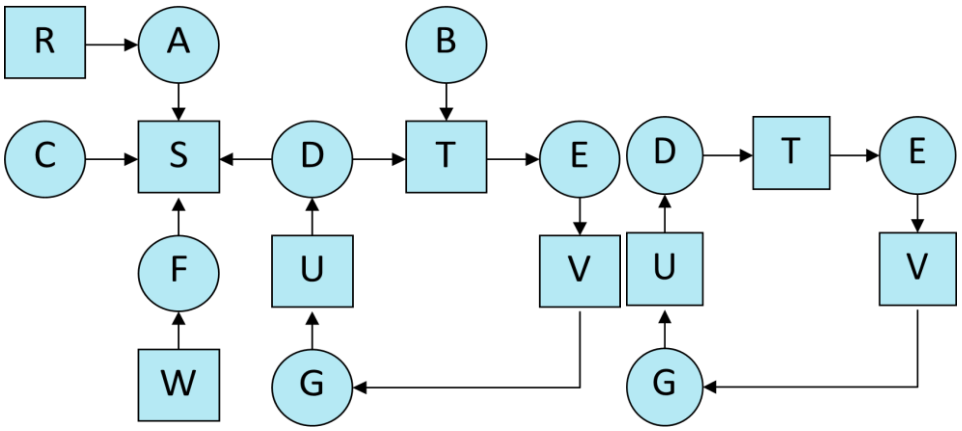

Рис. 7.1. Алгоритм редукции графов

Самый простой алгоритм является алгоритмом редукции графов, который заключается в следующем: просматриваются все процессы, прежде всего смотрится - есть ли такие процессы, которые ресурсами обладают, но ничего не ждут; если такие процессы есть, то считается, что такой процесс находится в безопасном состоянии, он может закончиться. Тогда, если этого ресурса ждет какой-нибудь другой процесс, то на графе меняется стрелка, которая ведет от процесса к ресурсу - на стрелку, которая ведет от ресурса к процессу. То есть он этому процессу отдается. Если процессов несколько (как изображено на рис 7.1.), которые ждут освобождения одного и того же ресурса S, то выбирается какой-нибудь из них, не важно какой. Дальше это действие продолжается до тех пор, пока в системе не остается дуг, которые ведут от процесса к ресурсам, которые показывают, что ресурс чего-то ждет. Это означает, что тупиков там нет. Либо остается некоторый набор вершин, которые ведут от процессов к ресурсам, тогда то, что осталось - образует цикл в этом графе, который обозначает, что там есть тупик. (процессы D, E, G в тупиковой ситуации). В действительности эта задача обнаружения циклов в ориентированных графах является классической задачей теории графов. Этот подход - метод редукции графов является одним из самых примитивных и

простых, существуют более хитрые алгоритмы. Важно отметить, что при выборе алгоритма очень много зависит от того, как граф представляется. Есть масса разнообразных способов представления ориентированных графов в памяти компьютера. Для некоторых из них алгоритмы получаются гораздо проще, чем для других.

Восстановление после тупиков

Предположим, что мы с этой задачей справились, мы построили данный граф. Предположим, что алгоритм обнаружения справился со своей задачей и обнаружил цикл в подграфе, который показывает, что несколько процессов попало в тупик. Что дальше делать? Понятно, что мы искали тупик не для того, чтобы убедиться в его наличии. Пользователям необходимо что-то сделать, чтобы система могла продолжить работать. Один из таких путей - это восстановить нормальное состояние системы, чтобы там не было тупика, и заставить систему работать дальше. Обсудим разные способы восстановления после тупиковых ситуаций. Понятно, что если система попала в дедлок, то её можно вывести из этой ситуации, нарушив одно из условий существования тупиковых ситуаций. При этом, конечно, очень может быть, что один или несколько процессов частично или полностью потеряют результаты проделанной работы. То есть их придется, может быть, перезапускать, а возможно - удовлетвориться тем, что их пришлось закончить.

В действительности сложность восстановления обусловлена рядом факторов. В большинстве систем (можно сказать, что в никаких системах) нет достаточно эффективных средств для того, чтобы приостановить процесс, каким-то образом его из системы изъять, а потом возобновить. Если даже предположить, что такие средства есть, то для их использования требуются большие затраты и помощь человека, то есть помощь оператора. И, наконец, чтобы восстановить систему после серьезной тупиковой ситуации, может потребоваться много работы.

Восстановление при помощи перераспределения ресурсов. Один из способов восстановления - это принудительно завершить некоторый процесс, то есть вывести его из системы, чтобы его ресурсы освободились, и можно было их использовать другими процессами. Для определения того, какой процесс из системы выводить, требуются усилия оператора. Самое главное, что в действительности нельзя процесс возобновить, его можно только начать с начала. То есть в действительности - это борьба с тупиковой ситуацией за счет убиения процессов. В предыдущих лекциях мы говорили, что убить процесс в операционной системе - это не очень простое дело. Всё зависит от того, в каком состоянии он сейчас находится. Очень может быть, что процесс, выбранный для того, чтобы убрать его из системы, не удастся аварийно завершить. То есть это очень непростое решение. В некоторых случаях может оказаться возможным временно забрать ресурс у его текущего владельца и передать его на использование другому процессу. Например, если процесс выводит данные на какой-то монопольно захваченный принтер, то можно остановить процесс и отнять у него принтер. Позвать

человека, чтобы он сложил напечатанные листы, а после этого отдать принтер другому процессу, который поработает, после этого можно отдать его предыдущему процессу, просто продолжить, как будто ничего не случилось, сложить бумагу в лоток, после чего он продолжит свою печать. Возможность забрать ресурс у процесса, дать его другому процессу, а потом вернуть его назад без нанесения ущерба - сильно зависит от природы ресурса. Если бы это был не принтер, а магнитная лента, то так сделать было бы нельзя, потому что в каком состоянии окажется магнитная лента, после того, как с ней поработал другой процесс - совершенно не известно. В этом случае возобновить работу предыдущего процесса может оказаться невозможным. Это решение, которое очень частное. Это возможно в принципе, это можно сделать теоретически, если у процесса нет так называемых побочных эффектов.

Восстановление через откат. В ряде систем реализованы средства рестарта с контрольной точки (сохраненного состояния системы в некоторый момент времени). Представим, например, что во время работы процесса по линии связи посылаются текстовые сообщения о состоянии вычислений. То есть где-то там, далеко-далеко люди получают эти сообщения и смотрят, как всё это происходит. Понятно, что эти сообщения могут передаваться только после того, как в процессе установлена **контрольная точка**. Что делать процессу, когда его откатили к контрольной точке? Он никак не может ликвидировать то, что уже было сделано. Он не может послать сообщение, которое отменит все ранее посланные сообщения. На них в действительности уже могли отреагировать. Это можно делать только для процессов, которые не имеют таких побочных эффектов. Там, где эти средства не предусмотрены, их должны организовать разработчики прикладных программ. Если проектировщики системы знают, что тупик вероятен, то они могут периодически организовывать для процессов контрольные точки. Понятно, что это могут сказать разработчики программ, но операционная система этого узнать никак не может. Если это можно сделать, то в этом случае после обнаружения тупиковой ситуации видно, какие ресурсы вовлечены в цикл на графе, в цикл кругового ожидания. Тогда, для того, чтобы произвести восстановление системы, необходимо чтобы процесс, который владеет нужным ресурсом, был откачен к тому моменту времени, к той контрольной точке, которая предшествует его запросу на этот ресурс.

Вероятно, это самый эффективный способ приостановки и возобновления, если не учитывать, что в действительности устанавливать контрольные точки мы не очень умеем в общем случае. Именно так работают системы управления базами данных: они обнаруживаю тупиковые ситуации и строят такого вида графы. Там откатываются транзакции, то есть в действительности восстановить состояние базы данных при откате транзакции СУБД умеют. Другой вопрос, что обычно с транзакцией связано какое-то приложение. Как откатить приложение к нужной точке - это дело уже не СУБД, это дело операционной системы, и это, к сожалению, мы делать не умеем в общем случае. В действительности это не так уж и важно, потому что, если говорить честно, то в операционных системах не очень умеют строить и графы ожидания

ресурсов. Это происходит из-за того, что для того, чтобы построить граф - необходимо собрать воедино все запросы процесса к разнообразнейшим ресурсам и все ожидания ресурсов от процесса. К сожалению, такого единого горла, через которое поступают все запросы и ожидания, в операционных системах нет. Предположим, что в Unix устройство запрашивается через операцию "системный вызов open" специальных файлов, но там до этого могут быть семафоры, которые специальными файлами не являются, много чего может быть. Построить такой общий граф ожидания - это в действительности означает, что мы должны опросить все возможные компоненты системы, обращение к которым может привести к блокировке процесса. Что сделать чрезвычайно трудно и нескладно. Поэтому, если говорить про графы ожидания для операционных систем, то это скорее утопия, чем реальная возможность.

Восстановление через ликвидацию одного из процессов. Если, тем не менее, мы умеем построить графы ожидания, то простейший способ устранения тупика - это ликвидировать один или несколько процессов. Например, можно ликвидировать какой-то процесс, который находится в цикле графа. Посмотреть, пропадет ли цикл или нет, ели не пропал, то можно ликвидировать еще один процесс и т.д. Не очень понятно, как, собственно, этот процесс выбирать. Ведь может так оказаться, что мы попали в тупик. В цикле участвует 10 процессов, может быть, что достаточно ликвидировать один процесс, и освободившихся ресурсов хватило бы чтобы дедлок разрушить. А можно выбрать такой путь, когда необходимо 9 из 10 процессов ликвидировать, чтобы тупик исчез. То есть необходимо выбирать, с одной стороны, процесс, у которого запрошено побольше разных ресурсов, а с другой - чем больше ресурсов процесс успел запросить, тем он старше, тем более вероятно, что он сам скоро закончится и эти ресурсы вернет. По возможности лучше убивать тот процесс, который может быть без ущерба возвращен к началу, такие процессы называются идемпотентными, например, процесс компиляции. Процесс, который изменяет, например, содержимое базы данных, не всегда может быть запущен повторно. Так что - как выбрать процесс для завершения - это вопрос абсолютно неочевидный.

Способы предотвращения тупиков

Способы предотвращения тупиков путем тщательного распределения ресурсов. Поговорим теперь о том, как можно распределять ресурсы таким образом, чтобы вообще было нельзя попасть в тупик. Целью предотвращения тупиковый ситуаций является обеспечение условий, которые исключают возможность их возникновения. То есть, предоставляя ресурс в распоряжение процесса, система должна принять решение - безопасно ли это или не безопасно. Возникает вопрос: можно ли построить такой алгоритм, который помогает всегда избегать тупиков и делать правильный выбор насчет безопасности? Ответ - да, в принципе такой алгоритм существует: тупиков можно избегать, но только в тех случаях, когда заранее известна некоторая дополнительная информация.

Рассмотрим способы предотвращения тупиков за счет тщательного распределения ресурсов:

Алгоритм банкира. Один из алгоритмов предотвращения тупиков основан на концепции безопасных состояний. Тупиковых ситуаций можно избежать, если распределять ресурсы, придерживаясь определенных правил. Среди алгоритмов этого рода наиболее известен алгоритм банкира, который был предложен датским профессором Дейкстрой. Среди того, что он предложил в свое время, а он предлагал много разных вещей в области операционных систем, все-таки наиболее ценно с точки зрения практики - это семафоры. Алгоритм банкира, конечно, интересен, но все-таки он представляет больше теоретический интерес, применять его на практике практически невозможно. Алгоритм имитирует действия банкира, который, располагая некоторым источником капитала, принимает ссуды и выдает платежи. При этом его задача - выдавать платежи в тех случаях, когда он уверен, что он при этом не обанкротится. Мы применим этот алгоритм к ситуации, когда у системы в наличии n устройств, каждое из которых монопольно запрашивается, например, n магнитных лент. Суть алгоритма состоит в следующем:

- Поскольку максимальный размер капитала - это n, то операционная система принимает от процесса запрос на ресурс, если **максимальная потребность** в этом ресурсе не превышает n единиц.
- Пользователь гарантирует, что если операционная система будет в состоянии удовлетворить его запрос, то все **запрошенные ресурсы будут возвращены** системе в течение некоторого конечного времени.
- Текущее состояние системы называется надежным, если операционная система может обеспечить всем процессам **выполнение их запросов в течение конечного времени.**
- Выделение устройств возможно только в том случае, если **состояние системы остается надежным.**

Рассмотрим пример надежного состояния для системы с тремя пользователями и 12 устройствами, где 10 устройств сейчас запрошены, а 2 имеются в наличии. Пусть текущая ситуация такова: первый пользователь или первый процесс имеет запрошенным одно устройство, а максимальная потребность у него - 4 устройства. Второй пользователь запросил и получил 4 устройства, максимальная его потребность - 6 устройств, третий пользователь запросил 5 устройств, его максимальная потребность - 8.

	Текущее количество	Максимальная потребность
Первый пользователь	1	4
Второй пользователь	4	6
Третий пользователь	5	8

Система удовлетворяет только те запросы, которые оставляют её в надежном состоянии, а все остальные отклоняет. Термин "ненадежное состояние" означает, что при некоторой последовательности событий система может зайти в тупик. Рассматриваемое нами состояние является надежным. Последующие действия системы (которые, собственно, означают, что состояние надежно) могут быть таковы: вначале удовлетворить запросы второго пользователя, который в результате получит 6 устройств, дождаться, когда он выполнит свою работу и освободит свои 6 устройств. После этого 6 устройств достаточно, чтобы удовлетворить запросы 1 и 3 пользователей, то есть нужно 3 устройства отдать первому пользователю и 3 устройства - третьему пользователю. То есть это состояние надежно - известно, как необходимо себя вести.

У алгоритма банкира имеются серьезные недостатки, которые делают его скорее теоретическим, чем пригодным практически, из-за которых могут потребоваться другие подходы для решения проблемы тупиков:

- Во-первых, алгоритм исходит из того, что общее количество ресурсов фиксированное.
- Во-вторых, требуется, чтобы число работающих процессов, которые запрашивают ресурс, оставалось постоянным;
- Требуется, чтобы менеджер ресурсов гарантированно удовлетворял запросы на ресурсы за конечный период времени, а в реальной жизни, конечно, системе необходимы более конкретные гарантии. В компьютерной жизни все время конечно, там не бывает бесконечного времени. Утверждать, что этот алгоритм гарантирует, что все получат то, что хотят и за конечное время - это не говорит нам ничего.
- Требуется, чтобы клиенты гарантированно возвращали ресурсы, причем за конечное время. В реальных системах требуются гораздо более конкретные гарантии.
- Требуется, чтобы пользователи заранее указали свои максимальные потребности в ресурсах. В действительности, когда ресурсы запрашивают динамически, то заранее трудно оценить максимальные потребности. Пользователь, от имени которого запущен процесс, может сам не знать, какие ресурсы этому процессу потребуются при данных параметрах.

Рассмотрим другие, более пригодные на практике способы предотвращения тупиков:

Предотвращение тупиков за счет нарушения условий их возникновения. Как в реальной системе можно избежать тупиков, если отсутствует информация о будущих запросах? А она отсутствует, операционная система (как и все: и программы, и люди) - никогда не знает будущего. Для того, чтобы ответить на этот вопрос, вернемся к четырем условиям возникновения тупиков: условие взаимного исключения

- mutual exclusion; условие ожидания ресурсов - hold and wait; условие неперераспределяемости - no preemtion; условие кругового ожидания - circular wait. Если удастся организовать работу системы так, чтобы, по крайней мере, одно из этих условий не удовлетворялось, то очевидно, что тупик невозможен.

1. **Нарушение условия взаимного исключения**. Тупиков не будет, если нет монополизации ресурсов, если нет взаимного исключения при работе с ресурсами. Понятно, что если использовать реальные физические устройства, то их никак нельзя сделать совместно используемыми. Принтер - это один из наиболее очевидных примеров, но то же самое касается и ряда других устройств. Разрешение двум процессам писать на один принтер в одно и то же время приведет к хаосу. Если сделать виртуализацию принтеров - метод спулинга с промежуточной буферизацией печати в файлах, то одновременное использование печати несколькими процессами становится возможным. В этом случае единственным процессом, который реально взаимодействует с принтером, является спулер - дежурный процесс, который разгружает очередь спулинга и выдает данные на печать. Конечно, это устраняет тупик на основе использования принтера. К сожалению, не для всех устройств возможна виртуализация, например, трудно представить такого рода виртуализацию таблиц процессов. Не очень приятным побочным эффектом являются потенциальные тупики, о которых мы уже говорили, которые происходят из-за того, что разные процессы конкурируют за дисковое пространство для буфера спулинга. Тем не менее, в той или иной форме идея виртуализации ресурсов, для того чтобы было можно избежать тупиковых ситуаций, используется часто. Идея, в действительности, совершенно тривиальная - с помощью виртуализации будем создавать столько виртуальных устройств, сколько необходимо. Каждое из них используется монопольно, а реально они используют один общий принтер за счет промежуточного процесса, который делает привязку виртуальных устройств к физическому.
2. **Нарушение условия ожидания дополнительных ресурсов.** В 1968 г **Хавендер (Havender)** предложил следующую стратегию: каждый процесс должен запрашивать все требуемые ему ресурсы сразу (за один раз), причем процесс не может начать выполняться до тех пор, пока все требуемые ресурсы не будут ему предоставлены. Если же процесс удерживает некоторые ресурсы и получает отказ в выделении дополнительных ресурсов, то он должен сначала освободить все свои запрошенные ресурсы и, если требуется, запросить их снова вместе с дополнительными. Один из очевидных способов предотвращения тупиков состоит в том, чтобы заставить все процессы запрашивать все требуемые ресурсы перед началом выполнения, то есть - это принцип "все или ничего". Если система может выделить процессу все, что ему требуется, то он может спокойно работать до завершения. Если хотя бы один из ресурсов занят, процесс будет ожидать. Этот подход не является очень красивым и привлекательным,

конечно. Он приводит к тому, что снижается эффективность работы компьютера, потому что запрашиваются все ресурсы, которые необходимы процессу во время его жизни (не обязательно те, которые будут необходимы в данный момент). Однако так бывает редко, чтобы все запрашиваемые устройства были необходимы программе одновременно. В принципе, конечно, можно было бы разделить программу на несколько шагов, для каждого шага программы запрашивать ресурсы отдельно, но проблема состоит в том, что во многих случаях процессам до начала их работы неизвестно, сколько ресурсов им потребуется. Поэтому занести эти параметры в программу невозможно. Если такая информация есть, то можно воспользоваться алгоритмом банкира, который мы рассмотрели выше. Но, как правило, такой информации нет, по крайней мере на уровне краткосрочного планирования, на уровне управления процессами, которые уже существуют в системе. Тем не менее, для обеспечения гарантированного отсутствия тупиков в пакетных операционных системах, можно требовать, чтобы при оформлении пакета, при оформлении задания перечислялись все ресурсы, которые потребуются этому заданию. Мы говорили об этом в первой лекции, понятно, что это жесткое требование, которое программистам трудно удовлетворить. Это требование очень неприятное, но зато точно не будет никаких тупиков.

3. **Нарушение принципа неперераспределяемости.** Второй принцип Хавендера состоит в том, чтобы было можно отбирать ресурсы у процессов, которые их удерживают, до завершения этих процессов. Если бы это было всегда возможно, то можно было бы добиться невыполнения третьего условия возникновения тупиков - hold and wait. Но вопрос заключается в том, как отнимать ресурсы у процессов, которые ими владеют. В действительности для этого необходимо процессы откатывать, по-другому никак не получится, потому что если процесс уже использует ресурс, то отобрать его никак не нельзя. Откатывать процессы - это дело очень нетривиальное, как это делать - непонятно. Произвольный процесс с неизвестной природой, неизвестно, что он делает с внешним миром - просто так откатить невозможно. Непонятно, как для него устанавливать контрольные точки. Если процесс использует какой-то набор ресурсов в течение некоторого времени, а потом у него их силком отбирают, то он потеряет всю работу, проделанную до настоящего момента. Это наверняка будет эквивалентно тому, что процесс аварийно завершается. Весь вопрос в цене этого решения, которая может быть слишком высокой, если такая ситуация возникает часто. Процесс проработал десятки минут, накопил массу запрошенных ресурсов, потом у него их отбирают, и в лучшем случае процесс откатывается на начало, а в худшем - завершается. Это очень нехорошо. Другим недостатком этой схемы является то, что даже если к этому не прибегать и автоматически воспроизводить процессы хотя бы от контрольной точки, то они могут попадать в ситуацию, когда у них все время отбирают ресурсы. Это явная дискриминация,

и процессы никогда не посчитаются, никогда не закончатся, что будет очень нехорошо для пользователей, которые работают с данной системой.

4. **Нарушение условия кругового ожидания**. Как ни странно, методы, которые используются для нарушения условия кругового ожидания, как раз относятся к более-менее применимым, правда, не в универсальных системах. Циклического ожидания можно избежать несколькими способами:

    - Первый способ не очень пригоден на практике - это разрешать процессу использовать в каждый момент времени только один ресурс. Если нужен второй ресурс - сначала освободи первый. Бывают такие процессы, для которых - это условие, которое они могут выполнить, но бывают процессы, которые никак не могут работать только с одним ресурсом. Как быть со спулером? Он читает файлы с ленты и печатает их на принтере, он не может обладать при этом только одним ресурсом, ему необходимы и лента, и принтер.

    - Вся проблема - это цикличность, когда процессы выстраиваются в циклический список и каждый из них ждет всех остальных по кругу. Способ избегания состоит в том, чтобы присвоить всем ресурсам уникальные номера (например, возрастающие) и потребовать, чтобы процессы запрашивали ресурсы в порядке возрастания их номеров. Легко убедиться, что если все процессы придерживаются этого правила, то круговое ожидание возникнуть не может. То есть ни один процесс не сможет потребовать ресурс, который уже запрошен каким-то другим процессом, то есть процессы, по крайней мере, не могут попасть в тупик. Один процесс может запросить, но тот, у кого он запросил, тот процесс, который этот ресурс держит, не сможет запросить ресурс, который захвачен первым процессом. В действительности это легко доказать. Небольшой вариацией этого алгоритма является нумерация в возрастающем порядке, но не ресурсов, а запросов процесса. То есть после последнего запроса и освобождения всех ресурсов - можно опять разрешить процессу выполнить первый запрос. Но очевидно, что для универсальной системы (в которой работают непредсказуемые программы, выполняются непредсказуемые процессы) невозможно найти порядок ни ресурсов, ни запросов, который удовлетворил бы поведение всех процессов. Для специализированных систем это можно сделать - там, где работает предопределенный набор процессов, которым заранее известно, что им необходимо.

Родственные проблемы

Тупики не ресурсного типа. К сожалению, если для синхронизации использовать семафоры (для организации критических участков), то такого рода

тупики возникают естественно в результате программных ошибок. Как показывает опыт, самое неприятное для разработчиков операционных систем - если семафоры используются и внутри ядра операционной системы для синхронизации процессов внутри ядра. В этом случае тупики, которые носят такой системный характер, естественно возникают в результате ошибок при программировании ядра операционной системы. Эти ошибки в действительности избежать крайне трудно, самое печальное, что их крайне трудно находить. Здесь стоит подумать, потому что мы ранее рассматривали, что для пользователей, которые использую семафоры для организации критических участков, в качестве альтернативы можно предложить монитор. Мониторы позволяют обеспечивать неявные взаимные исключения при входе в монитор. Они сильно облегчают жизнь на предмет тупиков. Хотя даже в случае мониторов можно легко попасть в тупик, когда внутри монитора выполняется вызов wait, и процесс не дожидается, когда ему объявят сигнал. Это не совсем тупик, это ситуация, когда процесс будет вечно ждать внутри монитора. Самое главное, что если есть параллелизм внутри ядра, то необходимо использовать дешевые средства синхронизации. Самые дешевые - это простые двоичные семафоры. В этом случае без тупиков крайне трудно обойтись, потому что нет ни одной операционной системы, которая была бы полностью, до конца отлажена, наверняка там когда-нибудь, что-нибудь такое проявится.

Starvation/голод - это близкая к тупикам проблема. В динамических системах постоянно происходят запросы процессов к ресурсам. Естественно, в системе должна быть реализована некоторая политика для того, чтобы было можно принимать решения относительно того, кто получит ресурсы и когда. Эта политика может быть дискриминационной по отношению к некоторым процессам, хотя формально они находятся не в тупике. Например, предположим, что имеется физический принтер. Система выделяет его по принципу "первым печатать файл наименьшего размера", то есть спулер выдает на печать, но всегда выбирает файлы наименьшего размера. С одной стороны, это приводит к тому, что принтером может пользоваться большее число процессов, но пользователи с большими файлами могут попасть в состояние starvation, то есть они могут никогда не дождаться своей печати. Если возникает ситуация голода, то можно избежать бесконечного ожидания, например, применяя не такую политику, как печатать всегда сначала маленькие файлы, а, например, FCFS - первый пришедший обслуживается первым. Тогда не будет преимущества у маленьких печатей, но зато не будет и starvation.

Заключение:

- Возникновение тупиков является потенциальной проблемой любой операционной системы. Более того, лично я не знаю ни одной операционной системы, в которой гарантированно не могли бы возникать тупики.

- Тупики возникают, если имеются группы процессов, каждый из которых пытается получить монопольный доступ к некоторым ресурсам и претендует на ресурсы, принадлежащие другому процессу.
- В итоге все эти процессы могут оказаться в состоянии бесконечного ожидания.
- С тупиками можно бороться, можно их обнаруживать, можно избегать тупиков, можно возобновлять работу системы после возникновения тупиков. Однако все эти средства стоят дорого, и соответствующие усилия необходимо предпринимать только в тех системах, в которых игнорирование тупиковых ситуаций приводит к катастрофическим последствиям. Формально это, конечно, так, но фактически это может быть причиной недостаточной отладки системы.

Лекция 8. Простейшие схемы управления памятью

Организация памяти

Начиная с этой лекции, мы будем рассматривать разные вопросы управления памятью. Основной памятью, то есть физической памятью, в которой реально находятся команды и данные, обрабатываемые процессором и виртуальной памятью. Мы начнем с простейших вещей: как программа настраивается на то, чтобы её было можно выполнять на компьютере, в чем состоят простейшие схемы управления памятью. Мы будем обсуждать разнообразные вопросы построения подсистем управления памятью в современных операционных системах. В основном это касается организации виртуальной памяти, которая обеспечивает поддержку и безопасное функционирование больших виртуальных сегментированных адресных пространств процессов. Тщательное проектирование аппаратно-зависимых и аппаратно-независимых компонентов соответствующего менеджера управления виртуальной памятью дает возможность организовать производительную работу этих компонентов. Если говорить про машинно-независимые компоненты управления виртуальной памятью, то здесь, конечно, имеется ввиду операционная система Unix, потому что другие примеры машинно-независимой организации неизвестны.

Главная задача любой вычислительной системы состоит в том, чтобы выполнять программы. Для этого программы и данные, обрабатываемые этой программой (по крайней мере, частично), должны находиться в **основной, главной памяти - main memory**, которую иногда называют оперативной памятью. Основная память (после ресурса процессоров) является наиболее важным ресурсом вычислительных систем. Для работы с памятью требуется наличие в операционной системе тщательно проработанных средств управления. Не так давно основная память действительно находилась на втором месте по стоимости ресурсов вычислительной системы. В действительности и сейчас это так: хотя объемы доступной основной памяти стали гораздо больше, чем раньше, это всё равно самый дорогой ресурс компьютеров после процессора. Часть ядра операционной системы, которая управляет памятью, называется **менеджером памяти**, соответственно он подразделяется на менеджер основной памяти и менеджер виртуальной памяти (там, где она поддерживается). В ходе развития операционных систем в менеджерах памяти было реализовано несколько основополагающих идей. Коротко их рассмотрим:

- Во-первых, это **идея сегментации**. Понятие "сегменты памяти" появились в связи с потребностью в совместном использовании разными процессами фрагментов программного кода (текстовых редакторов, разнообразных библиотек, в том числе математических библиотек и т.д.). Если бы не было сегментов памяти, которые доступны разным процессам, то в памяти каждого процесса необходимо было бы хранить полностью дублирующую информацию сегментов, потому что по давнишнему соглашению программный код при

управлении программы изменять нельзя, то есть это только читаемые куски памяти. Эти отдельные участки памяти, которые хранят данные, отображаемые системой в адресное пространство нескольких процессов, получили название сегментов. Начиная с этого момента, память стала двумерной: есть набор сегментов виртуальной памяти и линейный адрес в каждом сегменте. Адрес состоит из двух компонентов: номер сегмента и смещение внутри сегмента. Оказалось удобным размещать в разных сегментах данные разных типов: код программы, данные (если говорить про программирование на языке С, то, как правило, имеются ввиду начальные состояния статических переменных, то есть переменных, про которые заранее известно, где они находятся в памяти). Можно контролировать характер работы с конкретным сегментом, приписав ему атрибуты, например, права доступа или допустимые типы операций над сегментами. В большинстве современных операционных систем поддерживается сегментная организация памяти, в некоторых архитектурах сегментация памяти поддерживается оборудованием (но в последние годы редко слышно, что используется аппаратная поддержка сегментации).

- Вторая идея состоит том, чтобы разделять память на **физическую** и **логическую** Идея состоит в том, что адреса, которые вырабатываются при выполнении команд, к которым обращаются адреса основной памяти - отличаются от адресов, которые реально существуют в основной памяти.

  - Адрес, который генерируется командами выполняемой программы, обычно называют **логическим.**

  - Если в системе поддерживается виртуальная память, то он называется **виртуальным** адресом.

  - Тот адрес, который видит устройство памяти, который реально привязан к устройству памяти, обычно называется **физическим** адресом.

  Задачей операционной системы (возможно, с поддержкой аппаратуры) является отображение или связывание логического адресного пространства с физическим адресным пространством, то есть с адресным пространством, которое реально соответствует ресурсам основной памяти.

- **Идея локальности**. Идея на первый взгляд тривиальна, потому что, как правило, свойство локальности присуще и обычной человеческой жизни, и природе. То есть обычно в пространстве есть локальность, то есть объекты, которые пространственно близки, они часто характеризуются похожими свойствами. Есть временная локальность, например, если в час дня светит солнце, то очень вероятно, что оно светило на полчаса раньше и будет светить на полчаса позже. Это более вероятно, чем то, что за полчаса до этого шел дождь, а через полчаса он пойдет снова. Свойство локальности - это свойство,

конечно, не формальное и не строгое, а эмпирическое, оно присуще и работе операционный системы. В действительности очень многие решения в операционных системах принимаются как раз с учетом того, что программам свойственна эта локальность. Локальность памяти можно объяснить, если учитывать, что в течение некоторого отрезка времени ограниченный фрагмент кода (то есть любой кусочек кода) работает с ограниченным набором данных. Как правило, это так, хотя и не всегда - в действительности бывают программы, в которых свойство локальности не поддерживается. Свойство локальности позволяет организовать иерархию хранения данных. В этом случае наиболее быстрая память используется для хранения и использования минимума необходимых данных, остальные данные размещаются на устройствах с более медленным доступом, они перемещаются в быструю память по мере необходимости. Типичный пример иерархии - это регистры, кэш, основная память - то, что поддерживается на аппаратном уровне. И внешняя память (**вторичная память**), то есть память, которая подключается на основе подключения внешних устройств. Она управляется при работе компьютеров с помощью операционных систем.

Основная память. Можно считать, что основная память - это некоторый непрерывный массив слов или байт, каждое слово имеет свой физический адрес. Если основная память расширяется с помощью механизмов виртуальной памяти, то это дает дополнительные преимущества: во-первых, основная память может иметь недостаточный объем, чтобы постоянно содержать все необходимые программы и данные. Во-вторых, по крайней мере на текущее время, она имеет **свойство изменчивости - volatile**, то есть она энергозависимая - теряет свое содержимое при отключении питания. Внешняя память, соответственно, может хранить большие объемы данных постоянно/constantly.

Менеджеры памяти в операционных системах в целом занимаются следующим:

- отображают адреса программы на конкретную область физической памяти;
- распределяют основную памяти между конкурирующими процессами, которые борются за то, чтобы иметь в памяти свои программы и данные;
- защищают адресные пространства процессов;
- производят выгрузку образов процессов, то есть их содержимого на внешнюю память, если в основной памяти не хватает места для всех процессов;
- ведут учет свободной и занятой памяти.

Имеется достаточно много схем управления памятью, выбор конкретной схемы зависит от многих факторов. При выборе схеме важно учитывать: механизм управления памятью или идеологию построения системы управления, архитектурные особенности

используемой системы, структуры данных в операционной системе, которые используются для управления памятью, и алгоритмы, которые применяются для этих целей.

В этой лекции мы рассмотрим простейшие схемы управления памятью, в следующих лекциях мы рассмотрим: преобладающую схему организации виртуальной памяти, какие существуют аппаратные и программные средства поддержки, далее - какие есть алгоритмы управления виртуальной памятью.

Связывание адресов. Есть логическая и физическая память, когда выполняется программа, её команды вырабатывают логические адреса. Одной из функций подсистемы управления памятью является преобразование адресных пространств. Пользовательская программа имеет дело с логическими адресами, которые получаются в результате компиляции программ в результате преобразования символьных имен, которые присутствуют в программе на каком-то языке программирования. Логические адреса обычно образуются на этапе создания загрузочного модуля (компоновки программы), то есть когда готовится файл, приспособленный к выполнению. Набор адресов, который генерируется программой, называют, как правило, логическим **(виртуальным) адресным пространством.** Ему соответствует некое **физическое адресное пространство**. Максимальный размер логического адресного пространства обычно определяется разрядностью процессора и, соответственно, размером адреса, который вырабатывается в командах. Для 32-ух разрядных процессоров - это 2**32, сейчас для 64-ех разрядных процессоров - 2**64. 2**64 - это астрономическое число, обычно такого размера основную память на компьютер поставить невозможно. Связывание логического адреса, которое порождается командами, выполняемыми в программе с физическими адресами, может быть произведено до начала выполнения соответствующих команд или в момент выполнения команд.

Обычно до того, как команда начинает выполняться на компьютере, проходит нескольких шагов:

- сначала создается текст на языке программирования;
- далее с помощью компилятора порождаются объектные модули;
- из одного или нескольких объектных модулей строится загрузочный модуль или файл EXE;
- после загрузки этого файла в основную память порождается бинарный/двоичный образ процесса в памяти.

В каждом конкретном случае адреса могут представляться различными способами. Обычно адреса в исходных текстах программ бывают символическими, независимо от того, на каком языке пишутся программы. Будем считать, что мы рассматриваем обычные процедурные языки, в которых есть понятие переменных,

подпрограмм, функций и т.д. Иначе слишком сложно, если ещё говорить про логические языки, то трудно сказать, что там соответствует адресу. Компилятор связывает символические адреса с перемещаемыми адресами. Перемещаемые адреса - это адреса, которые могут корректироваться в процессе связывания объектных модулей. В действительности это коррекция дает возможность компоновщику или редактору связей связать несколько объектных модулей в один загрузочный модуль и при этом уже привязать перемещаемые адреса к соответствующим логическим адресам или виртуальным адресам программы. Каждое связывание - это фактически отображение одного адресного пространства в другое адресное пространство.

Привязка команд и данных к памяти в принципе может быть сделана на следующих шагах:

- **Этап компиляции - Compile time.** Самый простой, самый тривиальный способ - это привязывать программу на этапе компиляции. Если при компиляции известно точное место размещения процесса в памяти, то есть известно, чему соответствует нулевой адрес программы в реальном адресном пространстве, то можно генерировать абсолютные адреса. В этом случае, если стартовый адрес программы меняется (если перемещается этот как бы ноль), что соответствует нулевому адресу, то необходимо перекомпилировать весть текст программы. Так было в самых примитивных системах, которые использовались на заре персональных компьютеров, тогда использовалась дисковая операционная система MS-DOS, которую трудно даже назвать операционной системой. Там не было никаких возможностей, кроме как задать все стартовые точки на стадии компиляции.

- **Этап загрузки - Load time.** Это можно делать в процессе загрузки программы. Если неизвестно, какая будет стартовая точка процесса на стадии компиляции, то компилятор генерирует перемещаемый код. То есть в этом случае в объектный модуль помещается таблица перемещения, в которой в каждой переменной, то есть каждой точке в программе, которая должна иметь связь с абсолютными адресами, помещаются данные об её относительном адресе и координаты в объектном модуле. Чтобы можно было подкорректировать срезу, пройдя по таблице перемещения все относительные адреса. В этом случае окончательное связывание откладывается до момента загрузки выполняемого модуля в память. Если стартовый адрес меняется, то есть если меняется привязка нуля, то нужно всего лишь перезагрузить код, перенастроить код с учетом измененной величины.

- **Этап выполнения - Execution time.** Можно производить привязку команд и данных к памяти во время выполнения. Если во время выполнения образ процесса можно перемещать из одного сегмента памяти в другой, то связывание можно отложить до времени выполнения. Тогда желательно иметь специализированное оборудование, например, регистры перемещения. Вместо

того, чтобы менять все адреса, когда меняется ноль процесса, можно загружать специальные регистры. Например, в IBM System/360 был один такой регистр - стартовый адрес задачи. На первых персональных компьютерах было четыре таких регистра для разных сегментов адресного пространства (например, в MS-DOS).

Простейшие схемы управления памятью

Рассмотрим, как происходило управление памятью в процессе развития операционных систем, как оно менялось, что делалось, и какие были простые алгоритмы. Когда операционные системы только появились, то применялись очень простые методы управления памятью. Самый простой метод - это **статическое распределение памяти**. В этом случае требовалось, чтобы все программы каждого процесса полностью помещались в основной памяти. Тем не менее это не означало, что нельзя поддерживать несколько процессов, то есть система позволяла образовывать новые пользовательские процессы до тех пор, пока все они одновременно помещались в основной памяти.

Эта идея расширялась **идеей простого свопинга**. Самый простой свопинг был на заре операционной системы Unix. В этом случае требовалось, чтобы образ каждого процесса целиком располагался в основной памяти, но время от времени (это делалось для того, чтобы было можно обеспечивать для пользователей режим разделения времени) образ некоторого процесса целиком перемещался из основной памяти во внешнюю память. Соответственно, он заменялся в основной памяти образом некоторого другого процесса. В действительности, конечно же, в чистом виде сейчас такая техника не используется, но мы увидим, каким образом модифицировалась идея свопинга в современных методах управления виртуальной памятью. Такого рода управление памятью используется во встроенных (embedded) компьютерах, в которых действительно есть потребность в разделении времени, но нет потребности в очень строгом управлении, в справедливом разделении.

Схема с фиксированными разделами. К простейшим схемам управления основной памятью относится схема с фиксированными разделами. В этом случае на этапе генерации операционной системы или в момент её старта вся основная память разбивается на несколько разделов, каждый из которых имеет фиксированный размер. Когда образуется новый процесс, то он помещается в тот или иной раздел. Как правило, происходит условное разбиение физического адресного пространства. Связывание логических адресов процесса и физических адресов происходит на этапе его загрузки в конкретный раздел. Как показано на схеме, у каждого раздела может иметься отдельная очередь (как на правой части схемы) или может существовать одна глобальная очередь для всех разделов (как на левой части схемы). В первом случае процесс приписывается к разделу заранее, и система знает, в какой раздел его необходимо помещать. Во втором случае система сама подбирает раздел, в который можно поместить образ этого процесса. Такая схема с разделами была реализована в

IBM OS/360 (MFT) и в операционной системе компании Digital Equipment Corporation (DEC) RSX-11.

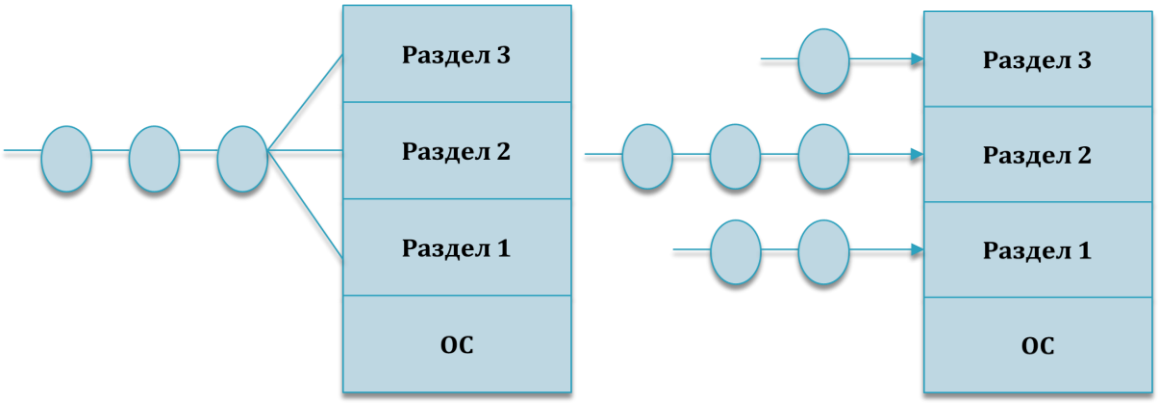

Рис. 8.1. Схема с фиксированными разделами: с общей очередью процессов, с отдельными очередями процессов

Когда приходилось работать с RSX-11, то это выглядело довольно чудновато: необходимо было до запуска задачи сообщить системе, в каком разделе мы хотим её выполнять. Совершенно непонятно, как пользователь системы должен был знать про эти разделы. Во втором случае (как на левой части схемы) подсистема управления памятью смотрит, какого размера образ процесса, какого размера выполняемая программа, выбирает подходящий раздел, производит загрузку программы с требуемым пересчетом адресов, и запускает программу на этом месте основной памяти.

Вопрос: как выбирать раздел, в котрый помещать программу для выполнения? Здесь имеются три стратегии:

- **Стратегия первого подходящего - First fit**, в этом случае испльзуется первый подходящий по размеру доступный раздел.
- **Стратегия наиболее подходящего - Best fit**, в этом случае считается, что наиболее подходящим является тот раздел, в котором после выделения памяти требуемого раздела остается меньше всего свободного места, то есть он больше всего подходит по размеру.
- **Стратегия наименее подходящего - Worst fit**, в этом случае используется раздел, в котором после выделения памяти остается больше всего свободного места.

К сожалению, какой бы не использовался подход для обнаружения раздела, все равно это чревато внутренней фрагментацией памяти. То есть в действительности вероятность того, что программа будет занимать ровно столько места, сколько есть в размере, который для неё выбирается - очень маленькая. Как правило, там будет оставаться память. При использовании первого и второго подходов (то есть наиболее

подходящего и первого подходящего) фрагментация в общем может достигать половины. То есть, конечно, первый подходящий - это самый быстрый способ, потому что берется первый раздел, в котором есть достаточно места. Что касается именно фрагментации, чтобы не пропадала память внутри раздела, как ни странно, опыт показывает, что в действительности лучше работает третий подход - наименее подходящий раздел меньше всего делает фрагментацию (правда, при некоторых дополнительных условиях). Если разделы не дробятся, то, пожалуй, тут все равно ничего не поможет. Если мы выделили какой-то раздел для программы и там осталась половинка, то она будет простаивать. Такие методы распределения памяти применяются и в других компонентах операционных систем, например, для размещения файлов на диске. То есть это способы распределения памяти точно необходимого размера, все они чреваты тем, что возникает внутренняя фрагментация, к сожалению, это дефект общего подхода фиксированных разделов. Настройка адресов для схемы с фиксированными разделами возможна как на этапе компиляции, если заранее приписывается раздел (тогда компилятор знает, какой адрес там должен быть начальным), так и на этапе загрузки, если раньше это сделать нельзя, если разделы выбираются перед тем, как программа выполняется.

Один процесс в памяти. Частным случаем схемы с фиксированными разделами является организация распределения для однозадачных операционных систем. В этом случае вся память представляет собой один раздел, не считая, конечно, места, которое занимает операционная система. Остается только определить, как располагать пользовательскую программу по отношению к операционной системе: выше, чем она находится в памяти, снизу или посередине. При этом часть операционной системы может находиться в памяти с постоянным хранением - ROM (Read-only memory), как, например, в некоторых операционных системах сохраняется BIOS, драйверы устройств и т.д. На это решение влияет вопрос: где находятся вектора прерываний? Если они находятся в нижней части памяти, то есть в части памяти с большими адресами, то операционная система тоже располагается в нижней части памяти. Тогда наоборот - пользовательские программы начинают считать с нуля основной памяти, с нулевого адреса. Так было в старой системе для персональных компьютеров Microsoft MS-DOS, в дисковой операционной системе. Конечно, в этом случае требуется, чтобы пользовательская программа (процесс, который выполняется в однопользовательской системе) не портила код операционной системы. Её необходимо как-то защищать, это можно сделать при помощи граничного регистра, который содержит максимальный адрес, доступный пользовательской программе. То есть за пределы этого адреса она просто не может обратиться, ей не даст это сделать аппаратура, будет вызвано внутреннее прерывание.

Оверлейная структура. Поскольку размер логического адресного пространства процесса может быть больше размера выделенного ему раздела, а также больше, чем размер даже самого большого раздела - довольно часто использовалась техника, которая называется техникой **оверлеев (overlay)** или структуризацией программы с

перекрытиями. Основная идея состоит в том, чтобы сохранять в основной памяти в каждый момент времени только те части программы данных, которые сейчас требуются. Потребность в таком использовании памяти появляется, когда логическое адресное пространство системы маленькое, например, в 16-ти разрядных компьютерах (64 килобайта в PDP-11, 1 мегабайт в MS-DOS), а программа достаточно велика. В этом случае программа представляется в виде дерева. Каждый круг на рис 8.2. - это часть программы, причем каждый круг одного уровня настраивается на один и тот же начальный адрес основной памяти. B и C начинаются с конца. A, D и E будут находится сразу после C, то есть D и E тоже имеют одинаковые адреса. В этом случае коды ветвей оверлейной структуры программы сохраняются на диске как уже настроенные образы памяти, которые считываются менеджером оверлеев, когда это требуется.

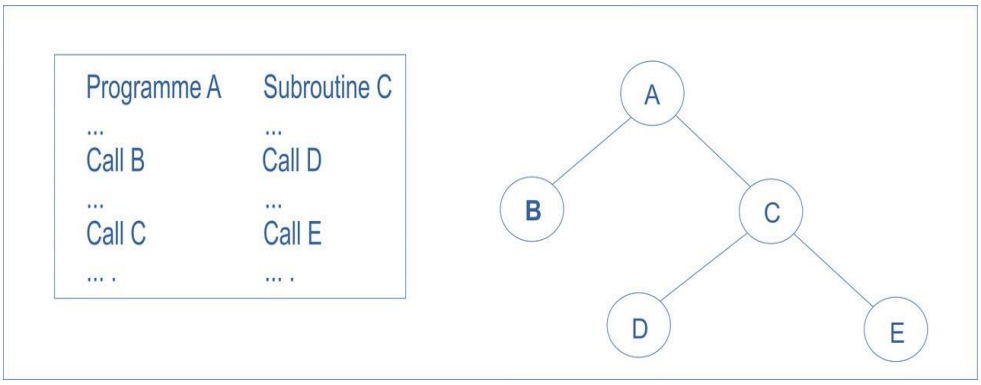

Рис. 8.2. Организация структуры с перекрытием

Для конструирования оверлеев необходимы специальные алгоритмы перемещения и связывания программ. Для описания оверлейной структуры такого рода обычно используется специальный несложный язык, который называется **overlay description language**. То есть совокупность файлов исполняемой программы ещё дополняется файлом, который описывает дерево вызовов внутри программы. Примерно так: A вызывает B и C, C вызывает D и E. Текст этого файла может выглядеть следующим образом: A - (B,C); C - (D,E), то есть в памяти должно присутствовать A, B или C, если присутствует C, то должен ещё присутствовать D или E. Оверлейный загрузчик понимает синтаксис такого файла, настройка памяти происходит заранее или в момент очередной загрузки одной из ветвей программы. В действительности можно все настроить заранее, потому что известно, какие у них будут начальные адреса.

Оверлейные структуры - это очень гибкая вещь, разбиением на такие ветки занимается сам программист. Если программист хорошо понимает, что и когда ему необходимо для того, чтобы программа работала эффективно, то она может хорошо работать и на небольшой памяти. На PDP-11 в системе RSX-11 кроме дисковых оверлейных структур были ещё memory оверлеи. В PDP-11 по тем временам была большая основная память, а виртуальная, логическая память - маленькая. Для всего 16-ти разрядов была 2**22 по максимуму, поэтому там применялся очень интересный

подход: все дерево оверлеев целиком сохранялось в основной памяти, а ветки A, B, C, D, E настраивались на одинаковые виртуальные адреса внутри адресного пространства процесса. Оверлейные структуры не подкачивали с диска необходимые веточки, а операционная система настраивала на эти кусочки память. Такие оверлеи в действительности работали очень быстро, потому что обменов с дисками не было, но понятно, что для этого необходимо все держать в основной памяти и мультипрограммности может не хватить.

Для дисковых оверлеев не требуются специальной поддержки операционной системы, то есть их можно полностью реализовать на уровне пользователя с использованием простой файловой структуры. Всё, что необходимо от операционной системы - это выполнять несколько больше операций ввода-вывода. Типовое решение - это компоновщик, который строит дерево оверлеев и настраивает на память, он вставляет в программы специальные команды, которые включают оверлейный загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

От программиста требуется, чтобы он очень аккуратно проектировал оверлейную структуру, то есть требуется, чтобы он очень хорошо понимал структуру своей программы, кода, когда и какие требуются данные, необходимо знать язык описания оверлейных структур. Поэтому оверлеи применялись только в ранних операционных системах на компьютерах с ограничениями на память. Они практически полностью были вытеснены технологией виртуальной памяти, когда сама операционная система (без вмешательства программиста) решает, какие куски программы необходимо держать в основной памяти. В действительности оверлеи демонстрируют ту самую потребность в локальности программ. Если программа не обладает свойством локальности, то, конечно, оверлейную структуру для неё не очень-то и построить. Необходимо, чтобы в каждый конкретный момент требовалась только часть всего кода или всех данных, которые в программе в принципе содержатся.

Свопинг/swapping. Для пакетных систем можно обойтись фиксированными разделами и никаких более сложных вещей не использовать. Имеется очередь заданий, задание заканчивается, раздел, в котором это задание выполнялось - освобождается, если этот раздел годиться для следующего задания из пакета, то оно туда загружается, и т.д. Если система должна поддерживать разделение времени, то может возникнуть такая ситуация, когда все требуемые пользовательские процессы в основную память не помещаются. Тогда приходится прибегать к свопингу, то есть к перемещению образов процесса из основной памяти на внешнюю память и обратно - целиком. Когда это происходит не целиком, когда перемещаются во внешнюю память части образов процесса - это называется **пейджингом/paging.** Частичная выгрузка уже связана с виртуальной памятью, и это мы рассмотрим в следующих лекциях.

Выгруженный процесс, который перемещен целиком во внешнюю память, может быть возвращен в то же самое адресное пространство или в другое в

зависимости от метода связывания. Если мы применяем ранний метод связывания на стадии компиляции, то необходимо возвращать процесс туда, к чему он привязан. Если можно перенастраивать программу, которая выполняется в процессе, то процесс можно перемещать на другое место, возвращать не туда, откуда его откачали. Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. В системах со свопингом время переключения контекстов ограничивается временем загрузки/выгрузки процессов. Для того, чтобы эффективно использовать процессор, необходимо, чтобы величина кванта времени, который выделяется процессу на процессоре, было гораздо больше, чем время, которое занимает его перемещение из основной памяти во внешнюю и обратно.

Оптимизация свопинга может быть связана с выгрузкой реально используемой памяти, то есть процессов, которые используют, или выгрузкой процессов, которые в настоящее время не функционируют. Кроме того, обычно выгрузка производится в специально выделенное дисковое пространство для свопинга. Это быстрее, чем через обычную, стандартную файловую систему, то есть нет накладных расходов, которые обычно порождает файловая система. Пространство свопинга распределяется большими блоками (иногда экстентами разного размера). Не требуется ни поиск файлов по именам, ни методы распределения памяти по маленьким блокам. Все делается на более грубом и быстром уровне. Во многих версиях Unix (пожалуй, уже во всех версиях) своего рода свопинг, про который мы поговорим более подробно, когда будем говорить про управление виртуальной памятью, в чистом виде обычно не используется. Свопинг начинает работать только тогда, когда система слишком загружена, загружена память, когда не хватает свободной памяти.

Мультипрограммирование с переменными разделами

В принципе, система свопинга может основываться на распределении памяти с фиксированными разделами. Однако, как мы видим, на практике использование фиксированных разделов приводит к внутренней фрагментации и существенным потерям основной памяти в тех случаях, когда мы не попадаем по размеру (а точное соответствие бывает редко). В этом смысле более эффективной является схема с переменными или динамическими разделами. В этом случае сначала вся основная память свободна и заранее на разделы не поделена. Когда система хочет образовать новый процесс, то есть в неё поступает новая задача, для неё выделяется требуемая память. Когда процесс перемещается во внешнюю память (то есть срабатывает свопинг), эта память временно освобождается. В результате может получиться, что по истечении некоторого времени память становится дырчатой, то есть представляет собой набор занятых и свободных участков. Сначала был большой раздел (как на рис. 8.3. слева), в него загружен процесс P_1, далее в него загружен процесс P_2, потом P_3, после этого процесс P_1 заканчивается, образуется дырка, в неё загружается процесс P_4, но он занимает меньше места, остается небольшая дырка. Конечно, когда свободные куски памяти становятся смежными, их можно объединить в один фрагмент. Например,

если благополучно закончатся процессы P_2 и P_3, то образуется большой кусок свободной памяти.

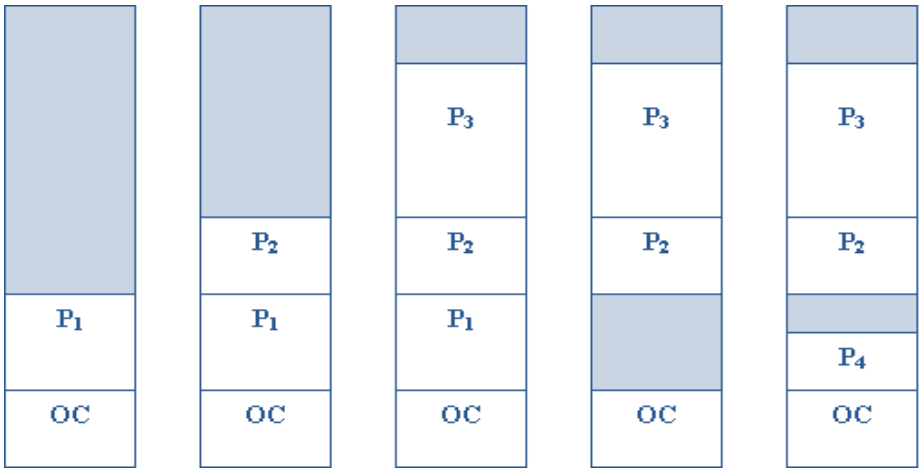

Рис. 8.3. Динамика распределения памяти между процессами (цветом показана неиспользуемая память)

Типовой цикл работы менеджера памяти:

- анализируется запрос на выделение свободного участка, то есть раздела, который необходим для программы;
- выбирается подходящий свободный участок среди имеющихся, это происходит в соответствие со стратегией первого подходящего, наиболее подходящего и наименее подходящего;
- процесс загружается в выбранный раздел, занимая ровно столько места, сколько ему необходимо;
- вносятся изменения в таблицы свободных и занятых областей.

Аналогично после завершения процесса: память освобождается, если рядом есть свободные фрагменты, то они сливаются и т.д. Связывание адресов может быть на этапах загрузки и выполнения, потому что разделы выделяются динамически.

Этот метод по использованию памяти является более гибким и эффективным, чем с метод фиксированных разделов. Но ему свойственен другой феномен, не менее неприятный, чем внутренняя фрагментация, которую мы наблюдали при работе с фиксированными разделами. Внешняя фрагментация - это именно такое явление, когда в памяти есть много свободных фрагментов, но все они маленькие. Ни одного из них не хватает для того, чтобы туда можно было загрузить какой-то процесс. В основном причиной фрагментации является то, что мы никогда не можем найти кусок ровно такого размера, который требуется - он почти наверняка будет немного больше.

Интересно, что как раз здесь, с точки зрения внешней фрагментации распределение памяти методом наименее подходящего дает меньшую внешнюю фрагментацию, но всё равно дает. Проблема в действительности заключается в том, что свободной памяти много, а фрагментов необходимого размера нет. Проблемы фрагментации могут быть различными.

В зависимости от суммарного, общего размера основной памяти и среднего размера образов процессов эта проблема может быть большей или меньшей. В среднем при наличии n блоков памяти пропадает n/2 блоков, то есть треть памяти. Это известное правило 50%: при использовании половины памяти, другая половина пропадает. Два соседних свободных участка, в отличие от двух соседних процессов - могут быть объединены в один. В действительности, к сожалению, это получается редко, одно из решений проблемы внешней фрагментации состоит в том, чтобы разрешить адресному пространству процесса не быть непрерывным, что разрешает выделять процессу память в любых доступных местах. Один из способов реализации такого решения - это **paging**, который используется практически во всех современных операционных системах. Мы будем рассматривать paging, начиная со следующей лекции. Если мы работаем без виртуальной памяти, то есть без paging, то единственным способом борьбы с внешней фрагментацией является сдвижка, то есть **сжатие** - перемещение всех занятых и свободных участков в сторону возрастания или убывания адресов - так, чтобы вся свободная память образовала одну непрерывную область. Этот метод иногда называют **схемой с перемещаемыми разделами**. В идеале фрагментация после сжатия должна отсутствовать, однако сжатие является дорогостоящей процедурой. Алгоритм выбора оптимальной стратегии сжатия очень труден. Как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

Есть компромиссный вариант распределения памяти, который сочетает достоинства и недостатки распределения памяти с внутренней фрагментацией и распределения памяти с внешней фрагментацией - это **метод близнецов или система близнецов**. Данная система выглядит следующим образом: вся память должна иметь размер равный степени числа 2, например, 2^N, можно запрашивать куски памяти не больше, чем 2^N, если запрашивается кусок больше, чем 2^{N-1}, то выделяется вся память целиком, если запрашивается кусок меньше и равно 2^{N-1}, то память делится пополам на два куска размером 2^{N-1}, если запрошенный кусок больше, чем 2^{N-2}, то отдается один из фрагментов размером 2^{N-1}. Если процессу для работы требуется кусок памяти меньше, чем 2^{N-2}, то, например, верхний фрагмент делится ещё на два, и т.д., пока мы не найдем ближайшую сверху степень числа 2 - такую, что требуемый процессу кусок будет больше, чем половина степени числа 2. Видно, что здесь мы заведомо идем на то, что возникает внутренняя фрагментация, но она ограниченная, то есть мы никогда не потеряем памяти больше, чем мы запрашиваем. Мы можем потерять максимум памяти (почти столько же, сколько запрашиваем), если это очень близко к степени числа 2. Внутренняя фрагментация есть, но известно какой она степени, зато с внешней

фрагментацией в этом случае гораздо лучше, потому что метод двоичных близнецов работает таким образом, что вероятность того, что свободные куски окажутся рядом после освобождения памяти - достаточно высока. Здесь свободные куски легко объединяются, если они попали рядом. Метод близнецов - это хороший способ распределения памяти.

Есть ещё более хороший способ распределения памяти - это **метод чисел Фибоначчи** у близнецов, когда используются не степени числа 2, а числа Фибоначчи соответственно. То есть вся память должна иметь размер, который соответствует некоторому числу Фибоначчи - F^N, соответственно дробление происходит по меньшим числам Фибоначчи. В этом случае меньше потерь памяти по причине внутренней фрагментации, не хуже обстоят дела с внешней фрагментацией.

Заключение:

В этой лекции были рассмотрены простейшие способы организации работы менеджера основной памяти. В следующих лекциях будут описываться современные решения, которые связаны с поддержкой виртуальной памяти.

Лекция 9. Средства поддержки виртуальной памяти

Понятие виртуальной памяти

В этой лекции мы поговорим о том, что такое виртуальная память, для чего она необходима, какие у неё основные свойства, какие аппаратные средства бывают для поддержки виртуальной памяти со стороны операционной системы.

Проблема размещения в основной памяти программ, размер которых превышает размер допустимой основной памяти, существует давно. Как мы знаем, эта проблема не исчезает по мере возрастания объем основной памяти. Один из вариантов решения проблемы размещения больших программ - это использование overlay/организации структур с перекрытием, мы говорили об этом в прошлой лекции. Для того, чтобы организовывать оверлейные структуры, требуется активное участие программиста. Для того, чтобы делать разбиение на отдельные оверлейные уровни и загружать необходимые кусочки - необходима серьезная работа программиста в приложении. Поскольку это требует дополнительной квалификации и не всегда тривиально - было предложено попробовать в действительности переложить решение проблемы размещения больших программ на компьютер, на программное обеспечение операционной системы.

По мере развития архитектуры компьютеров усложнялась и организация памяти. Соответственно, расширились и усложнялись задачи операционных систем, связанные с управлением памятью. Одним из основных усовершенствований архитектур компьютеров стало появление **виртуальной памяти - virtual memory.** Многие люди считают, что впервые виртуальная память была реализована в 1959 году в Великобритании, в Манчестерском университете в рамках разработки компьютера Атлас. В действительности практически в это же время виртуальная память появилась в советском компьютере, в лучшем компьютере Советского Союза - БЭСМ-6. Это делалось совершенно независимо от англичан и практически одновременно. Сама идея виртуальной памяти появилась в конце 50-х - начале 60-х годов, она реально набрала популярность в конце 60-х годов.

Обычно с использованием виртуальной памяти решаются **две задачи:**

Во-первых, виртуальная память позволяет использовать адресное пространство, гораздо большее, чем емкость физической, основной памяти конкретной вычислительной машины. Принцип локальности почти всегда действует для реальных программ. Он означает, что обычно нет потребности в том, чтобы программы и данные размещались в физической, основной памяти целиком. Эта возможность выполнения программы, которая лишь частично находится в основной памяти, обеспечивает ряд преимуществ:

- В этом случае программа не огранивается размером основной памяти, упрощается разработка программ, поскольку можно использовать большие

виртуальные адресные пространства, не заботясь о размере используемой памяти.

- Возможность частичного размещения программы или образа процесса в основной памяти и гибкого перераспределения памяти между разными процессами - позволяет разместить в основной памяти больше программ, что естественно увеличивает загрузку процессора или процессоров и общую пропускную способность системы, потому что ресурсы работают всё время.

- Поскольку программа перемещается по частям во внешнюю память из основной и обратно, то объем дискового ввода-вывода при использовании виртуальной памяти меньше, чем при свопинге, когда образы процессов откачиваются и подкачиваются целиком, и поэтому каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы видимости практически неограниченной адресуемой пользовательской памяти при наличии основной памяти существенно меньших размеров - является очень важным аспектом операционных систем и вычислительной техники вообще. Это касается и современный громадных адресных пространств для 64-разрядных процессоров, но это было не менее важно, когда появились 32-разрядные компьютеры.

Введение виртуальной памяти позволяет решать и другую задачу, которая не менее важна для использования в действительности компьютеров в многопользовательском режиме - это обеспечение контроля доступа к отдельным сегментам памяти, в частности **защита пользовательских программ или процессов** один от другого, **защита операционной системы** от пользовательских программ. Для этих целей виртуальная память поддерживалась и на компьютерах с маленькой длиной адреса, на 16-разрядных компьютерах, в которых объем основной памяти (по крайней мере, на старших моделях) значительно превышал размер адресного пространства в 64 Кбайт (размер виртуальной памяти). Например, для 16-разрядных компьютеров PDP-11/70 имелись установки, которые поддерживали до 2 Мбайт основной памяти. Тем не менее операционной системой этого компьютера поддерживалась виртуальная память, основным смыслом которой было **обеспечение защиты и перераспределения** основной памяти между пользовательскими процессами. В действительности, конечно, виртуальная память на PDP-11/70 была достаточно убогой, там имелось всего 8 сегментов в памяти, говорить о том, что это была реальная виртуальная память - довольно трудно. Это была защищенная, сегментированная память. Конечно же, интерес к виртуальной памяти появился по-настоящему, когда возникли 32-разрядные компьютеры.

В системах с виртуальной памятью адреса, которые генерируют команды, выполняемые в программе, называются виртуальными адресами. Они формируют виртуальное адресное пространство. Если механизмы виртуальной памяти не

поддерживаются, то виртуальное адресное пространство, которое формируется при выполнении программы, непосредственно отображается в физическое пространство. Тогда физическое и виртуальное пространство - это одно и то же. В действительности известны и чисто программные реализации виртуальной памяти, например, виртуальная память на программном уровне реализовывалась в одной из операционных систем для БЭСМ-6, которая делалась в Институте прикладной математики им. М.В. Келдыша Российской академии наук. Там реализовывался длинный адрес на программном уровне, виртуальные адреса отображались физически чисто программным путем, но все это работало очень медленно, поэтому больше представляло академический интерес. На самом деле, широкое развитие технологии виртуальной памяти, конечно, началось после того, как возникла значительная аппаратная поддержка. Идея аппаратуры, которая поддерживает механизмы виртуальной памяти, состоит в том, что адрес памяти, который вырабатывается командой, интерпретируется аппаратурой не как реальный адрес некоторого элемента основной памяти, а как некоторая структура, в которой адрес является одним из компонентов наряду с атрибутами, которые характеризуют способ обращения по этому адресу.

Традиционно считается, что имеется **три модели виртуальной памяти**:

- **страничная**, когда вся память делится на блоки одного и того же размера, которые называются страницами;
- **сегментная**, когда виртуальная память состоит из сегментов разного размера;
- их комбинация - **сегментно-страничная** модель виртуальной памяти, в которой имеются сегменты, но каждый сегмент состоит из страниц одного и того же размера.

На большинстве платформ существует и аппаратно поддерживается страничная модель виртуальной памяти. Сегментно-страничная модель - это объединение идеи страничной модели и идеи сегментации. В вычислительных системах, в которых сегменты не поддерживаются на аппаратном уровне, они реализуются архитектурно-машинно-независимым компонентом менеджера памяти. Реализуется именно в операционных системах категории Unix, в том числе и в Linux. интересно, что Linux использует машинно-независимую реализацию сегментно-страничной виртуальной памяти даже на тех архитектурах, где есть поддержка сегментов (он не использует аппаратную поддержку). Сегментная организация в чистом виде в настоящее время практически не встречается. Она была на компьютерах семейства PDP-11/70 в свое время.

Средства поддержки

Для того, чтобы виртуальная память работала с приемлемой скоростью, необходимо, чтобы её механизмы поддерживались аппаратурой, иначе все получается

очень медленно. Поэтому полностью машинно-независимый компонент управления виртуальной памятью, который бы совсем не использовал никаких машинно-зависимых средств, - просто невозможен. Но имеются значительные части программного обеспечения, связанного с управлением виртуальной памятью, для которых неважны детали аппаратной реализации, аппаратной поддержки. Достижением современных операционных систем является аккуратное, грамотное и эффективное разделение средств управления виртуальной памяти на машинно-независимую и машинно-зависимую части. В современных операционных системах - это про операционные системы Unix, потому что в действительности ни в каких других операционных системах никогда ничего подобного не делалось. Стоит задуматься, зачем городить подобное, например, Microsoft, если они исторически работают на компьютерах одной и той же архитектуры. Им просто нет необходимости это делать. Unix это было необходимо, потому что главное достоинство операционной системы Unix - это высокая степень мобильности, возможность использования одной и той же операционной системы на разных аппаратных платформах, а также простой перенос системы с одной аппаратной платформы на другую.

Аппаратно-машинно-зависимая часть подсистемы управления виртуальной памяти. В этой лекции мы обсудим, аппаратно-машинно-зависимую часть подсистемы управления виртуальной памяти, что в неё входит и каким образом поддерживается аппаратурой. Как устроена машинно-независимая часть этой подсистемы, будет рассмотрено в следующей лекции.

Итак, имеется большое (для 32-разрядных архитектур это обычно 2**32 = 4 Гб) виртуальное адресное пространство и физическое пространство существенно меньшего размера. Пользовательский процесс или операционная система должны иметь возможность производить запись или считывать по указанному виртуальному адресу. При этом задача операционной системы - сделать так, чтобы записываемые данные оказались в физической основной памяти, если впоследствии её будет не хватать, чтобы данные были надежно сохранены во внешней памяти. Поэтому важным компонентом менеджера виртуальной памяти является система или **функция отображения (трансляции) адресов**, преобразование виртуальных адресов в физические адреса основной памяти. Для того, чтобы такое преобразование можно было осуществлять, необходимо иметь таблицу, которая показывает, какие области виртуальной памяти в каждый момент времени находятся в основной памяти и где эти области в ней располагаются. Элементарной единицей для которой производится адресация в компьютерах - является байт. Теоретически было бы можно делать отображение по байтам, но если бы это делалось таким образом, то таблица была бы недопустимо велика. Для её хранения потребовалось бы больше реальной памяти, чем для поддержки самих процессов. Поэтому необходим способ, который позволяет значительно сократить объем данных, с помощью которых производится отображение. В этом состоит одна из причин, по которым отображаемые данные группируются в

блоки одного и того же размера, которые в контексте виртуальной памяти называются страницы/ page.

Страничная память

В наиболее простом и наиболее часто используемом случае страничной виртуальной памяти виртуальная память и основная память представляются состоящими из наборов блоков или страниц одного и того же размера. В этом случае виртуальные адреса делятся на **страницы - page**. Соответствующие единицы в основной памяти, то есть кусочки основной памяти равные размеру страницы называются **страничными кадрами - page frames**. Система поддержки страничной виртуальной памяти в целом называется **пейджингом - paging** (листанием). Она называется листанием, потому что можно представить, что любой процесс как бы "перелистывает" свою виртуальную память, чтобы найти то место, куда ему надо обратиться. Когда обращается, он неявно запрашивает у операционной системы нужную страницу, поэтому иногда эту схему организации виртуальной памяти называют paging **by demand/листание по требованию**. То есть процесс, которому необходим доступ к какой-то странице виртуальной памяти, требует через внутреннее прерывание у операционной системы предоставить ему в это место виртуальной памяти физическую страницу основной памяти. Поскольку вся виртуальная память состоит из страниц, передача данных между основной памятью и внешней памятью, в которой сохраняются копии страниц основной памяти, всегда делается целыми страницами - блоками одного и того же размера.

Страницы, в отличие от сегментов (про которые мы говорили на прошлой лекции), имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Виртуальный адрес в страничной системе виртуальной памяти - это упорядоченная пара (p, d), где p - это номер страницы в виртуальной памяти, а d - смещение внутри страницы p, по которому размещается адресуемый элемент памяти. Если процесс обращается по некоторому виртуальному адресу, то ему разрешается продолжать выполняться, если его текущая страница находится в основной памяти, если p в данный момент времени реально отображается на страницу физической основной памяти. Если текущей страницы в основной памяти нет, то задача операционной системы - предоставить доступ к этой странице, которая должна быть либо переписана - то есть подкачана из внешней памяти, если она уже существовала в виртуальной памяти этого процесса, либо образована - если её там раньше не было. Страница, которую запрашивает процесс, может быть расположена в любом свободном страничном кадре основной памяти.

Вся система отображения виртуальных адресов в физические - сводится к системе отображения виртуальных страниц в физические и представляет собой **таблицу страниц - page table**. Для преобразования адресного пространства каждого процесса используется одна или несколько таблиц страниц, которые обычно хранятся в основной памяти. Для ссылки на таблицу страниц используется специальный регистр

процессора, который и определяет виртуальные адресные пространства того процесса, который в настоящее время выполняется на процессоре.

Выполняемый процесс обращается по виртуальному адресу v = (p,d). Механизм отображения ищет номер страницы p в таблице отображения и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d, смещение остается таким, какое указано в адресе. Этот адрес объявляется физическим адресом соответствующего элемента памяти. Таким образом мы попадаем на то место, куда требуется процессу.

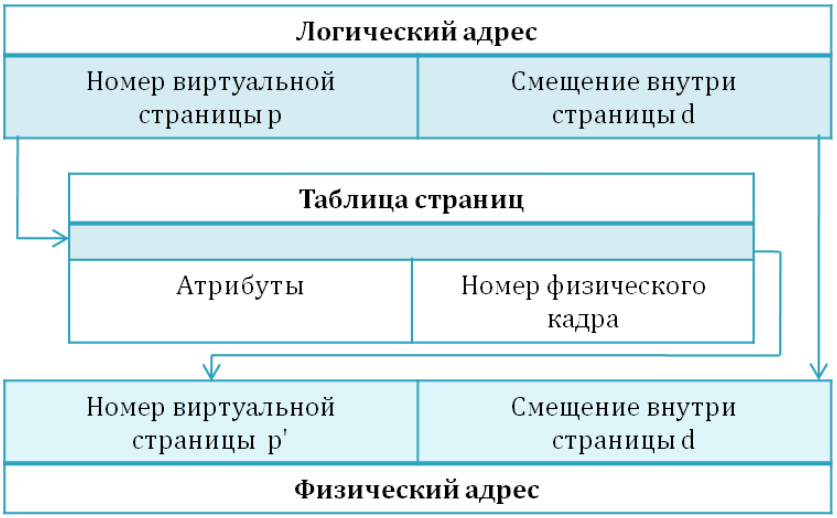

Рис. 9.1. Связь логического и физического адресов при страничной организации памяти

Например, если выполняется машинная команда записи с некоторого регистра по нулевому адресу, то адрес 0 находится на первой виртуальной странице. В действительности аппаратуре необходимо обратиться к первому элементу таблицы страниц. Реально она может быть расположена во втором страничном кадре В реальных системах функция отображения сохраняет бит присутствия страницы в физической памяти. При установлении реального адреса аппаратура проверяет - стоит ли бит нахождения страницы в основной памяти. Если там стоит бит, то все работает как на рассмотренной ранее схеме. Если нет, то возникает внутреннее прерывание - **исключительная ситуация**, которая специальным образом интерпретируется операционной системой. Она называется **страничным нарушением - page fault** или прерыванием по отсутствию страницы виртуальной памяти.

При обработке страничного нарушения страница выделяется в свободный кадр физической памяти. Если необходимо, то копия страницы подкачивается из внешней памяти, после чего продолжается прерванный код. Это называется by demand, то есть в действительности страничное нарушение - это и есть требование предоставить доступ к

основной физической памяти. Если при обработке page fault оказывается, что свободных страничных кадров в основной памяти нет, то система выполняет замещение страницы, то есть выгружает на диск страницу, которая по мнению операционной системы менее всего необходима в данный момент. Это наиболее сложное решение операционной системы, про него будет следующая лекция, где мы рассмотрим, как это делается. Номер страницы служит индексом в таблице страниц - page tabl, на которую указывает специальный регистр процессора, позволяя определить номер кадра основной памяти по виртуальному адресу. Атрибуты, содержащиеся в строке таблицы страниц, используются для того, чтобы аппаратуре было можно контролировать доступ к соответствующей странице и по необходимости производить ее защиту.

Основным преимуществом схемы чистого пейджинга, то есть чисто страничной организации виртуальной памяти - является отсутствие **внешней фрагментации**. Именно поэтому обычные страницы выбираются в степени числа 2, чтобы было можно легко распределять память. Это делается чрезвычайно просто: с помощью единой шкалы, которая говорит, какие кадры свободны, а какие заняты. Однако при использовании страниц, как и при использовании разделов фиксированного размера может возникать **внутренняя фрагментация**. Если память всегда запрашивается в виде кадра, страницы, то в среднем может пропадать половина страницы каждой страницы на процесс. Здесь важным аспектом является различие точек зрения пользователя и системы на то, что такое есть виртуальная память. С пользовательской точки зрения пользователя виртуальная память - это непрерывное адресное пространство, которое содержит только одну скомпонованную программу. Реальное отображение на странице от пользователя скрыто и контролируется операционной системой. Процессу пользователя недоступна чужая память, он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы. Для управления физической памятью операционная система поддерживает таблицу кадров, которая содержит одну запись на каждый физический кадр, показывающий его состояние. Можно в действительности свести к шкале, потому что необходимо хранить один бит - свободно или занято (как правило, так и делается).

Отображение должно происходить корректно во всех случаях, даже в сложных. Для этого операционная система поддерживает одну таблицу страниц для виртуальной памяти каждого процесса или несколько таблиц страниц, если виртуальная память разбивается на сегменты. При использовании одной таблицы страниц для ссылки на нее обычно используется специальный регистр. При переключении процессов диспетчер (менеджер виртуальной памяти) должен найти таблицу страниц для того процесса, на который происходит переключение. Тем самым указатель (ссылка на неё) входит в контекст процесса.

В большинстве современных компьютеров со страничной организацией виртуальной памяти все таблицы страниц хранятся в основной памяти. В

действительности это решение появилось не сразу, пока адресные пространства были маленькими (например, в PDP-11 - 16 разрядов или в БЭСМ - 6 - 15 разрядов) таблицы страниц хранились на специальных аппаратных регистрах. То есть смена контекста, смена виртуальной памяти заключалась в том, что было необходимо перезапустить регистры приписки. Конечно, такой доступ к виртуальной памяти был очень быстрым, но регистры - есть регистры, то есть на самом деле обращение к регистрам приписки было настолько же быстрым, как ко всем универсальным или индексным регистрам. Проблема стала острой, когда машины стали 32-разрядными. В этом случае размер таблицы приписки становился потенциально настолько большим, что регистров было для этого недостаточно. Решение, которое было придумано (это было впервые сделано компаний DEC, когда появилось семейство VAX-11) - это введение сверх быстродействующей буферной памяти - кэша.

Кэш - это набор блоков с ассоциативной адресацией. Таблицы страниц, таблицы приписки при первом обращении к виртуальной памяти в данном процессе работают через основную память, но при этом соответствующий кусочек таблицы приписки (64 байта) перемещался в соответствующий блок кэша. В дальнейшем доступ к частям виртуальной памяти, которые уже использовались в этом процессе, производился со скоростью кэша, то есть со скоростью тех же регистров. Доступ к кэшу - ассоциативный, то есть производился параллельный поиск во всех блоках кэша, в которых хранились части таблицы приписки, чтобы найти нужный кусочек. Это решение действительно стандартное, оно применяется последние 40-50 лет во всех компьютерах, в которых поддерживается кэш.

Сегментная и сегментно-страничная виртуальная память

Поговорим про другие модели организации виртуальной памяти, а конкретно про сегментную и сегментно-страничную организации. Сами идеи сегментации мы рассматривали в предыдущей лекции. Напомню, что первым поводом для того, чтобы память можно было сделать сегментной, а один и тот же сегмент мог входить в логическое пространство разных процессов - состоял в том, чтобы можно было не дублировать неизменяемые части программ, в которых находится программный код. Отсюда появился сегмент программного кода, потом уже появились идеи, как можно сделать другие сегменты. При сегментной организации виртуальный адрес адресного пространства по-прежнему является двумерным и состоит из двух полей: номера сегмента и смещения внутри сегмента. С точки зрения операционной системы сегменты - это логические сущности. Их основное назначение - это хранение и защита доступа к однородной информации (такой, как код, данные, стек, сегменты разделяемой памяти и т.д.).

С точки зрения пользователя сегментированная виртуальная память процесса представляется не как линейный массив байтов, а как набор сегментов переменного размера (данные, код, стек). Сегментация - это схема управления памятью, которая поддерживает этот взгляд пользователя. Обычно сегменты содержат именно

однородную информацию: машинный код, статические переменные, стек, процедуры, массивы, но не содержат данные смешанного типа. Для того, чтобы был смысл, необходимо, чтобы была однородная информация.

Логическое адресное пространство представляет собой набор сегментов. У каждого сегмента имеются имя или номер, размер и другие параметры, такие как уровень привилегий для доступа к этому сегменту, разрешенные виды обращений, флаги присутствия или отсутствия сегмента в основной памяти и т.д. Пользовательская программа специфицирует каждый адрес двумя значениями: номером сегмента и смещением. Это отличается от схемы страничной организации, от схемы пейджинга, в которой пользователь задает только один адрес, а аппаратура сама разбивает его на номер страницы и смещение таким способом, который программисту, вообще говоря, знать необязательно. Каждому сегменту соответствует линейная последовательность адресов: от нулевого адреса, до некоторого максимального, который соответствует сегменту в данный момент времени. Разные сегменты могут иметь разные длины, которые могут меняться динамически, например, сегмент стека может расти. В каждом элементе таблицы сегментов, которые используются для отображения виртуального адреса на физический, кроме физического адреса начала сегмента (если соответствующий виртуальный сегмент содержится в основной памяти) - ещё находится размер сегмента. Если при обращении по виртуальному адресу размер смещения в выходит за пределы текущего размера сегмента, то возникает прерывание. Это прерывание может интерпретироваться как ошибка, если сегмент не расширяется, как, например, выход за пределы программного сегмента у кода - это всегда ошибка. Или это может приводить к тому, что сегмент динамически расширяется, как это происходит, например, для сегмента стека.

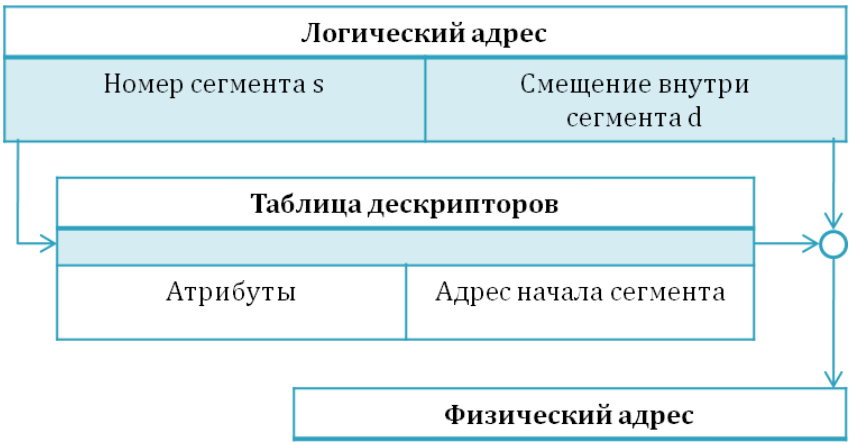

Рис. 9.2. Преобразование логического адреса при сегментной организации памяти

Логический адрес - это упорядоченная пара v = (s, d), где s - это номер сегмента, d - это смещение внутри сегмента. При наличии аппаратной поддержки для

работы с сегментами поддерживается таблица дескрипторов или описателей сегментов, а программа обращается к дескрипторам по номерам селекторам. Тогда контекст процесса включает набор сегментных регистров, которые содержат селекторы текущих сегментов кода, стека, данных и т.д. Они определяют, какие сегменты будут использоваться при разных видах обращений к памяти. На аппаратном уровне определяется допустимость обращений к памяти по атрибутам, которые содержатся в описателях.

Необходимо отметить, что сегментная организация виртуальной памяти в чистом виде в основном не используется. Это происходит из-за того, что она приводит к тому, что необходимо выделять в основной памяти куски произвольного размера, которые соответствуют требуемому размеру сегмента. Как мы это наблюдали в связи с распределением памяти разделами изменяемого размера - это приводит к внешней фрагментации, то есть основная память дробится на кусочки, включающие много мелких пустых фрагментов. Как всегда, время от времени требуется выполнять процедуру сжатия основной памяти. Это очень неудобно, долго и громоздко для операционной системы и для пользователей вычислительной системы. Аппаратная поддержка сегментов распространена относительно слабо (главным образом, на процессорах архитектуры Intel), она характеризуется довольно медленной загрузкой селекторов в сегментные регистры, выполняемой при каждом переключении контекста и при каждом переходе между разными сегментами.

В системах со страничной организацией памяти сегментация реализуется на независимом от аппаратуры уровне. Из-за этого возникает идея пейджинга самих сегментов, что приводит **сегментно-страничной организации виртуальной памяти.**

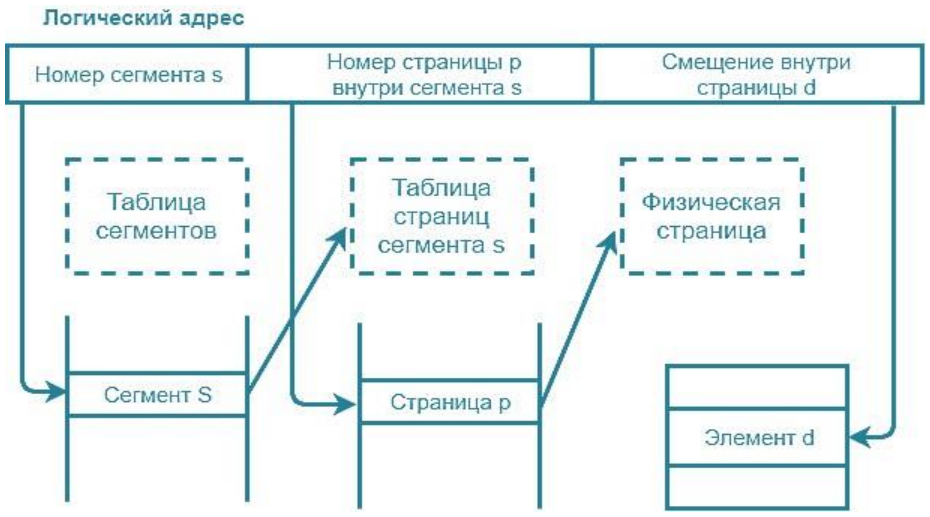

Рис. 9.3. Упрощенная схема формирования физического адреса при сегментно-страничной организации памяти

В этом случае виртуальный адрес состоит уже из трех полей: номера сегмента виртуальной памяти, номера страницы внутри сегмента и смещения внутри страницы. Преобразование виртуального адреса в физический становится двухуровневым, используются две таблицы отображения: таблица сегментов, которая связывает номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента. В действительности внутренняя фрагментация может возникать в этом случае только в последней странице сегмента, потому что если сегмент выделяется как набор страниц, то все страницы (кроме последней) заполняются плотно.

Таблица страниц

Давайте рассмотрим на более детальном уровне, как выглядит структура таблиц. Мы будем говорить именно про страничную организацию памяти. Виртуальный адрес состоит из **виртуального номера страницы - high-order bits** (это старшие биты памяти) и **смещения - low-order bits** (нижние биты адреса, младшие биты адреса). Номер виртуальной страницы используется как индекс в таблице страниц для нахождения описателя соответствующей виртуальной страницы - это **entry,** элемент таблицы страниц. Из этой записи, если страница находится в основной памяти, берется **номер кадра - page frame number**, к нему приписывается смещение, и тем самым формируется физический адрес. Кроме этого, элемент таблицы страниц содержит атрибуты страницы, в частности, биты защиты.

Основная проблема для эффективной реализации таблицы страниц состоит в том, что в современных компьютерах виртуальные адресные пространства обладают большими размерами в соответствии с разрядностью архитектуры процессора. Например, даже в 32-разрядных процессорах виртуальные адресные пространства могут составлять 4 Гб, для 64-разрядных (это астрономическое значение) компьютеров эта величина равна 2**64 байт. Какие при этом могут быть размеры таблицы страниц? В 32-разрядном адресном пространстве, если размер страницы 4К (это считается достаточно большим размером страницы) - получаем 1миллион страниц, а в 64-разрядном адресном пространстве и того более. Тем самым, таблица для 32-разрядной адресации должна содержать миллион записей, причем каждая запись состоит из нескольких байт. При страничной организации памяти каждому процессу требуется своя таблица страниц, а в случае сегментно-страничной организации - требуется таблица страниц для каждого сегмента. В любом случае таблица страниц может быть слишком большой. Кроме того, отображение должно быть быстрым, поскольку оно требуется при каждом обращении к памяти, это происходит практически в каждой машинной команде. Главным образом, как мы рассматривали ранее, проблема решается за счет использования кэша - ассоциативной памяти. Кроме того, чтобы избежать необходимости всё время хранить такую громадную таблицу в основной памяти, и чтобы было можно хранить только несколько ее фрагментов (это, опять же, возможно на основании принципа локальности) - во многих вычислительных системах используются **многоуровневые таблицы страниц**.

Рассмотрим рисунок 9.4., на котором показано, как может быть устроена многоуровневая таблица страниц: предположим, что 32-разрядный адрес делится на три части: 10-разрядное поле Ptr1, 10-разрядное поле Ptr2 и 12-разрядное смещение Offset.

- 12 разрядов смещения позволяют указывать относительную позицию байта внутри страницы размером 4К;
- 1024 строки в таблице верхнего уровня с использованием поля Ptr1 - ссылаются на 1024 таблицы второго уровня, каждая из которых тоже содержит 1024 строки;
- При помощи содержимого поля Ptr2 каждая строка таблицы второго уровня указывает на конкретную физическую страницу.

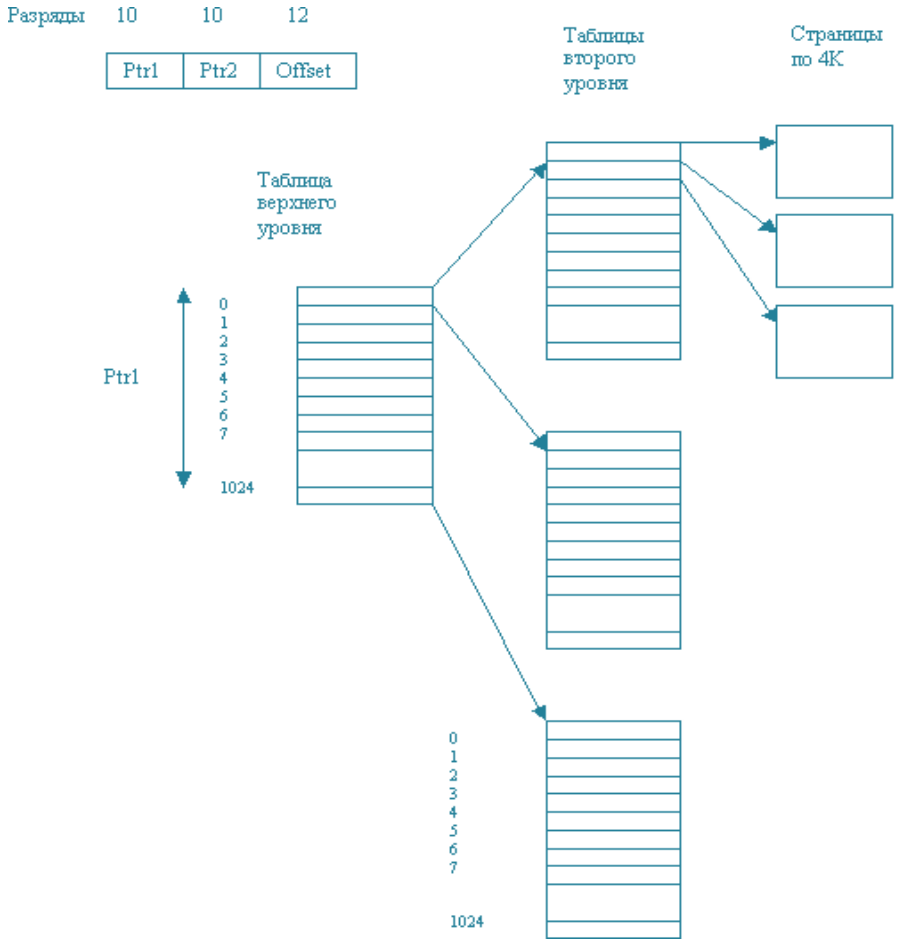

Рис. 9.4. Устройство многоуровневой таблицы страниц

Это позволяет избежать постоянной поддержки в основной памяти всех таблиц второго уровня, то есть в этом случае (если таблица второго уровня для какого-то элемента таблицы первого уровня в основной памяти отсутствует) в соответствующем

элементе таблицы первого уровня ставится бит прерывания, система обрабатывает этот бит и загружает в основную память (если она уже существует) ту часть таблицы второго уровня, которая сейчас требуется для обращения к памяти.

Рассмотрим пример с некоторыми круглыми цифрами. Пусть процессу требуется всего 12 Мб памяти: 4 Мб в нижней части памяти со старшими адресами для программного кода, 4 Мб в нижней части для данных (там же) и 4 Мб в верхней части памяти для стека. Между дном стека (то есть концом стека) и верхом данных (началом) имеется гигантское неиспользуемое пространство размером 4 Гб - 12 Мб. Для этого случая требуются одна таблица верхнего уровня и три таблицы второго уровня. Этот подход можно обобщить на три (или более) уровней таблицы. В действительности это достаточно экономно.

Размер записи таблицы страниц от системы к системе может быть разным, но наиболее частый случай - это 32. Самое важное поле элемента таблицы страниц - это номер физического кадра. Целью страничного отображения является именно нахождение этого значения. Кроме того, в описателе страницы содержатся:

- бит наличия страницы в основной памяти;
- биты защиты, то есть что можно делать с этой страницей: 0 - read/write, читать и писать, 1 - read only, только читать, и т.д.;
- биты модификации страницы, которые говорят, что эта страница изменена, то есть туда происходила запись;
- биты, которые показывают, что к странице было обращение;
- биты, которые разрешают или запрещают кэширование соответствующих данных.

Если стоит бит, что страница в основной памяти отсутствует, то остальное содержимое элемента таблицы страниц аппаратуру уже не задевает, то есть там обычно храниться адрес страницы на диске, но это не считается частью таблицы страниц, потому что аппаратура с этим уже не работает, это дело операционной системы.

Если предположить, что для каждого уровня поддерживается отдельная таблица в памяти, то для преобразования адреса может потребоваться несколько обращений к основной памяти. Число уровней в таблице страниц зависит от особенностей архитектуры: например, в DEC PDP-11 была одноуровневая реализация, в Intel и DEC VAX - двухуровневая, в процессорах Sun SPARC и DEC Alpha (последнем процессоре, который был сделан компанией DEC пока она была независимой) - трехуровневая, в процессорах компании Motorola - можно было задавать число уровней, в RISC-процессоре MIPS R2000 вообще не поддерживалась таблица страниц, это делалось на программном уровне. Поиск нужной страницы, если эта страница отсутствует в

ассоциативной памяти, в этом случае должна взять на себя операционная система - это так называемый **zero level paging.**

Ассоциативная память

Поиск нужной страницы в многоуровневой таблице страниц, требующий несколько обращений к основной памяти при преобразовании виртуального адреса к физическому, занимает много времени. Это непродуктивное время, в ряде случаев такая задержка недопустима. Эта проблема также решается на уровне аппаратуры. В соответствии с принципом локальности большая часть программ в течение некоторого небольшого промежутка времени обращается к небольшому числу страниц, поэтому интенсивно используется только небольшая часть таблицы страниц. Естественное решение - это снабдить компьютер устройством для отображения виртуальных адресов в физические без обращения к таблице страниц. Требуется небольшая и быстрая кэш-память, которая хранит нужную в данный момент часть таблицы страниц. Такие устройства называются **ассоциативной памятью**, иногда их называют **ассоциативными регистрами** или **translation lookaside buffer, TLB**. В этом случае одна запись в TLB содержит информацию про одну виртуальную страницу, поля записи соответствуют полям таблицы страниц. Отображение виртуальных страниц, данные о которых хранятся в TLB, производится быстро, но кэш-память является дорогостоящей и обычно из-за этого имеет ограниченный размер, например, число записей в TLB может ограничиваться от 8 до 2048. Память называется ассоциативной, поскольку происходит одновременное сравнение номера виртуальной страницы с соответствующим полем - одновременно во всех строках TLB. Поэтому, собственно, она и дорогая, поддержка ассоциативного доступа - это дорого. В строке, в которой поле виртуальной страницы совпадает с искомым значением, находится номер страничного кадра.

Если поддерживается ассоциативная память (TLB), то вначале номер виртуальной страницы ищется в ассоциативной памяти. Если там находится соответствующий виртуальный адрес, то все нормально, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется, то есть в действительности возникает прерывание по ошибке, исключительная ситуация. Если описателя страницы в ассоциативной памяти нет, то происходит обращение к таблице страниц в основной памяти, и аппаратура заменяет одну из записей TLB записью о найденной странице. Понятно, что в этом случае аппаратура должна принимать решение о том, какую запись заменить. Какие критерии используются - сказать затруднительно, потому что они разные в разных архитектурах. Отсюда видно, что когда меняется контекст процесса, то в действительности TLB пустая. Операционная система оттуда должна все сбросить. При начале работы любого процесса, все обращения к виртуальной памяти происходят через страницу таблиц, которые находятся в основной памяти. Поэтому, после смены контекста, каждый процесс

некоторое время ведет себя очень медленно, пока не заполнится ассоциативна память, а обращения не будут происходить через TLB.

Число удачных поисков номера страницы в TLB по отношению к обычному числу поисков называется **hit ratio** (hit - совпадение, ratio - пропорция, отношение). Это число успехов, то есть hit ratio - это часть ссылок, часть преобразований виртуального адреса в физический, которую можно сделать с использованием ассоциативной памяти. Если процесс обращается к одним и тем же страницам, то это повышает hit ratio. Если предположить, например, что для доступа к таблице страниц требуется 100 наносекунд (нс), а для доступа к ассоциативной памяти - 20 нс, то при 90% hit ratio мы имеем среднее время доступа - 0.9*20+0.1*100 = 28 нс. То есть наличие такой ассоциативной памяти сильно повышает среднее время доступа к страницам виртуальной памяти.

То, что современные операционные системы на современной аппаратуре работаю достаточно производительно - показывает, что ассоциативная память используется достаточно эффективно. Высокое значение hit ratio связано с наличием у данных объективных свойств: пространственной и временной локальности. При переключении процессов необходимо, чтобы новый процесс не находил в ассоциативной памяти данных, которые относятся к предыдущему процессу, например, необходимо полностью очищать TLB, поэтому использование ассоциативной памяти увеличивает время переключения контекстов. На сколько ассоциативная память работает при multithread организации процессов - сказать трудно, потому что принцип локальности работает для каждого thread. Если мы меняем контекст thread внутри одного и того же адресного пространства, то очень может быть, что в новом thread на процессоре не будет использоваться старая TLB, а всё время будет нулевой hit ratio, мы все время будем обращаться через таблицу приписки в основной памяти. Это довольно тонкие моменты, про которые трудно сказать, как они работают в общем случае.

Инвертированная таблица страниц

Несмотря на возможность использования многоуровневой организации таблицы страниц, проблему представляет использование таблиц большого объема. Это особенно актуально для 64-разрядных архитектур. Одним из вариантов решения является применение **инвертированных таблиц страниц - inverted page table, IPT**. Они применяются на процессорах семейства Power PC, на некоторых моделях Hewlett-Packard, IBM RT и AS/400. В этом случае имеется всего одна таблица, она содержит по одной записи на каждый страничный кадр основной памяти. Одной таблицы достаточно для всех процессов. Для её хранения требуется фиксированная часть основной памяти (независимо от разрядности и числа процессоров). Например, для старых Pentium, в которых было всего 256 Мб основной памяти, требовалась всего одна инвертированная таблица размером в 64 тысячи строк. Понятно, что размер увеличивается в зависимости от того, сколько реально физической основной памяти подключено к компьютеру.

Применение IPT обладает одним существенным минусом - в этом случае записи не отсортированы по возрастанию номеров виртуальных страниц, что усложняет поиск. То есть в действительности аппаратура должна каким-то образом искать внутри инвертированной таблицы страниц элемент по адресу. Хотелось бы использовать ассоциативный доступ, но это - основная память, так что - нельзя. Один из способов решения проблемы - это использовать хэш-таблицу виртуальных адресов. В этом случае номер виртуальной страницы отображается в хэш-таблицу с использованием хэш-функции. Каждой странице основной памяти соответствует одна запись в хэш-таблице и в IPT. Если возникают коллизии, то есть виртуальные адреса могут выдавать одно и тоже значение свертки, то они сцепляются в список, который обычно не длиннее двух записей. Обратим внимание, что в действительности это должна делать аппаратура.

Какой может быть размер страниц? Разработчики операционных систем для существующих компьютеров редко могут как-то влиять на размер страницы. Однако для новых компьютеров решение относительно оптимального размера страницы является актуальным. Одного наилучшего размера нет - есть набор факторов, которые влияют на размер. Обычно размер страницы - это степень двойки от 2**9 (512 байт) до 2**14 байт (4 096 байт). Чем больше размер страницы, тем меньше будет размер структур данных, которые необходимы для того, чтобы преобразовывать адреса, но тем больше будут потери, связанные с внутренней фрагментацией.

Как необходимо выбирать размер страницы? Во-первых, нужно учитывать размер таблицы страниц. Здесь желательно иметь как можно больший размеры страницы, то есть чем меньше страниц, тем таблица страниц меньше. С другой стороны, коэффициент использования основной памяти улучшается при уменьшении размера страниц, поскольку в среднем при чисто страничной организации пропадает половина страницы, при сегментно-страничной организации - половина последней страницы процесса. Кроме того, необходимо также учитывать объем ввода-вывода для взаимодействия с внешней памятью и другие факторы. Совсем хорошего ответа здесь нет, историческая тенденция состоит в том, что размер страницы возрастает. Здесь необходимо смотреть: сколько стоит основная память, насколько долго выполняется обмен с внешней памятью и т.д., чтобы можно было хоть как-то прикинуть, какого размера страницу необходимо в действительности использовать.

Как правило, размер страницы задается на аппаратном уровне. В PDP-11 – это 8 Кб (максимальный размер сегмента 8 Кб), в VAX-11 – это 512 байт. В некоторых архитектурах размер страницы может задаваться программно, например, в Motorola 68030, в Windows размер страницы колеблется от 4 до 8 Кб. В большинстве коммерческих операционных систем поддерживается только один размер страниц, который, как правило, совпадает с размером блока внешней памяти или кратен блоку памяти.

Заключение:

- В этой лекции мы рассмотрели аппаратные особенности поддержки виртуальной памяти.
- Разбиение адресного пространства процесса на части и динамическая трансляция адреса позволяют поддерживать выполнение процесса даже при отсутствии некоторых его компонентов в основной памяти.
- Подкачка недостающих компонентов из внешней памяти производится только при необходимости
- Результатом является возможность выполнения больших программ, у которых размер может превышать размер основной памяти.
- Для обеспечения необходимой производительности отображение адресов производится на аппаратном уровне с использованием многоуровневых таблиц страниц и ассоциативной памяти (кэша).

На следующей лекции мы будем рассматривать вопросы программной поддержки виртуальной памяти, в частности очень важным является вопрос о замещении страниц основной памяти в условиях, когда её недостаточно для предоставления необходимых объемов всем выполняемым процессам.

Лекция 10. Управление виртуальной памятью в операционных системах

Исключительные ситуации при работе с памятью

На этой лекции, которая будет последней лекцией, посвященной управлению памятью в операционных системах, мы поговорим о том, как реально происходит управление виртуальной памятью. Сначала мы рассмотрим исключительные ситуации при работе с виртуальной памятью; поговорим про стратегии управления страничной виртуальной памятью; видимо, самая интересная часть этой лекции - это алгоритмы замещения страниц; потом мы рассмотрим особую ситуацию, которая возникает при управлении виртуальной памятью - thrashing, что можно делать с этой нехорошей ситуацией, свойства локальности, модель рабочего набора; демоны пейджинга; наконец, мы обсудим, как устроена аппаратно-независимая модель управления виртуальной памятью в операционной системе Unix; отдельно немного поговорим о некоторых аспектах работы менеджера виртуальной памяти.

Обычно, в случае операционной системы Unix - операционные системы опираются на некоторые собственные представления об организации виртуальной памяти. Это представление используется в аппаратно-независимой части подсистемы управления виртуальной памятью. Эта аппаратно-независимая часть связывается с конкретной аппаратной реализацией, поддержкой аппаратной виртуальной памяти с помощью аппаратно-зависимой части. Как достигается возможность поддержки виртуальной памяти, размер которой значительно больше, чем размер доступной основной памяти? Мы помним, как в общем устроена таблица страниц: там может присутствовать специальный флаг, специальный признак, который означает отсутствие реальной страницы в основной памяти, который соответствует данной виртуальной странице. Наличие этого флага приводит к тому, что аппаратура вместо того, чтобы выполнить нормальное реальное отображение виртуального адреса в физический адрес основной памяти - прерывает выполнение соответствующей команды и передает управление соответствующему компоненту ядра операционной системы.

Когда программа обращается к виртуальной странице, которая отсутствует в основной памяти, то есть "требует" доступа к данным или программному коду - операционная система удовлетворяет это требование путем:

- путем выделения страницы оперативной памяти, свободного кадра основной памяти;
- путем перемещения (если это требуется) в неё копии страницы, находящейся во внешней виртуальной памяти;
- соответствующей модификации элемента таблицы страниц (после чего может быть продолжено выполнение команды).

Это требование происходит путем возникновения аппаратного прерывания - **исключительной ситуацией - exception,** поэтому соответствующий механизм управления виртуальной памятью называется **paging by demand**/листание по требованию, то есть прерывание. В действительности это прерывание по отсутствию станицы виртуальной памяти является частным случаем исключительной ситуации при работе с памятью, частным случаем внешнего прерывания, оно называется **страничным нарушением - page fault.**

Ключевая информация о характере отображения виртуального адресного пространства в физическое пространство основной памяти хранится в таблице страниц. Кроме сведений о наличии или отсутствии нужной страницы в основной памяти атрибуты соответствующей страницы могут разрешать или запрещать конкретные операции обращения к памяти. Что реально происходит, когда требуемая операция с виртуальной страницей запрещена или этой страницы нет в основной памяти?

- В этом случает ядро операционной системы должно быть как-то оповещено о происшедшем. Стандартным образом для этого используется механизм **исключительных ситуаций/exceptions** или **внутренних прерываний**.
- При попытке выполнить такого рода операцию - возникает исключительная ситуация - **страничное нарушение/page fault.**
- Это приводит к вызову специальной программы в ядре операционной системы - обработчика исключительной ситуации - **exceptions** handler, который решает, что реально необходимо предпринять, чтобы обработать конкретный вид страничного нарушения.

Страничное нарушение может происходить в разных случаях. Например, если мы запрещаем (а это, как правило, так) писать в ту часть памяти, которая содержит коды команд, то в этом случае соответственно у страницы стоит атрибут "только чтение" (страничное нарушение при попытке обращения); если происходят попытки чтения или записи страницы с атрибутом "только выполнение" - тоже возникает нарушение. В любом случае вызывается обработчик страничного нарушения, который является частью ядра операционной системы, ему передается причина возникновения исключительной ситуации и соответствующий виртуальный адрес.

Поскольку сейчас мы рассматриваем управление виртуальной памятью, то больше всего нам интересен вариант обращения странице, которая отсутствует в основной памяти. Именно обработка такого рода прерываний во многом определяет производительность системы со страничной виртуальной памятью. Понятно, что чем реже возникают такие страничные нарушения, чем быстрее они обрабатываются, тем более производительно работает вся система. Время эффективного доступа к странице виртуальной памяти, которая отсутствует в основной памяти, складывается из следующих компонентов:

1. Время обслуживания **исключительной ситуации - page fault,** это более-менее стандартное время, от которого никуда не денешься.
2. Время чтения **(подкачки)** страницы из внешней памяти. Иногда, если основная память заполнена, там нет свободных кадров, то необходимо освободить какую-нибудь из страниц основной памяти (возможно, вытолкнуть её копию во внешнюю память). Всё это называется - **замещение** страницы.
3. Время рестарта процесса, то есть возврата из прерывания процесса, который вызвал данные страничные нарушения - **page fault.**

Время выполнения первого и третьего компонентов можно сократить только за счет тщательного программирования соответствующего кода ядра операционной системы. Можно эти несколько сотен команд постараться сделать как можно более эффективными. Если необходимо обращаться к дисковой внешней памяти, то время подкачки страницы с диска все равно будет близким к нескольким десяткам миллисекунд (если это произвольный обмен). В действительности, если снизить вероятность страничного нарушения до $5*10^7$, то тем самым производительность страничной системы упадет всего на 10%. Таким образом, основной задачей системы управления памятью является обеспечение того, чтобы прерывания по отсутствию страниц виртуальной памяти в основной памяти возникали как можно реже. Для этого необходимо, чтобы в основной памяти находились те страницы виртуальной памяти всех процессов, которые сейчас одновременно присутствуют в системе, то есть для этого требуется правильный выбор алгоритма замещения страниц.

Стратегии управления страничной памятью

Обычно рассматривают три возможных стратегии:

1. **Стратегия выборки/fetch policy** - это стратегия, на основе которой операционная система решает - в какой момент необходимо поместить страницу, копия которой присутствует во внешней памяти, в основную память. Соответствующая выборка может быть **по запросу/by demand** или **с упреждением/preem fetch.** Алгоритм выборки действует, когда процесс обращается к отсутствующей в основной памяти странице, содержимое которой в данный момент находится на диске. Обычно действие алгоритма заключается в том, что страница с диска загружается в свободную физическую страницу виртуального адреса, обращение по которому вызвало исключительную ситуацию. Соответственно, устанавливается отображение на эту физическую страницу виртуального адреса, обращение по которому вызвало исключительную ситуацию. То, что мы сейчас рассмотрели - это by demand/по требованию, но существует модификация алгоритма выборки, при которой применяется чтение с упреждением. Кроме страницы, вызвавшей исключительную ситуацию, в этом случае в память одновременно загружается несколько страниц, которые её окружают (так называемый кластер). Такой

подход выборки с упреждением предназначен для того, чтобы было можно уменьшить накладные расходы, возникающие при обработке большого количеством исключительных ситуаций, которые возникают при работе программы с большими объемами данных или кода. Кроме того, такой подход оптимизирует и работу с внешней памятью, поскольку появляется возможность загрузки нескольких страниц за одно обращение к диску. Необходимо отметить, что полезность такого подхода достаточно сомнительна, потому что в действительности у операционной системы отсутствует какое-либо представление о том, что будет дальше делать программа, которая обратилась к странице отсутствующей виртуальной памяти. Очень вероятно, что те данные, которые находятся именно в этой странице, программе будут ещё необходимы, но насколько ей необходимы данные, которые находятся в соседних станицах - у операционной системы нет никакой информации. В действительности упреждающая выборка, конечно же, спекулятивная, то есть в расчете на лучшее, но посчитать - насколько это чему-либо поможет - невозможно. Это как раз тот случай, когда операционная система старается сделать что-то очень умное, но делает это в отсутствии какой-нибудь информации, которой можно обосновать то, что она делает.

2. **Стратегия размещения/placement policy** - эта стратегия определяет - в какое место основной памяти следует поместить страницу, которую мы хотим процессу дать, то есть ту страницу, куда мы хотим переписать копию страницы из внешней памяти. Если это виртуальная память со страничной организацией, то в этом случае годится любой свободный страничный кадр. Если это система с сегментной организацией, то в этом случае стратегия размещения связана с тем, что необходимо найти фрагмент основной памяти нужного размера, здесь ситуация очень похожа на то, как приходится работать при распределении памяти с разделами переменного размера. Мы помним, что там возникает внешняя фрагментация. Это в действительности (как все проблемы с сегментной организацией виртуальной памяти) требует достаточно больших накладных расходов на то, чтобы с этой внешней фрагментацией бороться. Например, можно применять что-то типа метода близнецов, то есть выделять память большего размера, чем требуется (равный степени числа 2) с тем, чтобы за счет возможной внутренней фрагментации минимизировать внешнюю фрагментацию. Но идеальных решений не бывает - там всё равно возникнет какая-то фрагментация.

3. **Стратегия замещения/replacement policy** - эту стратегию необходимо применять, когда нет свободной основной памяти, то есть необходимо освободить какую-то страницу основной памяти, то есть вытолкнуть её содержимое во внешнюю память.

Выбор конкретной стратегии очень важен для того, чтобы вся система управления виртуальной памятью работала эффективно. В действительности любая стратегия, которую можно считать разумной, позволяет оптимизировать хранение в основной памяти информации, наиболее необходимой для выполнения соответствующего процесса. Чем мы ближе к тому, что реально требуется процессу, тем меньше у него возникает прерываний по отсутствию страниц - страничных нарушений.

Алгоритмы замещения страниц

Алгоритмы замещения страниц - это действительно наиболее важная часть всей работы ядра операционной системы, связанной с управлением виртуальной памятью. Наиболее ответственным действием страничной системы управления виртуальной памятью является выделение страницы основной памяти для удовлетворения требования доступа некоторого промесса к отсутствующей в основной памяти виртуальной странице. Поскольку память виртуальная, то одно из её назначений - это обеспечение возможности программе работать с адресным пространством, размер которого существенно превосходит размер основной памяти. Поэтому с большой вероятностью (когда система загружена, когда в операционной системе одновременно поддерживается выполнение нескольких процессов) не удастся найти свободную страницу основной памяти. Тогда операционная система должна найти некоторую занятую страницу основной памяти (выбрать её по некоторым критериям), переместить (если это требуется) ее содержимое во внешнюю память. Соответствующим образом модифицировать элемент таблицы страниц, который раньше указывал на страницу, которую мы освобождаем, чтобы при попытке обращения к соответствующей странице виртуальной памяти, в следующий раз было прерывание по отсутствию страницы.

В общем случае при замещении страницы приходится дважды обмениваться с внешней памятью: первый раз, когда страницу необходимо освободить, её текущее содержимое надо записать на диск; второй раз, кода необходимо вернуть в кадр основной памяти то содержимое, которое когда-то было из основной памяти выбрано и сохранено в области подкачки. Процесс замещения может быть оптимизирован, правильнее сказать - алгоритмы замещения могут быть более умными, если соответствующая часть ядра использует бит модификации. Бит модификации - это один из атрибутов элемента таблицы страницы, который устанавливается в 1, если после того, как этот бит был обнулен, эта страница изменялась. Тогда при выборе кандидата на замещение проверяется бит модификации, если бит не установлен, то это означает, что соответствующая страница не изменялась с того времени, как она была помещена в внешнюю память. Аналогичная техника применяется к read-only страницам, доступ к которым разрешается только по чтению (страниц кода, например, или страниц, в которых хранятся какие-то константные значения).

Имеется большое число разных алгоритмов замещения страниц. В 70-80 -е годы эта тема была одной из наиболее "горячих" тем в области операционных систем.

Алгоритмы появлялись, если не каждую неделю, то каждый месяц.

Все алгоритмы замещения страниц делятся на два класса: **локальные и глобальные.**

- **Локальные алгоритмы** работают следующим образом: каждому процессу выделяется некоторое число страниц основной памяти. Число может быть фиксированным или динамически настраиваемым, но в любом случае, если процессу требуется страница основной памяти (то есть её необходимо выделить), то в качестве кандидата на замещение рассматриваются только те страницы, которые уже входят в виртуальную память этого процесса. То есть ядро в этом случае никогда не отнимает страницу основной памяти не у того процесса, в котором собственно возникло страничное нарушение. Эти алгоритмы и называются локальными, потому что они не смотрят в чужую виртуальную память.
- **Глобальный алгоритм** замещения выбирают для замещения физической страницы среди всех страниц основной памяти независимо от того, в виртуальную память какого процесса эта страница в данный момент входит. Глобальные алгоритмы обладают несколькими недостатками:
    - Во-первых, глобальные алгоритмы приводят к тому, что одни процессы становятся чувствительными к поведению других процессов. Например, если в системе работает некоторая смесь процессов, одни из которых использует большой объем основной памяти, которая ему объективно необходима, то все остальные процессы могут замедлиться из-за того, что у них будет меньше памяти.
    - Во-вторых, если некорректно работающий процесс возникает в системе (например, процесс всё время расширяет свой стек, тем самым у него все время возникают некоторые запросы на всё новые и новые страницы), то он может захватывать все больше и больше основной памяти, если система недостаточно внимательно к этому относится. Тем самым он мешает другим процессам жить. Это может быть некорректно работающее приложение, может быть приложение, которое просто написано таким образом. Например, представим, что какая-то программа вычисляет n факториал рекурсивным образом, если не делаются какие-нибудь ухищрения на стадии подготовки такой программы, то она будет неограниченно растить стек (в зависимости от того, какой у неё параметр - от чего считается факториал). Программа может со своей точки зрения работать корректно, но память при этом расходовать совершенно безумно. Поэтому в многопроцессной системе - системе, которая обеспечивает одновременное выполнение нескольких процессов - лучше использовать более сложные, но эффективные локальные алгоритмы. В этом случае требуется, чтобы для каждого процесса сохранялся список

физических страниц основной памяти, которые принадлежат этому процессу. Этот список страниц иногда называют **рабочим набором процесса - working set**. Что такое рабочий процесс, мы более подробно рассмотрим позже. В действительности с этим понятием связано понятие локальности, которое в случае управления виртуальной памятью имеет важнейшую роль, а также алгоритмы подкачки, которые основаны на поддержке рабочих наборов в процессах и использовании принципов локальности.

Уже давно в большинстве компьютеров (начиная с конца 60-х - начала 70-х годов) в аппаратуре компьютеров имеются некоторые простейшие аппаратные средства, которые позволяют собирать статистику обращений к памяти, позволяют операционным системам смотреть не вперед, но хотя бы назад. Для этого с каждым элементом таблицы станиц связываются два специальных флага, которые хранятся в этом элементе:

- **флаг обращения**, который устанавливается автоматически, когда происходит любое обращение к этой странице;
- **флаг изменения**, который устанавливается, если производится запись в эту страницу.

Для того, чтобы собирать статистику, операционная система должна периодически сбрасывать эти флаги. С флагами связана интересная хитрость, которая реально применяется в аппаратуре - дешевая, массовая основная память обычно делается с контролем правильности записи в соответствии с четностью. Это делается следующим образом: когда в память пишется некоторое слово из каких-то регистров процессора, то подсчитывается циклическая сумма по модулю 2 для всех битов этого слова, то есть формируется бит, который называется битом четности. Он дополняет эту циклическую сумму до нечетности: если в результате циклического суммирования получился 0, то этот бит устанавливается в единицу, если получилась 1, то он устанавливается в 0. Когда слово читается из памяти, то бит тоже считывается на регистр. Когда необходимо, то проверяется, какой он (0 или1). Если оказалось, что он неправильно установлен, то возникает контроль памяти. Эти флаги обращения должны устанавливаться каждый раз, когда происходит обращение к странице, поэтому если каждый раз необходимо проверять четность (то есть считать контрольную сумму по модулю 2), то эта операция становится такой же дорогой, как если просто писать в память. По этому поводу делается следующее: в действительности флаг обращений пишется в два разряда - это два нуля или две единицы. Понятно, что если запись происходит сразу в два разряда, то четность не меняется, и можно при такой записи её не контролировать, не считать заново. Такая маленькая хитрость в действительности позволяет сохранять производительность аппаратуры при использовании таблицы страниц, если элемент таблицы страниц на самом деле в это время находится в основной памяти.

Алгоритм FIFO

First In, First Out - это самый простой алгоритм. В этом случае стратегия состоит в том, что замещению подлежит та страница, которая дольше всего находится в основной памяти. Считается, что раз она там долго находится, то, скорее всего, она уже не нужна. Для этого каждой странице присваивается временная метка о том, когда она была приписана к виртуальной памяти некоторого процесса (если это глобальный алгоритм, FIFO может быть и глобальным, и локальным). Реализовать это можно путем создания очереди страниц основной памяти, в конец которой страницы попадают, при попадании в основную память, то есть когда она приписывается к виртуальной памяти некоторого процесса. Из начала берется страница, когда необходимо освободить память. То есть для замещения выбирается страница, которая дольше всего находится в основной памяти. В этом случае предположение, что самая старая станица скорее всего более не нужна - ложное, потому что мы можем представить такой сценарий, когда при многопользовательском режиме использования компьютера в сети работают программисты, они пишут программы, всем им необходим какой-то текстовый редактор, станицы кода этого текстового редактора тем самым могут быть в памяти давно, но их нет необходимости заменять. В этом случае, если заменяется активная страница, которая реально используется - всё продолжает работать корректно, но немедленно происходит страничное нарушение, то есть в действительности это означает, что мы неправильно выбрали страницу.

Аномалия Belady. Интуитивно ясно, что чем больше основная память, то есть чем больше страничных кадров, тем реже вроде бы должны происходить страничные нарушения/page fault. Как ни странно, дело не всегда обстоит именно так. Как установил венгерский математик **Ласло Биледи (Belady)** - некоторые последовательности обращений к страницам в действительности приводят к увеличению числа страничных нарушений, если процессу выделено больше страничных кадров. Это явление называется аномалией Belady или аномалией FIFO.

Рассмотрим, как ведет себя алгоритм, если имеется три страницы основной памяти (табл.1) и четыре страницы виртуальной памяти (табл.2). Для определенности будем считать, что речь идет о локальном варианте алгоритма FIFO.

3 страницы основной памяти (табл.1)

Строка обращения	0	1	2	3	0	1	4	0	1	2	3	4
Самая новая	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Самая старая			0	1	2	3	0	0	0	1	4	4
Page fault	✓	✓	✓	✓	✓	✓	✓			✓	✓	✓

4 страницы основной памяти (табл. 2)

Строка обращения	0	1	2	3	0	1	4	0	1	2	3	4
Самая новая	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Самая старая				0	0	0	1	2	3	4	0	1
Page fault	✔	✔	✔	✔			✔	✔	✔	✔	✔	✔

Наверху таблицы показана строка обращения, то есть в каком порядке процесс обращается к страницам своей виртуальной памяти. В следующих трех строках показано - какая страница является самой новой в основной памяти при работе алгоритма FIFO, внизу - самая старая страница, в промежутке - какая страница в середине, галочками отмечено - когда возникают исключительные ситуации по отсутствию страницы. Мы видим, что при такой строке обращения, если имеется всего 3 страницы основной памяти (табл.1), то соответственно возникает всего 9 прерываний/faults по отсутствию страницы. В табл. 2 показано, как работает алгоритм FIFO при такой же строке обращения, если имеется 4 страницы основной памяти. Оказывается, что в этом случае при работе того же алгоритма возникает 10 прерываний/faults. То есть при некоторых допустимых видах обращений алгоритм работает с меньшим числом прерываний по отсутствию страницы при наличии меньшего объема основной памяти. Аномалию FIFO можно считать курьезом, который иллюстрирует сложность операционной системы, в которой интуитивный подход не всегда является приемлемым. В действительности необходимо смотреть на реальные ситуации и реальные возможные сценарии поведения процессов.

Оптимальный алгоритм Биледи. Одним из последствий обнаружения аномалии Belady является поиск оптимального алгоритма. Можно доказать, что этот алгоритм имеет минимальную частоту страничных нарушений среди всех алгоритмов. Алгоритм работает следующим образом: он замещает ту страницу основной памяти, которая в будущем не будет использоваться в течение наиболее долгого промежутка времени. В этом случае каждая страница помечается числом команд, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка. Поскольку этот алгоритм рассчитан на то, что операционная система знает будущее (знает, как процесс будет вести себя не в настоящее время, не в прошлом, а в будущем), то, очевидно, что реализовать его нельзя. Аналогичные проблемы были связаны с планированием процессов. Так, например, алгоритм SJF/Shortest-Job-First тоже был рассчитан на то, что система знает, каким следующим будет требование процесса на пребывание на процессоре. Этот алгоритм можно реализовывать только в приближенном виде, то есть можно прогнозировать будущее на основе прошлого. Аналогично для алгоритмов замещения страниц: чтобы алгоритм замещения был близок к идеальному - система должна как можно точнее предсказывать будущие обращения процессов к памяти.

В области управления виртуальной памятью алгоритм Биледи тем самым имеет, с одной стороны, теоретическое значение, с другой стороны - его можно использовать как некоторый эталон для оценки качества тех алгоритмов, которые можно реализовать. Интересна следующая особенность: несмотря на то, что как раз для операционных систем практического смысла алгоритм Биледи не имеет, в области программирования его использовать можно. Одна из областей, где он реально используется - это компилятор. В действительности при компиляции машинного кода приходится решать задачу, которая на самом деле достаточно близка к задаче оптимального управления виртуальной памятью - это распределение регистров. Обычно в компьютерах имеется некоторый ограниченный набор регистров, а компилятор должен обеспечить такую политику их распределения, чтобы в каждый момент времени на регистрах находились те данные, которые именно в это время необходимы. То есть нужно, чтобы в тот момент, когда компилятору необходимо разместить какие-то данные на регистре (обычно регистры к этому времени заняты) - требуется выбрать тот регистр, содержимое которого можно записать в основную память, заменить его. Если операционная система будущего поведения процесса не знает, то компиляторы знают - как будет вести себя программа в будущем, потому что они всю программу компилируют. Алгоритмы, которые близки к тому принципу, что необходимо заменять тот регистр, который в будущем дольше всего не будет использоваться (то есть данные, которые на нем находятся, дольше всего не потребуются) - почти всегда может быть использован для компиляторов.

Алгоритм LRU - The Least Recently Used. Это алгоритм, который очень часто используется на практике. Он основан на следующей характеристике: недавнее прошлое является хорошим ориентиром для прогнозирования ближайшего будущего. Основное отличие между алгоритмом FIFO (по поводу которого демонстрируется аномалия Биледи) и оптимальным алгоритмом Биледи состоит в том, что FIFO смотрит назад (то есть предполагает, что та страница, которая дольше всего находится в основной памяти - дольше всего не потребуется в будущем), а другой - вперед. Если использовать прошлое для аппроксимации будущего, то в основе алгоритма LRU лежит то предположение, что стоит замещать страницу, которая не использовалась в течение наиболее долгого времени в прошлом. Не использовалась - значит и не будет использоваться.

Least recently used часто используется в разных областях, не только в виртуальной памяти. Близкую задачу, например, приходится решать в системах управления базами данных, где на программном уровне реализуется кэш дисковых страниц в основной памяти (и тоже приходится решать проблему замещения). LRU принято считать хорошим алгоритмом, основная проблема состоит в том, как его реализовать. Необходимо иметь связанный список всех страниц основной памяти, в начале которого будут страницы, которые используются наиболее часто. Список необходимо будет обновлять при каждом обращении, поэтому станицы в списке приходится искать. Теоретически есть вариант реализации алгоритма LRU со

специальным устройством - это 64-битный регистр, у которого после выполнения каждой команды значение увеличивается на 1. Тогда в таблице страниц можно содержать соответствующее поле, в которое заносится значение регистра при каждом обращении к странице. При возникновении исключительной ситуации по отсутствию замещается та страница, у которой наименьшее значение этого поля.

Можно показать, что как оптимальный алгоритм Биледи, так и алгоритм LRU не страдают от аномалии Беледи. По этому поводу имеется целый класс алгоритмов замещения страниц, которые называются **стековыми (stack) алгоритмами**, которые не проявляют аномалии Белейди. Это именно те алгоритмы, для которых аномалия Биледи не действует. Для них при одной и той же строке обращения множество страниц в памяти для n кадров - всегда является подмножеством страниц для n+1 кадра. Чистая реализация алгоритма LRU невозможна без поддержки специального оборудования. В действительности приходилось реализовывать этот алгоритм без такой поддержки, это была не совсем чистая реализация, но достаточно близкая к ней. Для этого можно поддерживать список страниц основной памяти с привязкой их к страницам виртуальной памяти, в которые они соответственно входят. В глобальном варианте алгоритма LRU, можно просто квантовать время процессора на некие кусочки, и после каждого кванта смотреть на признаки обращения к страницам всех процессов. Если за период этого кванта обращение было, то соответствующие страницы основной памяти в этом списке переставляются в начало списка. Во время обработки все признаки обращения сбрасываются. Тогда очевидно, что страницы, к которым долго не было обращений, оседают в конце списка. Из них как раз можно выбирать кандидатов на замещение. Это неточная оценка, потому что мы смотрим - было ли хотя бы одно обращение в течение этого периода, то есть подтверждено/не подтверждено. Но, по крайней мере, такая реализация проста и работает более-менее стабильно. Рассмотренные варианты LRU в чистом виде в принципе реализуются, но требуется специальная аппаратная поддержка, которой нет в современных процессорах. Хотелось бы иметь такой алгоритм, который, с одной стороны, близок к LRU, но для реализации которого такая сложная и специальная поддержка не требуется. Один из таких возможных алгоритмов - это алгоритм NFU.

Алгоритм NFU - Not Frequently Used. В этом случае для каждой страницы основной памяти заводится программный счетчик, время квантуется, при каждом прерывании по времени (а не после каждой инструкции, не после каждой команды) операционная система просматривает все страницы, находящиеся в памяти (когда все они входят в какую-нибудь виртуальную память). У каждой страницы с установленным флагом обращения значение счетчика увеличивается на единицу, а флаг обращения сбрасывается. Кандидатом на освобождение оказывается страница, у которой наименьшее значение счетчика, то есть это та страница, к которой реже всего обращались. Видно, что в этом алгоритме список поддерживать не надо. Страницу приходится искать среди всех страниц основной памяти. Самым главным недостатоком этого алгоритма является то, что он не забывает прошлое. Например, если к какой-то

странице в течение некоторого времени часто обращались, а потом процессы к ней обращаться перестали, то её счетчик все равно будет содержать большую величину, значит она не будет удаляться из основной памяти. Например, если работает какой-то компилятор с многими проходами, то могут быть страницы, которые активно использовались во время первого прохода компилятора, а потом они использоваться перестали. Тем не менее они будут удерживаться в основной памяти, потому что у них большие значения счетчика (мешая загрузке полезных в дальнейшем страниц). Возможна небольшая модификация алгоритма, которая поддерживает "забывание" прошлого. Для этого достаточно, например, чтобы при каждом прерывании по времени (то есть при каждом срабатывании этого таймера, при квантовании) содержимое каждого счетчика сдвигалось вправо на 1 бит, а потом увеличивается его значение для страниц с установленным флагом обращения. Ещё одним недостатком этого алгоритма является то, что для поиска кандидата на замещение требуется просматривать таблицы страниц.

Другие алгоритмы

Из простых, но эффективных алгоритмов следует отметить а**лгоритм Second-Chance,** который активно использовался на заре оперативной системы Unix. Это простая модификация алгоритма FIFO, которая при выборке кандидата на замещение всего-навсего смотрит - не было ли к ней обращения в последнее время. Если выбирается самая старая страница, а этот признак обращения тем не менее говорит, что к ней обращение было, то страница не замешается, бит сбрасывается, а страница становится в конец очереди. Когда она дойдет до конца, то мы ещё раз посмотрим, а не было ли к ней обращения, и т.д. В некотором смысле это напоминает Least recently used, но гораздо более грубый, конечно. Если оказывается, что процессы ведут себя настолько активно, что ко всем страницам основной памяти за последнее время были обращения, то алгоритм превращается в FIFO.

Алгоритм NRU - Not Recently-Used. В этом случае страница-кандидат на замещение выбирается на основе анализа битов модификации (то есть менялась ли страница, было ли к ней обращение). Основным кандидатом являются те страницы, в которых обращений не было. Если обращение было ко всем страницам, то выбирается страница, которая не менялась, чтобы не надо было её переносить из основной во внешнюю память.

Thrashing, локальность, рабочий набор

Рассмотрим, что такое trashing, как это связано с понятием "локальность", что такое рабочий набор. Встает вопрос: если мы выделили процессу некоторую память, то есть некоторое количество страниц основной памяти - что делать, если ему их не хватает, если он всё время просит каких-то дополнительных страниц? Необходимо ли его приостановить? Насовсем или на время? С освобождением всей занятой им основной памяти? Что необходимо понимать под достаточным количеством этой

памяти? В действительности понятно, что, хотя теоретически можно уменьшить число кадров (то есть число страниц основной памяти) процесса до минимума - всегда существует какое-то число активно используемых процессом страниц, без которых он слишком часто будет генерировать страничные нарушения, слишком часто будут возникать внутренние прерывания.

Эта высокая частота страничных нарушений называется **трешинг - trashing.** Много лет в Советском Союзе пытались придумать какое-то русское название этой ситуации. Когда-то, например, когда ещё не было магнитных дисков, на БЭСМ-6 эту ситуацию называли переворовариваем. Внешне это выглядит следующим образом:

- если работают **глобальные алгоритмы** замещения страниц, то глобальный трешинг возникает, например, при запуске любого процесса, немедленно возникает прерывание по отсутствию страницы, система работает - отбирает какую-то страницу у какого-то другого процесса. Соответственно, этот процесс ожидает, пока будет выполнена подкачка из внешней памяти содержимого страницы, которая должна будет там находиться, чтобы он продолжил работу. Система в это время запускает тот процесс, у которого только что страницу отняли. Оказывается, что это именно та страница, которая процессу сейчас необходима, то есть у него возникает исключительная ситуация/page fault, для него отбирается страница у третьего процесса и т.д. То есть в действительности при попытке запустить на процессоре какой-то процесс - каждый раз возникает страничное нарушение, а система занимается исключительно только обменами с внешней памятью для подкачки или откачки страниц и никак дальше не двигается. Внешне на БЭСМ-6 это выглядело так, как будто система начинает дрожать, на передних панелях БЭСМ-6 было много неонок, когда система находилась в рабочем режиме они все перемигивались, было видно, что система работает. А в описанной ситуации возникала статическая картинка, причем не просто статическая, а нервная: все дрожит и колоссальное количество обменов с магнитными барабанами, которые тогда ещё использовались для целей подкачки и откачки страниц основной памяти. Этот процесс можно хотя бы как-то наблюдать, то есть глобальный трешинг выглядит так, как будто система очень занята, но при этом она ничего не делает, при этом все время обменивается с внешней памятью.
- хуже, когда работает **локальный алгоритм**. Трешинг в локальном режиме: когда запускается процесс, у него соответственно возникает страничное нарушение, система заменяет некоторую его страницу памяти, следовательно, этот процесс не работает - ждет, пока закончится обработка page fault, когда она закончена - процесс запускается и немедленно требует ту страницу, которую у него только что отняли. В случае локального трешинга внешне вся система работает, то есть все процессы, кроме одного - двигаются, а один процесс почему-то всё время занимается обращениями к внешней памяти из области

подкачки. Такую ситуацию внешне распознать трудно. Единственное, на что это похоже - что процесс никак не двигается, то есть у него собственное время (сколько он пребывает на процессоре) никак не возрастает.

В действительности процесс находится в состоянии трешинга, если он занимается откачкой и подкачкой страниц намного больше времени, чем выполнением. Это критическая ситуация, которая возникает вне зависимости от конкретных алгоритмов замещения. Если говорить про глобальные алгоритмы замещения страниц, то чисто практически для борьбы с трешингом на БЭСМ-6 в операционной системе НД-70 использовался следующий подход: там ядро операционной системы могло предположить, что система попала в трешинг (то есть все процессы стоят, все время идут обмены с внешней памятью и т.д.). Оказывалось, что для того, чтобы из этой ситуации выйти (по крайней мере, на достаточно долгое время) достаточно какой-то процесс, обладающий своей виртуальной памятью, насильственно придержать, то есть вообще не выпускать его на процессор. Тогда в действительности рано или поздно у него отбирают память, а ситуация трешинга рассасывается, потому что процесс больше не вызывает page fault, так как его не пускают на процессор. С локальными алгоритмами, конечно, так не получится, там трешинг может возникнуть вне зависимости от конкретных алгоритмов замещения. В этой ситуации важно смотреть на то, какие алгоритмы необходимо выбирать.

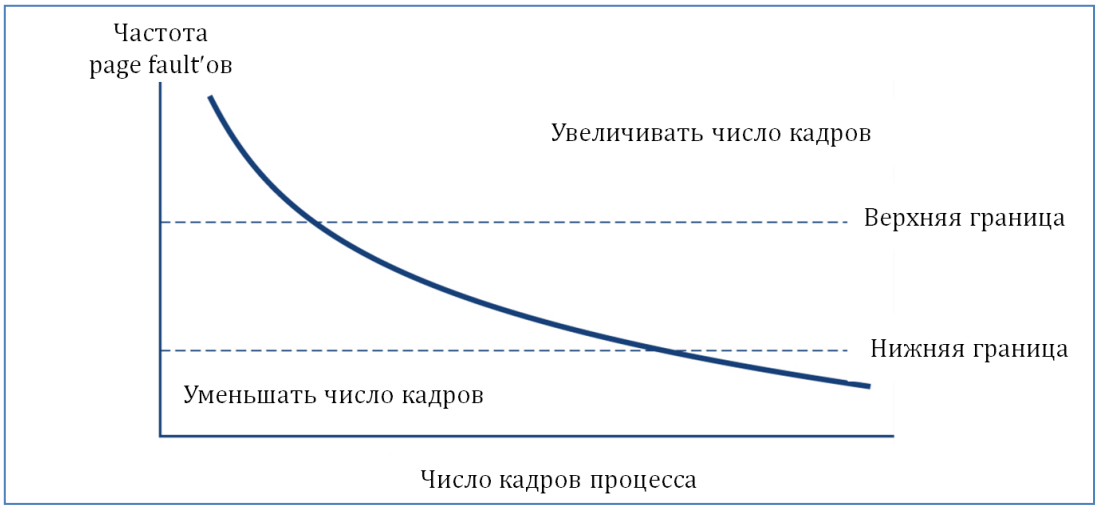

Рис. 10.1. Частота page faults в зависимости от количества кадров, выделенных процессу

Результатом трешинга обычно является снижение производительности всей системы, то есть система не выполняет полезной нагрузки. На графике (рис. 10.1.) изображено, как при глобальном алгоритме замещения процесс, которому не хватает основной памяти, отбирает кадры у других процессов, эти процессы отбирают их у третьих и т.д. В результате все процессы попадают в очередь запросов к устройству внешней памяти, а очередь процессов в состоянии готовности становится пустой.

Операционная система видит, что готовых процессов нет, следовательно, запускает ещё процессы. Тем самым пропускная способность системы падает.

Эффект трешинга, возникающий при использовании глобальных алгоритмов, может быть ограничен за счет использования локальных алгоритмов замещения. Тогда, если один из процессов попадает в состояние трешинга, то это не мешает работать другим процессам. Однако такие процессы проводят много времени в очереди к устройству внешней памяти, на котором ведется подкачка и откачка страниц, затрудняя подкачку страниц остальных процессов. Теоретически гарантирует отсутствие трешинга алгоритм Биледи, но он нереализуем на практике, поэтому его использовать нельзя.

Итак, **трешинг** - это противоестественная высокая частота страничных нарушений. Необходимо каким-то образом эту частоту контролировать. Когда она высока - это означает, что процесс нуждается в дополнительных кадрах основной памяти. Если установить желаемую частоту страничных нарушений, то можно регулировать размер процесса (размер - это сколько ему необходимо основной памяти), добавляя или отнимая у него кадры основной памяти. Иногда может оказаться целесообразным процесс придержать, не давать ему работать, освободив его кадры. Тот пример, который мы обсуждали ранее - это чисто технический прием, который именно это и делает. Тогда освободившиеся кадры выделяются другим процессам, у которых высокая частота страничных нарушений.

Чтобы предотвращать трешинг нужно выделить процессу столько кадров, сколько ему требуется. Но как узнать, сколько кадров ему нужно? По этому поводу существует несколько подходов. Необходимо попытаться выяснить, сколько кадров процесс реально использует. Этот подход определяет **модель локальности** выполнения процесса.

Концепция локальности

Суть модели или концепции локальности состоит в следующем: когда процесс выполняется, он реально двигается от одной локальности к другой. **Локальность** - это набор страниц, которые активно используются вместе. Обычно программа состоит из нескольких различных локальностей, которые могут перекрываться, то есть переходить из одного состояния в другое при сохранении некоторой части тех же самых страниц. Например, если в программе срабатывает вызов процедуры, то определяется новая локальность, которая включает команды этой процедуры, ее локальные переменные (переменные, которые объявлены внутри этой процедуры), и некоторого подмножества множества всех глобальных переменных, которые в этой процедуре используются. После завершения процедуры процесс возвращается в локальность вызова, то есть оставляет локальность этой процедуры. Но он может вернуться к этой локальности снова. То есть на самом деле локальность определяется кодом, составом команд и данными программы. Следует заметить, что модель локальности - это принцип,

который положен в основу работы любого кэша, в том числе аппаратных кэшей процессоров. Если бы доступ к любым типам данных был случайным, кэш был бы бесполезным. Если процессу выделять меньше кадров, чем ему нужно для поддержки его локальности, то он будет находиться в состоянии трешинга.

В действительности, хотя чисто эмпирически установлено, что почти всегда программам, которые пишутся людьми, свойства локальности присущи - бывают важные примеры программ, которые не поддерживают модель локальности. У них совершенно хаотически разбросаны обращения к памяти, мы не можем сказать в любой момент времени, какие реально необходимы страницы и т.д. Понятно, что для таких программ кэш перестает работать, то есть при каждом обращении к основной памяти в кэш будет загружаться некоторый блок основной памяти, который никогда не будет использоваться повторно. Как ни странно, хотя таких примеров мало, выяснилось, что в действительности они очень важные. Современные компьютеры плохо справляются с решением задач, которые не обладают свойством локальности, но которые тем не менее требуется решать, потому что они жизненно важные. По этому поводу 15 лет назад была придумана архитектура, которая как раз была рассчитана на то, что свойство локальности в программах не поддерживается. Это архитектура процессора без кэша с очень большим числом аппаратно-поддерживаемых потоков управления threads. Каждый такой thread обладает своим набором регистров. если при выполнении какого-то thread происходит обращение к основной памяти (то есть там есть желание загрузить какие-то данные на регистр), то на аппаратном уровне этот thread откладывается. Поскольку их очень много (десятки тысяч), то к этому моменту обязательно найдется аппаратно-поддерживаемый thread, все данные которого находятся на регистрах. Он аппаратно запускается на процессоре. Скорость обращения к регистрам такая же, как обращение к кэшу. Понятно, что такой компьютер в действительности будет работать со скоростью кэша, если удастся выполняемую программу распараллелить на очень большое число таких маленьких потоков управления. Мне эта архитектура кажется исключительно интересной. У нас в России по этому поводу велись исследования, даже были некоторые прототипы. Но, к сожалению, эта работа в результате заглохла, хотя в США разработкой таких безкэшевых процессоров, которые основаны на поддержке программ, не обладающих свойством локальности, занимается известная американская компания Cray Research. То есть это программа, которая поддерживается американским правительством.

Модель рабочего множества - Working Set

Если работает чисто страничная виртуальная память, то процессы могут запускаться на процессоре без заранее подготовленного набора необходимых страниц в основной памяти. Тогда самая первая команда (обращающаяся к памяти, которая в этом процессе будет выполняться на процессоре), сгенерирует page fault/страничное нарушение по отсутствию в памяти необходимой страницы с кодами команд. Следующие страничные нарушения будут происходить при первом обращении к

области глобальных переменных, при выделении памяти для стека и т.д. И только после того, как процесс собрал большинство страниц, страничные нарушения становятся сравнительно редкими. Все это может работать в соответствии с выборкой по запросу (требованию), а не при выборке с упреждением. Конечно, можно написать программу, которая постоянно работает с очень разбросанными адресами, но у большинства процессов свойство локальности поддерживается. Грубо говоря, оно состоит в том, что на любой фазе своего выполнения каждый процесс работает с некоторым небольшим числом страниц (небольшим - условно, потому что это зависит, конечно, от специфики процесса, в любом случае - это меньше, чем общее число страниц, которые ему требуются при выполнении).

Питер Деннинг - это один из очень известных людей в области операционных систем, он назвал этот набор страниц **рабочим набором процесса/ process working set**. В некотором смысле тот подход, который предложен Деннингом, является практически реализуемой аппроксимацией оптимального алгоритма Биледи. Принцип локальности ссылок, как сформулировал Питер Деннинг, - это принцип, который не доказывается, но обычно подтверждается на практике. Он состоит в том, что если в период времени (T-t до T) программа обращалась к страницам (P1, P2, ..., Pn), то при достаточно точном выборе t с большой вероятностью эта программа будет обращаться к тем же страницам в период времени (T, T+t). То есть в действительности принцип локальности Деннинга применительно к управлению памятью - это повторение давно известного принципа: если не слишком далеко заглядывать в будущее, то можно его прогнозировать, исходя из не очень далеко отстоящего периода в прошлом. Набор страниц (P1, P2, ..., Pn) и есть рабочий набор программы или, правильнее говорить, соответствующего процесса в момент времени T.

Конечно, поведение программы может со временем меняться, тем самым и рабочий набор процесса может меняться во времени (как по составу страниц, так и по их числу). С точки зрения управления памятью наиболее важным свойством рабочего набора является, естественно, его размер. То есть в действительности для того, чтобы каждый процесс работал нормально - необходимо, чтобы операционная система выделяла им достаточное число кадров, чтобы в них поместился их рабочий набор. Тогда процесс будет вести себя правильно. Если при этом еще остаются свободные кадры основной памяти, то можно инициировать ещё один процесс. С другой стороны, если процессов слишком много или у них слишком большие рабочие наборы, и рабочие наборы всех, одновременно существующих процессов перестают помещаться в основной памяти, то начинается трешинг, и один из процессов надо приостановить, попридержать. Поэтому решение о размещении образов процессов в основной памяти должно, соответственно, базироваться на размере их рабочих наборов.

Для процессов, которые только образуются, решение о размере рабочего набора может быть принято эвристически. Во время работы процесса размер рабочего набора может динамически меняться. Система должна уметь определять: увеличивает ли

процесс свой рабочий набор или меняет его, переходит на новый рабочий набор. Если основной памяти слишком мало, чтобы было можно содержать рабочий набор процесса, то тем самым страничных нарушений много. Тогда процесс всё время пытается набрать свой рабочий набор, он показывает, что ему мало, и начинается много прерываний по отсутствию страницы. В системах с разделением времени, как мы помним, когда работает свопинг, образы процессов часто целиком перемещаются на внешнюю память. Что делать тогда, когда процесс заново возвращается в память? Необходимо ли ждать пока он снова соберет свой рабочий набор? Во многих системах, которые основаны на стратегии рабочих наборов, информация о них хранится, и они загружаются, даже если процесс еще не начался, не стартовал - это стратегия выборки с упреждением.

Размер рабочего набора может быть фиксированным, а может динамически настраиваться. Настраиваемые размеры, конечно, более правильные, потому что у реального рабочего набора (того, который требуется процессу) размер может меняться, как мы это обсуждали ранее. Рассмотрим один из алгоритмов динамической настройки размеров рабочего набора: при создании каждому процессу назначается минимальный рабочий набор, определяющий число страниц процесса, которые гарантированно находятся в физической основной памяти при его выполнении. Если процессу требуется больше физических страниц, чем минимальный размер рабочего набора, и в наличии имеется свободная физическая память, то система будет увеличивать размер рабочего набора, не превышая его некоторого максимального размера, установленного заранее максимального размера допустимого рабочего набора. Если же процессу требуется еще больше страниц, когда уже нет свободных станиц основной памяти, то дополнительные страницы увеличиваются за счет свопинга без увеличения рабочего набора. Поскольку замещаемые страницы рабочего набора в действительности могут ещё некоторое время оставаться в основной памяти, то при необходимости они могут быть возвращены в рабочий набор достаточно быстро, без каких-нибудь дополнительных дисковых операций. Когда свободной основной памяти становится слишком мало, то система стремится увеличить объем свободной памяти, урезая рабочие наборы для тех процессов, у которых размеры рабочих наборов превышают некоторые минимальные размеры.

Когда режим работы устанавливается, то система следит за количеством исключительных ситуаций по отсутствию страницы для каждого процесса. Если (при том, что у процесса рабочий набор находится в основной памяти) у него возникает page fault, и при этом основная память не слишком сильно загружена, то система просто увеличивает размер его рабочего набора. Если же процесс вообще не вызывает никаких прерываний по отсутствию страниц в течение некоторого времени, то система считает, что у него слишком много памяти и урезает его рабочий набор. В этом случае при использовании подобных алгоритмов система пытается обеспечить наилучшую производительность для каждого процесса, не требуя никакой дополнительной настройки системы со стороны пользователя.

Идея **алгоритма Деннинга**, который иногда называют **алгоритмом рабочих наборов**, состоит в том, что операционная система в каждый момент времени должна обеспечивать наличие в основной памяти текущих рабочих наборов всех процессов, которым разрешена конкуренция за доступ к процессору, то есть для всех готовых процессов. При полной реализации алгоритма Деннинга практически гарантируется отсутствие трешинга. Алгоритм реализуем, это доказано. Известный датский специалист в области операционных систем **Брин Хансен** выполнил в Дании такую разработку. Это был прототип аппаратуры, которая в точности поддерживала алгоритм Деннинга со специальной аппаратной поддержкой. Поскольку требуется, чтобы трешинга не было на произвольной аппаратуре - необходимо, чтобы это можно было реализовать на всех процессорах. Поэтому на практике применяются облегченные варианты алгоритмов замещения страниц, подкачки виртуальной памяти, основанные на идее рабочих наборов.

Демоны пейджинга

Рассмотрим, как это делается в среде Unix: алгоритмы, обеспечивающие отсутствие трешинга, реализуются с помощью специальных фоновых процессов, которые в мире Unix принято называть **демонами или сервисами**, которые периодически проверяют состояние памяти. Например, если они обнаруживают, что имеется мало свободных кадров (то есть основная память сильно загружена), то они могут сменить стратегию замещения страниц. Задача демонов управления памятью состоит в том, чтобы поддерживать систему в состоянии наилучшей производительности. Одним из примеров процесса демона является фоновый **процесс-stealer,** который реализует облегченный вариант алгоритма замещения страницы, он основан на идее рабочих наборов, его применяют во многих клонах операционной системы Unix.

Аппаратно-независимая модель памяти процесса

Реализация функций операционной системы, связанных с поддержкой памяти: ведение таблиц страниц, трансляция преобразований адреса, обработка страничных нарушений, управление ассоциативной памятью (кэшем) и т.д. - очень тесно связана со структурами данных, которые обеспечивают удобное представление адресного пространства процесса. Формат этих структур сильно зависит от используемой аппаратуры и особенностей конкретной операционной системы. Обычно виртуальная память процесса разбивается на сегменты пяти различных типов: код программы, данные (сегмент, в котором располагаются глобальные переменные программы, имеющие инициализацию), стек, совместно используемая память, сегменты файлов, отображаемых в виртуальную память. Обычно - это в мире Unix, и, похоже, что это правильно.

- **Сегмент программного кода** содержит только команды. Он не изменяется в ходе выполнения процесса. Обычно страницы этого сегмента имеют атрибут

read-only, поэтому можно использовать один экземпляр программного кода для разных процессов за счет общего использования сегмента программного кода.

- **Сегмент данных** содержит статические переменные программы и **сегмент стека**, который, соответственно, содержит, как правило, автоматические переменные, которые могут динамически менять свой размер и содержимо. Поэтому они должны быть доступны по чтению и по записи. Они являются частными сегментами любого процесса, поскольку каждый процесс ведет себя в этих сегментах по-своему.

- Для того, чтобы было можно использовать память сразу несколькими процессами - можно создавать совместно используемые **разделяемые сегменты** памяти, для которых допускается доступ по чтению и по записи. То есть они могут входить сразу в виртуальную память нескольких процессов. Для всех процессов они доступны через общую таблицу страниц по чтению и по записи.

Структуры данных, используемые для описания сегментной модели

В Unix эта часть страничной организации используется на аппаратно-независимом уровне, для описания страничной организации - аппаратно-независимая. Для описания виртуальной памяти процесса используются структуры-дескрипторы сегментов, которые связаны с аппаратно-зависимой структурой, данные которой используются при отображении виртуальных адресов в физические.

Описатель сегмента содержит индивидуальные характеристики каждого сегмента, в том числе:

- виртуальный адрес начала сегмента, то есть номер его первой страницы в виртуальном пространстве;
- размер сегмента;
- список допустимых операций;
- статус сегмента, состояние сегмента;
- указатель на таблицу станиц сегмента.

Кроме того, имеется несколько описательных структур на уровне страниц. Для управления физической основной памятью поддерживается несколько списков страниц: свободных, модифицированных, не допускающих модификации. Эти списки просматриваются для выбора нужной страницы в зависимости от ситуации. Описатель физической страницы, кроме прочего, содержит копии признаков обращения и модификации страницы, которые вырабатываются аппаратурой.

Поддержка обобщенной модели организации виртуальной памяти и очень тщательное продумывание связей между машинно-независимой и машинно-зависимой

частями подсистемы управления виртуальной памятью позволяет добиться того, что обращения к памяти, которые не требуют вмешательства операционной системы, производятся напрямую с использованием конкретных аппаратных средств. Если страница виртуальной памяти находится в основной памяти, то отображение происходит без вызова каких-то функций ядра операционной системы. Но все наиболее ответственные действия операционной системы, связанные с управлением виртуальной памятью, выполняются в машинно-независимой части с необходимыми взаимодействиями с машинно-зависимой частью.

Здесь необходимо сделать существенное замечание: решение по поводу того, какие страницы входят в рабочий набор, а какие не входят в рабочий набор, принимается за счет просмотра этой таблицы страниц, которая поддерживается на машинно-независимом уровне. Эти признаки туда копируются из машинно-зависимых структур. Мгновенно это не делается - всё равно есть какая-то задержка между тем, как реально выполнилось обращение к странице, и как признак обращения был скопирован в эту структуру. То есть в действительности тем самым эти алгоритмы замещения страниц при использовании такой модели принимают решения на основе недостоверной информации. Она как бы всегда чуточку отстает от реальной жизни. Какие могут быть последствия для правильного выбора страниц для замещений - судить сложно. Очевидно, что это как бы плата за то, что можно обеспечить машинно-независимую часть управления виртуальной памятью. Легко переносить операционную систему с одной аппаратной платформы на другую, но при этом очень спорный вопрос - насколько точно работают алгоритмы.

Загрузка исполняемого файла, то есть выполнение системного вызова **exec** обычно выполняется путем отображения этого файла - **mapping**, его частей (кода, данных) в соответствующие сегменты адресного пространства процесса. После того, как это отображение установлено, процесс начинает генерировать станичные нарушения для сегментов кода, страниц сегмента данных и стека, при этом с диска подкачиваются необходимые данные. Для стека подкачивать нечего, потому что стек по умолчанию пустой. Для кода и данных делается подкачка. Сегмент данных динамически может менять память, менять свой размер из-за того, что там находится куча, в которой происходит распределение памяти для динамических переменных. Для этого используются вызовы **malloc** или **free** (запрос или освобождение памяти), которые изменяют границу выделенного процессу адресного пространства, модифицируя значение переменной **brk** из структуры данных процесса. В результате происходит выделение основной памяти, а соответствующие строки таблиц страниц, элементы таблиц страниц получают какие-то осмысленные значения. За поддержку списков занятых и свободных областей памяти в сегменте данных отвечают системные библиотеки. На практике, та память в куче, которая освобождается - резервируется (то есть она на самом деле не освобождается по-настоящему) для того, чтобы было можно обслуживать будущие запросы процесса на выделение динамических переменных.

Отдельные аспекты функционирования менеджера памяти

Для обеспечения корректной работы менеджера виртуальной памяти, кроме принципиальных вопросов, связанных с поддержкой абстрактной модели виртуальной памяти и её аппаратной поддержкой, также необходимо учитывать множество всяких нюансов и мелких деталей. В качестве примера такого компонента, который более мелкого масштаба, но чрезвычайно важен, рассмотрим более подробно систему фиксации страниц в основной памяти.

В ряде случаев поддержка виртуальной страничной системы приводит к тому, что некоторые страницы должны быть зафиксированы в основной памяти, то есть система управления виртуальной памятью не должна замещать некоторые страницы. Действительно, факт наличия виртуальной памяти не означает, что отсутствуют операции ввода-вывода. То есть всё равно иногда необходимо явно обращаться к устройствам. Процесс может запросить ввод в некую свою виртуальную страницу, в буфер виртуальной страницы и ожидать его завершения. Тогда он, соответственно, будет отложен, заблокирован. На процессоре будет запущен некоторый другой процесс, у которого может возникнуть page fault. Обработка этого страничного нарушения может привести к замещению именно той страницы, куда задал обмен первый процесс. Поскольку есть специальный режим работы устройств при передаче данных напрямую в основную память - **DMA transfer**, то возникает полное безобразие с этой несчастной страницей, которая используется в разных целях.

Одно из решений этой проблемы - всегда использовать для ввода данных специальные буфера в адресном пространстве ядра, а после этого копировать их в адресное пространство пользовательского процесса. Но это делать нежелательно, потому что перепись - это в действительности бич операционных систем. Перепись - это длительная задержка выполнения процесса. Она непродуктивная, поэтому часто используется второе решение - фиксировать страницы в основной памяти, используя специальный бит фиксации. Зафиксированную страницу нельзя замещать, она расфиксируется после завершения операции ввода-вывода. В этом случае проблема будет решена.

Бит фиксации можно использовать и при нормальном замещении страниц. Например, может так получится, что некоторый низкоприоритетный процесс в конце концов получает процессор и подкачивает с диска нужную ему страницу. Если после этого он сразу вытесняется высокоприоритетным процессом, то этот процесс может сразу заменить вновь подкачанную страницу первого процесса (низкоприоритетного), поскольку на нее так и не было ссылок. Это может привести к тому, что этот низкоприоритетный процесс так никогда и не начнет работать, потому что как только он добирается до процессора, выдает прерывание, откладывается, страницу забирают, и так до бесконечности. То есть имеет смысл до первого обращения, до реального обращения пометить такую страницу битом фиксации, чтобы всё-таки когда-нибудь низкоприоритетный процесс начал работать. Использование бита локализации может

быть опасным, если забыть его отключить. Если такая ситуация имеет место, страница становится неиспользуемой. В Solaris разрешается использование данного бита в качестве подсказки, которую можно игнорировать, когда пул свободных кадров становится слишком маленьким.

Стоит поговорить о том, что такое мягкое реальное время. Ещё одной важной областью применения фиксации страниц оперативной памяти являются системы мягкого реального времени. Это система, которая стремится к тому, чтобы как можно быстрее реагировать на внешние события, то есть к тому, чтобы она гарантировано отрабатывала любое событие за какое-то установленное время. Но эти гарантии не должны быть чрезмерно жесткими. Потому это и мягкое реальное время, что они нечто гарантируют, но, если эти гарантии не соблюдены - не производят какие-то катастрофические последствия. Если это системы жесткого реального времени, к которым относится, например, система управления ядерными реакторами, где двигаются стержни в реакторе, то необходимо их вовремя остановить, иначе произойдет катастрофа. Здесь нельзя допускать выход за какие-то временные рамки. Если работает система жесткого реального времени управления стартом ракеты, то она должна отрабатывать строго за указанное время, иначе ракета просто не взлетит. Мягкое реальное время - это, например, система управления баллистикой космического аппарата на орбите. Такая система работает следующим образом: в зоне видимости пролетает космический аппарат, баллистики смотрят, что в действительности ему необходимо подкорректировать орбиту, они быстро считают - насколько времени и как включить двигатели, далее они должны передать на борт так называемые уставки на выполнение коррекции орбиты. Но если соответствующие программные системы не успеют это сделать, пока аппарат находится в зоне видимости, то катастрофы не произойдет. Аппарат гарантированно сделает ещё один виток, будет можно передать уставки в следующий раз, в следующей зоне видимости.

Чем отличаются системы мягкого реального времени от систем жесткого реального времени?

- **Системы жесткого реального времени** - это системы, которые требуют отладки вместе со всеми приложениями, которые будут в этих системах работать. То есть в таких системах трудно отделить операционную систему от приложений - всё работает вместе, а также отлаживается вместе. Как правило, моделируются все возможные ситуации, то есть полное покрытие, чтобы системы гарантировано никогда не вышли за пределы временных границ.

- **Системы мягкого реального времени** делаются следующим образом: если работает некоторая приоритетная схема управления процессором, то высокоприоритетные процессы выше всех остальных процессов. Они относятся как раз к системам мягкого реального времени, если этот процесс получает возможность работать, то есть его активизируют, то он немедленно получает

процессор в свое распоряжение. Память у него операционной системой никогда не отбирается, она всегда зафиксирована.

Еще одним примером использования фиксации страниц является кэш систем управления базами данных. В системах управления базами данных нельзя полагаться на кэширование, которое применяется в операционных системах общего назначения. Им необходимо знать - к каким страницам базы данных можно сейчас обращаться без доступа к внешней памяти. Для этого в СУБД фиксируют большие куски основной памяти, операционная система их никогда не видит. Они принадлежат только СУБД и управляются только системами управления базами данных.

Вообще говоря, виртуальная память противоречит требованиям реального времени, поскольку приводит к непредсказуемым задержкам при подкачке страниц. Системы мягкого реального времени работают практически без использования виртуальной памяти. Если система смешанного назначения, то есть там поддерживается как реальное время, так и разделение времени, то процессы разделения времени работают на более низких приоритетах, а высокоприоритетные процессы работают на фиксированной основной памяти без подкачки

Кроме систем управления страницами виртуальной памяти имеются и другие интересные проблемы, которые возникают в ходе этого процесса. Например, бывает не так легко повторно выполнить команды, которые вызывают страничные нарушения, так как фактически необходимо вернуться в середину команды. Но это больше проблема аппаратуры. Интерес представляют алгоритмы отложенного выделения памяти, такие как **Copy-On-Write (COW)**. Copy-On-Write – это если несколько процессов начинают выполнять один и тот же файл exe, например, какой-нибудь редактор, у которого есть сегмент данных. Как мы рассматривали ранее - это место, в котором находятся статические переменные, данные - это то, чем они инициализируются. Известный факт из мира С: в каком-то очень большом числе случаев (60%) в статические переменные, которые инициализированы, никогда не производится запись. Эти начальные значения используются именно как константы. Поэтому поначалу делается общая таблица приписки во всех процессах ко одному и тому же сегменту с блокировкой записи (только по чтению). Если в каком-то процессе туда происходит запись, то образуется копия соответствующей страницы. Тогда этот сегмент становится уже частным (по крайней мере, частично частым) для этого процесса. Это было когда-то впервые применено в Mach - микроядре операционной системы, разработанном в Университете Карнеги-Меллона. Это оказалось очень полезным, простым и очевидным способом.

Заключение:

- Система управления памятью является совокупностью программно-технических средств, которые обеспечивают производительное функционирование компьютеров.

- Успех реализации той части ядра операционной системы, которая отвечает за управление виртуальной памятью, определяется тем, насколько близка архитектура аппаратуры, на которой работает операционная система, к абстрактной модели виртуальной памяти этой операционной системы.
- В подавляющем большинстве современных компьютеров возможности аппаратуры существенно превышают потребности модели виртуальной памяти операционной системы. Поэтому создание машинно-зависимой части подсистемы управления виртуальной памятью в большинстве случаев не является слишком сложной задачей. Из-за этого перенос операционных систем категории Unix на новые архитектуры производится достаточно быстро и без особых накладных расходов.

Лекция 11. Файлы с точки зрения пользователя

Файловая система и её основные функции

Эта и следующая лекция, последняя лекция в этом курсе, касаются вопросов, связанных с организацией файловых систем. 11-я лекция посвящена обсуждению разных вопросов с позиции пользователей файловых систем, следующая - больше касается разработчиков файловых систем. Сегодня мы поговорим о том, что такое файлы в целом, про организацию файлов и файловых систем, про операции над файлами, про каталоги и логическую структуру файлового архива. Также рассмотрим организацию доступа к архиву файлов, операции над каталогами и вопросы защиты файлов.

История систем управления данными во внешней памяти начинается еще со времени, когда в качестве устройств внешней памяти были только магнитные ленты, но современный облик системы приобрели с появлением магнитных дисков. Это произошло в середине 60-х годов прошлого века. Магнитные диски и собственно идея файлов и файловых систем появилась в компании IBM. Первые файловые системы были реализованы внутри операционной системы семейства OS/360. До появления магнитных дисков и файловых систем в каждом приложении, в каждой прикладной программе приходилось по-своему решать проблемы именования данных и их структуризации во внешней памяти. Это было очень неудобно из разных соображений, в частности потому, что если в действительности использовать одну магнитную ленту для хранения архива одного человека, например, программиста, то на ней приходилось хранить много разных версий текстов программ, откомпилированные программы (несколько версий), тестовые данные для тестирования программы и т.д. Было необходимо запоминать, где находится каждая порция этих данных. Никакой системной поддержки не было. Конечно, это затрудняло поддержку на внешнем носителе нескольких архивов долговременно хранимых данных, потребность в которых существовала всегда со времени появления компьютеров.

Историческим шагом стал переход к использованию централизованных, общего назначения систем управления файлами или, как их обычно называют, файловых систем. Это не очень удачный термин, потому что файловой системой называют систему управления файлами и весь тот архив файлов, который управляется этой файловой системой. Но в мире операционных систем почему-то так принято. Система управления файлами отвечает за распределение внешней памяти, отображение имен файлов в адреса внешней памяти и обеспечение доступа к данным.

Файловая система или **система управления файлами** - это часть операционной системы (обычно ядра), которая отвечает за организацию эффективной работы с данными во внешней памяти, и обеспечение удобного интерфейса при работе с этими данными. Если заставить программиста работать с дисковыми устройствами напрямую, то потребовалось бы (кроме того, как работать с командами процессора и

знать какой-то язык программирования) хорошо знать, как устроен контроллер диска, какие у него есть регистры, как содержимое регистров влияет на работу контроллера. Непосредственное управление дисковыми устройствами - это прерогатива компонента системы ввода-вывода операционной системы, который называется драйвером магнитных дисков. Мы уже обсуждали, что драйверы - это ответственные программы операционных систем, но не сказать, что самые сложные. Возможно, их сложность заключается в том, что их очень много, потому что устройств много, соответственно, драйверов много. Еще одна сложность состоит в том, что по этой причине драйверы обычно программируются не разработчиком операционных систем. А уже потом, когда система работает, появляются новые устройства. То есть обычно драйверы отлажены не так хорошо, как ядро операционной системы в целом. Для того, чтобы писать драйверы, конечно, необходимо хорошо знать контроллеры устройств. Чтобы избавить пользователей компьютеров от сложностей взаимодействия с аппаратурой внешней памяти, была придумана достаточно понятная абстрактная модель файловой системы. Операции записи или чтения файлов проще понять и проще ими пользоваться, чем делать это на уровне низкоуровневых операций, которые позволяют обеспечить непосредственный доступ к устройствам внешней памяти.

Основная идея состоит в следующем: операционная система делит внешнюю память на блоки фиксированного размера (например, 4 Кб - 4 096 байт). Файл, который обычно представляет собой неструктурированную последовательность байт, реально хранится в виде логической последовательности блоков, которые не обязательно смежным образом находится на магнитных дисках. В некоторых операционных системах (например, в MS-DOS) адреса блоков, которые содержат данные файла, образуют связный список и выносятся в отдельную таблицу, которая целиком хранится в основной памяти. В более развитых операционных системах (в частности в операционной системе Unix) адреса блоков данных каждого файла хранятся в отдельном блоке внешней памяти. Реально не в одном блоке, а в нескольких блоках, если файл достаточно большой (который называется индекс или индексный узел - i-node).

Индексный узел файла состоит из списка элементов, каждый из которых содержит номер блока в файле и сведения о том, где он находится во внешней памяти. На уровне интерфейса, доступного конечным пользователям, для считывания очередного байта файла необходимо установить так называемую текущую позицию в файле, которую можно характеризовать смещением от начала или от конца файла. Если известен размер блока (операционная система знает размер блока внешней памяти), то по текущей позиции легко вычислить номер блока, который содержит необходимое начало той последовательности байт, которую нужно прочитать или записать. После этого, имея доступ к индексному узлу файла, можно по логическому номеру блока узнать номер реального блока дисковой памяти и производить с ним требуемый обмен. Базовой операцией, которая выполняется по отношению к файлу при чтении из файла,

является чтение блока с диска и перенос его в буфер, который находится в основной памяти. Об этом мы дополнительно поговорим в следующей лекции.

Файловая система позволяет на основе использования системы каталогов, которые в просторечье принято называть директориями, а последнее время папками, связать уникальное имя файла с блоками внешней памяти, которая содержит данные этого файла. **Каталоги** - это специальным образом организованные файлы, они играют роль индексов, каждый из которых содержит ссылки на свои подкаталоги или на файлы, если это конечный каталог в цепочке каталогов. С этой точки зрения вся файловая система (как архив файлов) представляет собой один большой индексированный файл. Кроме, собственно, файлов и структур данных, которые используются для управления файлами, под термином "файловая система" понимаются ещё и программные средства, которые реализуют разные операции над файлами. В этом смысле правильнее было бы говорить не файловая система, а система управления файлами.

Основные функции файловой системы:

1. **Идентификация файлов** - это связывание имени файла с выделенным ему пространством внешней памяти.
2. **Распределение внешней памяти для каждого файла**. При работе с каждым конкретным файлом пользователю не требуется самому знать, где этот файл находится на внешнем носителе информации. Например, чтобы загрузить некоторый файл в текстовый редактор с жесткого диска, не надо знать, на какой стороне какого конкретного магнитного диска, на каком цилиндре и в каком секторе находится этот документ или его начало, или середина, и т.д.
3. **Обеспечение надежности и отказоустойчивости.** Это очень важный показатель, потому что стоимость долговременно хранимых данных может во много раз превышать стоимость компьютера. Всем нам приходилось слышать стенания людей, у которых поломался диск на компьютере, и они готовы на что угодно, лишь бы только восстановить данные. Речь не идет о том, чтобы вернуть компьютер в работоспособное состояние, главное - это восстановить данные, потому что это гораздо более важно, чем компьютер, так как можно купить новый, в отличие от данных.
4. **Обеспечение защиты от несанкционированного доступа.** В наше время, когда очень много недостаточно порядочных людей занимаются недостаточно порядочными действиями - очень важно, чтобы доступ к данным, которые являются частными и конфиденциальными данными какого-то человека или организации - чтобы к ним нельзя было обратиться людям, которые не имеют к этому никаких прав.

5. **Обеспечение совместного доступа к файлам.** В действительности это важно, потому что файлы содержат данные, которые могут быть интересны не одному человеку, с ними может быть необходимо работать нескольким пользователям и нескольким приложениям. С этой целью должны быть достаточно удобные и мощные устройства синхронизации доступа.

6. **Обеспечение высокой производительности.** Здесь необходимо отметить, что файловые системы относятся к той категории средств управления данными, основное достоинство которых - это как раз простота и высокая производительность.

В действительности есть как минимум две категории средств управления данными: **файловые системы и системы управления базами данных**. Система управления базами данных делает все, что было перечислено выше, но на очень высоком качественном уровне. И защиту, и надежность, и отказоустойчивость, и совместный доступ к базам данных. Файловые системы тоже все это делают, но так, чтобы при этом не вызывались слишком большие накладные расходы. То есть основной смысл файловых систем состоит в том, чтобы не слишком нагружать накладными расходами программу, которой не нужно сверхвысоких качеств со стороны системы управления данными. Поэтому слишком сложные добавки в файловые системы не всегда оправданы, потому что сложность - это удел систем управления базами данных, файловые системы должны быть простыми.

Иногда в просторечье говорят, что файл - это именованный набор связанных данных, которые хранятся во внешней памяти. Для большей части пользователей вычислительных систем файловая система - это наиболее видимая часть операционной системы, с ней больше всего приходится работать. Она предоставляет механизм, который позволяет в оперативном режиме хранить и обеспечивать доступ как к данным, так и к программам для всех пользователей вычислительной системы. Файл с точки зрения пользователя - единица внешней памяти, то есть любые данные, которые хранятся во внешней памяти, должны быть частью какого-нибудь файла.

Важный аспект организации файловой системы состоит в том, что необходимо учитывать стоимость операций взаимодействия с внешней памятью. Поэтому необходимо понимать, о какой внешней памяти мы говорим, рассуждая про файловые системы. Если иметь ввиду традиционную дисковую память, которая устроена в виде устройств памяти на пакетах магнитных дисков с подвижными магнитными головками, то в этом случае, при выполнении операции чтения вся эта операция состоит из нескольких действий:

1. Подвод головки к дорожке, содержащей требуемый блок - это самая длинная часть этой операции, потому что это механическая часть, её нельзя выполнять слишком быстро (по разным физическим соображениям скорость при движении магнитных головок не может быть очень большой).

2. После необходимо ожидание, пока диск прокрутится на угловое расстояние - такое, чтобы нужный блок оказался под считывающей магнитной головкой - это вторая по времени часть операции, потому что магнитные диски нельзя слишком быстро крутить. Опять по физическим соображениям: они должны быть не очень тяжелые, поэтому они делаются из материалов, которые не выдерживают очень большой скорости (это пластик, так или иначе).

3. Само считывание блока, когда включается магнитная головка - это занимает маленькое время, по сравнению со всем остальным.

Из-за операции подвода головки для выполнения произвольной операции над магнитным диском, то есть когда необходимо подводить головки - требуется значительное время (десятки миллисекунд). То есть в действительности обращение к дисковым устройствам производится примерно в 10^5 раз медленнее, чем обращение к основной памяти. Как минимум в 10 000 раз, а то и в 100 000 раз медленнее. Из этого следует следующее: во-первых, если говорить про магнитные диски с подвижными головками, то критерием сложности алгоритмов, которые работают с внешней памятью, является число обращений к магнитным дискам, а во-вторых, если мы переходим от дисков с подвижными головками к твердотельным дискам (Solid-State диск, SSD, которые обычно производятся на основе флэш-памяти), то там нет этой механической части (ни первой, ни второй), поэтому время выполнения чтения гораздо меньше, чем у магнитных дисков. Критерий сложности: может быть уже необязательным количество обращений к внешней памяти, то есть необходимо считать, сколько тратится команд процессора. Поэтому даже файловые системы, которые работают на твердотельных магнитных дисках, в целом должны быть устроены не так, как файловые системы, работающие на дисках с подвижными головками. Мы будем говорить про файловые системы, которые работают с традиционной дисковой памятью.

В этой лекции мы обсудим вопросы структуризации, именования, защиты файлов; операции, которые над файлами можно выполнять; организацию файлового архива, то есть полного дерева каталогов. В следующей лекции мы обсудим проблемы выделения дисковой памяти, обеспечения производительной работы файловой системы и многие другие вопросы, которые касаются, прежде всего разработчиков систем управления файлами.

Общие сведения о файлах

Имена файлов. Файлы - это абстрактные объекты, задача файлов и файловых систем - хранить данные, скрывая от пользователей детали работы с устройствами. Когда процесс создает файл, он присваивает ему имя, когда процесс завершается - файл продолжает существовать и через свое имя может быть доступен другим процессам или пользователям. Поэтому те данные, которые хранятся в файловых системах, называют **persistent data - постоянно хранимые данные**, они не исчезают после завершения процесса, который эти данные сохранил. Правила именования

файлов зависят от операционных систем, во многих операционных системах имена файлов состоят **из двух частей**: имя + расширение, например, progr.c - это файл, содержащий текст программы на языке С, или autoexec.bat - это файл, который содержит команды shell, команды интерпретатора командного языка. Тип расширения файла позволяет операционным системам организовывать работу с файлом разных прикладных программ, в соответствии с заранее оговоренными соглашениями. Обычно операционные системы накладывают ограничения, как на символы, которые можно использовать в именах файлов, так и на длину имени файла. Так, например, в соответствии со стандартом POSIX, в операционных системах семейства Unix для именования файлов можно использовать имена до 255 символов.

Типы файлов. Важный аспект организации файловой системы и операционной системы состоит в том - должна ли операционная система поддерживать и распознавать типы файлов? Если она действительно это делает, то это может помочь её правильному функционированию. Например, не допустить вывода на принтер бинарного файла. Основными типами файлов, которые пользователи видят напрямую в файловых системах, являются:

- **регулярные (обычные) файлы - regular file**. Обычные файлы содержат те данные, которые в них помещают пользователи.
- **каталоги** - справочники, директории, папки и т.д. - это системные файлы, которые поддерживаются операционной системой для обеспечения структуры всего архива файловых систем. В каждом каталоге содержится перечень входящих в него файлов (то есть имен файлов) и устанавливается соответствие между файлами и их характеристиками (атрибутами).

Хотя внутри подсистемы управления файлами каждый обычный файл представляется в виде набора блоков внешней памяти, для пользователей обеспечивается представление файла в виде линейной последовательности байтов. Сразу отметим, что в действительности в операционной системе Multics на уровне пользователя как раз файл был виден как набор блоков внешней памяти. Это представление файла в виде линейной последовательности байтов, потока байт появилось в Unix как некая абстракция файла, которая позволяет использовать эту абстракцию при работе. Такое представление позволяет использовать абстракцию файла при работе с внешними устройствами (а не только с файлами) при организации межпроцессных взаимодействий и т.д. Например, можно рассматривать клавиатуру как текстовый файл, из которого компьютер получает данные в символьном формате. Поэтому иногда (в Unix это делается всегда) к файловым системам приписывают другие объекты операционной системы, выдавая их за файлы. Например, специальные символьные файлы и специальные блочные файлы (это в действительности устройства, которые скрыты за понятием "файл"), именованные каналы и сокеты, имеющие файловый интерфейс и т.д. Про некоторые из этих объектов мы уже говорили в других разделах данного курса.

Обычные или регулярные файлы. Дальше мы будем говорить в основном об обычных файлах. Обычные или регулярные файлы реально представляют собой набор блоков внешней памяти (возможно, пустой набор) на устройстве, на котором поддерживается файловая система. Эти файлы могут содержать как текстовую информацию (в формате ASCII или в каком-то другом формате), так и произвольную двоичную (бинарную) информацию. Текстовые файлы содержат символьные строки, которые можно распечатывать, просматривать на экране монитора или редактировать обычным текстовым редактором.

Нетекстовые или бинарные файлы. Обычно бинарные файлы имеют некоторую внутреннюю структуру, которая известна только тем программам, которым, собственно, она должна быть известна. Например, исполняемый файл, то есть файл, который содержит образ выполняемой программы, в операционной системе Unix состоит из пяти секций:

- заголовок;
- текст - это коды программ;
- данные - это прообраз сегмента данных, когда программа загружается для выполнения;
- справочник перемещения – location directory мы про это говорили, когда рассматривали настройку программ к памяти, то есть это некоторая специальная структура, которая связывается с каждым символическим именем, используемым в программе, некоторый адрес, который можно корректировать, если мы меняем стартовый 0 программы;
- таблицу символов.

Операционная система выполняет файл, только если он имеет нужный формат, то есть она ожидает, что файл exe имеет некоторый формат. Другой пример двоичного бинарного файла - это архивный файл, то есть файл, который упакован программой zip или arc, то есть это сжатый файл. Типизация файлов не слишком строгая в современных файловых системах. Обычно прикладные программы, которые работают с файлами, распознают тип файла по его имени в соответствии с принятыми соглашениями. Например, файлы с расширениями .c, .pas, .txt обычно программы пытаются воспринять - как файлы, у которых кодировка ASCII; файлы с расширениями .exe программы пытаются представлять - как исполняемые файлы, файлы с расширениями .obj или .zip - это файлы двоичные, соответственно, если это .obj, то подразумевается, что, скорее всего, это объектный файл, .zip - это упакованный файл и т.д.

Кроме имени операционные системы связывают с каждым файлом ещё и другую информацию, например, дату последней модификации файла, текущий размер файла и т.д. Эти и другие характеристики файлов называются атрибутами. В разных

операционных системах, в разных файловых системах список атрибутов может быть разным, но обычно он содержит следующие элементы:

- **основную информацию** - имя и тип файла (под типом имеется ввиду - обычный ли это файл, файл-каталог, специальный файл и т.д.);
- **адресную информацию** - устройство, на котором файл хранится, начальный адрес - где его первый блок, размер;
- **информацию об управлении доступом** - кто владелец файла, какие допускаются операции над файлом;
- **информацию об использовании** - когда файл был создан, когда последний раз читался, когда последний раз менялся и т.д.

Список атрибутов обычно хранится в структуре каталогов или в других структурах, которые обеспечивают доступ к данным файла. Об этом мы поговорим на следующей лекции.

Организация файлов и доступ к ним

Программист воспринимает файл в виде набора однородных записей. Запись - это наименьший элемент данных, который может быть обработан прикладной программой как единое целое при обмене с внешним устройством (то есть меньше он быть не может). В большинстве операционных систем, в соответствии со стандартом POSIX, размер записи равен одному байту. Что, конечно, идет от Unix. В то время, как приложения оперируют записями, реальный физический обмен с устройством внешней памяти осуществляется большими единицами, обычно блоками. Поэтому записи объединяются в блоки для вывода и разблокируются для ввода. В следующей лекции мы обсудим, как память распределяется на уровне блоков, внешнем для файловых систем, а сейчас поговорим о нескольких вариантах структуризации файлов.

Последовательный файл - это простейший вариант. В этом случае файл является последовательностью записей. Поскольку записи, как правило, однобайтовые, то файл представляет собой **неструктурированную последовательность байтов.** Обработка таких файлов предполагает последовательное чтение записей от начала файла, причем конкретная запись определяется ее положением в файле. Запись в этот файл может производится только в хвост. Файл может только дополняться или переписываться. Такой способ доступа называется чисто последовательным - это модель магнитной ленты. Если представить магнитную ленту, намотанную на катушку, то совершенно понятно, что с ней по-другому работать нельзя, так как она мотается от начала к концу. Соответственно, можно читать от начала и писать в конец магнитной ленты, потому что - где мы записали, то там она и кончается. Если в качестве носителя файла используется магнитная лента, то так и делается. Текущую позицию считывания

можно вернуть к началу файла. Это называется rewind, опять же от магнитной ленты, что означало - перемотать ленту на начало.

Файл прямого доступа. В реальной практике файлы хранятся не на магнитных лентах, а на магнитных дисках, на устройствах прямого/random доступа, поэтому содержимое всего файла может быть разбросано по разным блокам диска, которые можно считывать и записывать в произвольном порядке. Номер блока, поскольку все блоки одного размера, однозначно определяется позицией внутри файла. Под номером блока имеется ввиду относительный логический номер, который специфицирует данный блок среди всех блоков диска, принадлежащих файлу. В следующей лекции мы поговорим о том, как можно перейти с относительного номера блока к абсолютному его номеру на магнитном диске. Для доступа к середине файла в этом случае просмотр всего файла сначала не требуется. Для того, чтобы выполнить позиционирование, то есть для того, чтобы специфицировать ту точку в файле, с которой надо начинать чтение или куда желательно писать, используются два способа: абсолютное позиционирование, то есть с начала или с конца файла, или с текущей позиции - это относительное позиционирование. Всё это обеспечивает системный вызов seek. Такие файлы, байты которых могут быть считаны в произвольном порядке, называются файлами прямого доступа.

Таким образом, файл, состоящий из однобайтовых записей на устройстве прямого доступа, - это наиболее распространенный способ организации файла. Базовыми операциями для такого рода файлов являются считывание или запись символа в текущую позицию. На самом деле считывание байтов, потому что для файловой системы нет разницы, содержит файл символы или двоичный файл (то есть там байты, которые символы не представляют). В большинстве языков высокого уровня предусмотрены операции посимвольной записи в файл или чтения из него. Как мы помним, это действительно так для языков высокого уровня, но, например, для языков C и C++ таких операций в языке нет. Там есть библиотека, стандартизованная вместе с языком - stdio. Соответствующая функция этой библиотеки тоже обеспечивает чтение и запись в файлы.

Подобную логическую структуру имеют файлы во многих файловых системах, например, в файловых системах операционной системы Unix. Операционная система в этом случае никак не интерпретирует содержимое файла. Такая схема обеспечивает максимальную гибкость и универсальность файловых систем. С помощью базовых системных вызовов или функций библиотеки ввода-вывода пользователи могут как угодно структурировать файлы, то есть накладывать на байтовую структуру любые свои собственные структуры. В частности, одним из способов хранения баз данных во многих СУБД является использование как раз обычных файлов.

Другие формы организации файлов. Известны как другие формы организации файла, так и другие способы доступа к ним, которые использовались в ранних операционных системах. Они используются и сегодня в операционных системах

мэйнфреймов - mainframe, ориентированных на коммерческую обработку данных. Говорить о том, что мэйнфреймы относятся к ранним системам, достаточно рискованно, потому что это одно из самых надежных образований в компьютерном мире. Они, может быть, имеют не повышающийся спрос, но рынок их очень устойчивый. Не просто так компания IBM выросла в текущую, современную компанию на мэйнфреймах, которые и сейчас продолжают оставаться одной из основных частей бизнеса компании IBM. Так что к ним необходимо относиться уважительно.

Первый шаг в структурировании - это хранение файла в виде **последовательности записей фиксированного размера**, каждая из которых имеет некоторую внутреннюю структуру. Структура записи - это то же самое, что структура записи в языке Pascal, то есть это просто набор полей, каждое из которых имеет какой-то тип данных. В этом случае единицей операции чтения является запись, любая операция записи в файл перезаписывает или добавляет запись целиком. Раньше использовались записи по 80 байт (это соответствовало числу позиций в перфокарте, как на устройствах ввода) или по 132 символа - это число символов, которые помещались в одной строке принтера. В одной из старых операционных систем - CP/M файлы были последовательностями 128-символьных байт. С появлением CRT-терминалов эта идея утратила свою популярность.

Более часто стал использоваться другой способ представления файлов - это **файлы как последовательность записей переменной длины**, каждая из которых содержит ключевое поле в соответствующей позиции внутри записи. Они устроены примерено следующим образом: L1 - это значение ключа у записи L1, L2 - значение ключа у записи L2, и т.д. В этом случае базовая операция - считать запись с каким-либо значением ключа. Записи могут располагаться в файле последовательно, например, в порядке возрастания или убывания значений ключевого поля, или в каком-то более сложном порядке.

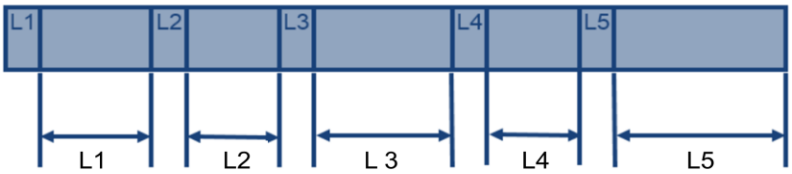

Рис. 11.1. Файл как последовательность записей переменной длины

Метод доступа в последовательном порядке или по значению ключевого поля к записям последовательного файла - называется **индексно-последовательным**. То есть можно просматривать последовательно, а можно через индекс с указанием значения ключевого поля.

В некоторых системах ускорение доступа к файлу обеспечивается за счет построения индекса к файлу. Индекс обычно хранится на том же устройстве, что и сам

файл, и состоит из списка элементов, каждый из которых содержит идентификатор записи, после которого идет указание о том, где эта запись в файле находится. Для поиска записи сначала происходит обращение к индексу, где находится указатель на нужную запись. Такие файлы называются **индексными** или **индексированными**, а метод доступа к ним называется **доступом с использованием индекса**.

Предположим, что имеется некоторый большой и неотсортированный файл, который содержит сведения о студентах. Он состоит из записей с несколькими полями (фамилия, имя, местожительства, возраст), и возникает задача быстрого поиска по какому-то из полей, например, по фамилии студента. Тогда для него можно построить индекс, где есть значение ключа - это имя студента. Соответственно дальше идет ссылка, номер записи, в данном случае для Иванова - это третья запись в этом файле.

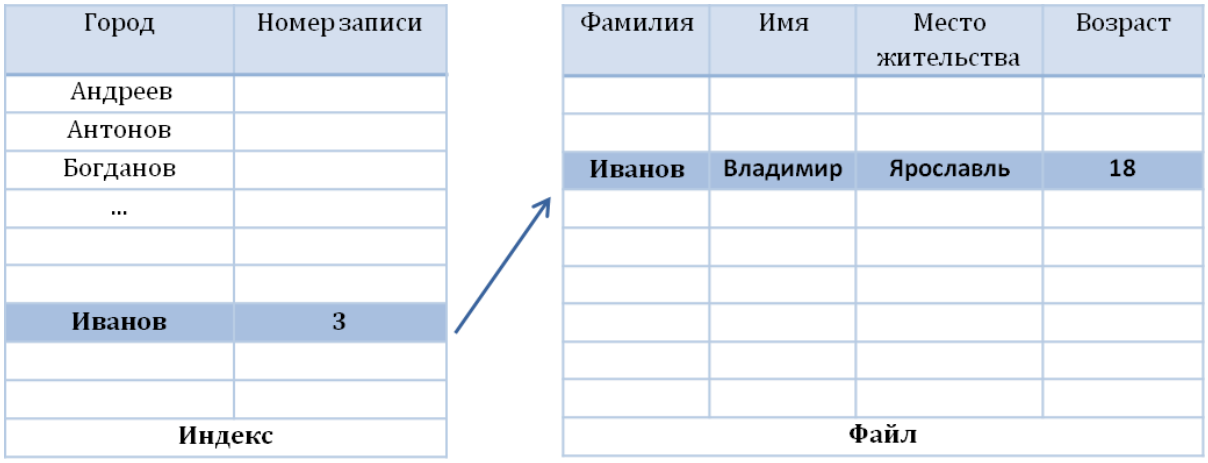

Рис. 11.2. Пример организации индекса для последовательного файла

Почти всегда для повышения эффективности доступа к файловым системам используется **избыточность данных**. Индекс - это, конечно, пример избыточных данных. Другим способом организации индекса является способ выделения дисковой памяти с помощью индексных узлов, которые появились в свое время в Unix. В этом случае операционные системы используют древовидную организацию блоков, при которой блоки, составляющие файл, являются листьями дерева, а каждый внутренний узел содержит указатели на множество блоков этого файла. Для больших файлов индекс может быть слишком большим, в этом случае создается индекс для индексного файла, то есть блоки промежуточного уровня или блоки косвенной адресации.

Операции над файлами

Операционная система должна обеспечить пользователям набор операций для работы с файлами, которые реализуются через системные вызовы. Чаще всего при работе с файлом каждый пользователь выполняет не одну, а несколько операций, потому что,

1. Во-первых, прежде всего нужно найти данные файла и его атрибуты по символьному имени.
2. Во-вторых, нужно считать требуемые атрибуты файла в соответствующую область основной памяти и проанализировать права пользователя на выполнение операции.
3. Наконец, после этого необходимо выполнить саму операцию, после чего освободить занимаемую данными файла область памяти.

Рассмотрим в качестве примера основные файловые операции операционной системы Unix:

1. Первая операция - это **создание пустого файла**, не содержащего данных. Смысл данного вызова состоит в том, чтобы объявить, что файл существует, и присвоить ему некоторый набор атрибутов. При этом выделяется место для файла на диске (может быть выделено, хотя данных ещё нет) и вносится необходимая запись в каталог, чтобы его можно было найти.
2. Парная операция - это **удаление файла** из файловой системы и освобождение занимаемого им дискового пространства.
3. Операция **открытие файла - open fail**. Цель этого системного вызова состоит в том, чтобы дать системе возможность проанализировать атрибуты файла и проверить права доступа к нему, кроме того - считать в основную память список адресов блоков файла для обеспечения быстрого доступа к его данным. Открытие файла сопровождается созданием дескриптора файла или управляющего блока файла. Дескриптор или описатель файла содержит всю информацию о файле, в котором процесс будет работать. Иногда под дескриптором понимается указатель на описатель файлов в таблице открытых файлов, который используется при последующей работе с файлом. Например, если говорить про библиотеку stdio, то при программировании на языке С операция открытия файла возвращает дескриптор fd (в действительности это некоторое целое число), который может быть использован при выполнении операций чтения или записи. Это действительно индекс по таблице открытых файлов данного процесса.
4. Парная операция - это **закрытие файла - close**. Если работа с файлом завершена, то его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл необходимо закрыть, чтобы сказать системе, что больше ничего не требуется, чтобы она могла освободить место во внутренних таблицах файловой системы.
5. **Позиционирование - lseek или seek**. Эта операция обеспечивает возможность указать место внутри файла, откуда затем будет производиться чтение, или куда будет производится запись данных, то есть задать текущую позицию файла.

6. **Чтение данных из файла**. Обычно это можно делать только с текущей позиции. Пользователь или соответствующая программа должны задать объем считываемых данных (то есть число байт) и предоставить для них буфер в виртуальной памяти, куда эта часть байта будет прочитана.

7. **Запись данных в файл с текущей позиции**. Выполнение зависит от того, где находится текущая позиция: если текущая позиция находится в конце файла, то запись происходит в хвост и размер файла увеличивается, в противном случае запись производится на место имеющихся данных, то есть они просто перезаписываются, то есть теряются.

8. Есть и другие операции, например, переименование файла - rename, получение атрибутов файла и т. д.

Существует два способа выполнения последовательности необходимых действий над файлами.

- В первом случае для каждой операции выполняются как универсальные, так и уникальные требуемые конкретные действия - это **схема без состояния - stateless**. Например, последовательность операций может быть такой: open, read1, close, ... open, read2, close, ... open, read3, close.

- Альтернативный способ - все универсальные действия выполнять в начале и в конце последовательности операций, а для каждой промежуточной операции выполнять только уникальные действия. В этом случае последовательность уникальных действий будет выглядеть так: универсальное действие open, универсальные действия read1, ... read2, ... read3, в конце - close.

В большинстве операционных систем можно использовать оба способа, но второй способ более экономичный и быстрый, потому что накладные расходы, которые требуются на универсальные действия open и close, выполняются один раз на много операций. С другой стороны, первый способ более устойчив к сбоям, потому что результаты каждой операции независимы от результатов предыдущей операции. Поэтому, например, он применялся в одной из первых распределенных файловых системах Sun NFS.

Логическая структура файлового архива

Число файлов на вычислительной системе может быть большим. Можно сказать, что сейчас на небольшом ноутбуке хранится несколько десятков тысяч файлов. Они занимают сотни гигабайт дискового пространства. Для того, чтобы с этими данными можно было эффективно работать, чтобы ими можно было эффективно управлять - требуется наличие в них четкой логической структуры. Прежде всего это касается, конечно, именования, потому что стоит задуматься, как будет трудно найти то, что необходимо, если будет линейный список имен, например, из 40 000 имен. То

есть будет необходимо очень сильно думать, как генерировать каждое имя, чтобы потом было можно каким-то образом его найти. Поэтому во всех современных файловых системах поддерживаются многоуровневые именования файлов за счет наличия во внешней памяти дополнительных файлов со специальной структурой - каталогов, которые с подачи Microsoft стали называть директориями. Сейчас их всё больше называют папками.

Каждый каталог содержит список имен каталогов и/или файлов, которые содержатся в данном каталоге. Все каталоги имеют один и тот же внутренний формат - это фактически таблица, где каждому файлу соответствует одна запись в этой таблице или файле-каталоге.

Имя файла (каталога)	Тип файла (обычный или каталог)	
Anti	K	атрибуты
Games	K	атрибуты
Autoexec.bat	O	атрибуты
mouse.com	O	атрибуты

Выше показан пример файла-каталога, в нем содержатся всего четыре объекта, из них два каталога, два просто фала: один - Autoexec.bat, то есть некоторый скрипт, другой - mouse.com - это системный файл.

Число каталогов зависит от того, какая это конкретно файловая система. В ранних операционных системах имелся только один корневой каталог, потом появились каталоги для пользователей, по одному каталогу для каждого пользователя. В современных операционных системах (в действительности такие структуры появились раньше, чем система Unix) уже почти 50 лет можно строить структуры дерева каталогов произвольного размера, произвольной вложенности. Файлы и каталоги образуют на дисках некую иерархическую древовидную структуру.

На рисунке 11.3. показано, что у этой структуры обязательно есть корень, могут быть все вершины нетерминальные, то есть промежуточные вершины - это обязательно каталоги. Листовые вершины могут быть как каталогами, так и файлами. Обычно деревья можно изображать по-разному, что касается формального представления деревьев, то более распространенной является структура перевернутого дерева (когда корень наверху). Поэтому верхнюю вершину дерева называют корнем, у которого, соответственно, нет предка, в него не входит ни одна дуга. Если элемент дерева не может иметь потомков, он называется терминальной вершиной или листом, это может быть каталог или файл. Нелистовые вершины, а каталоги - обязательно, содержат списки имен листовых и нелистовых вершин, то есть каталогов и файлов. Путь от корня к файлу в такой древовидной структуре однозначно определяет файл.

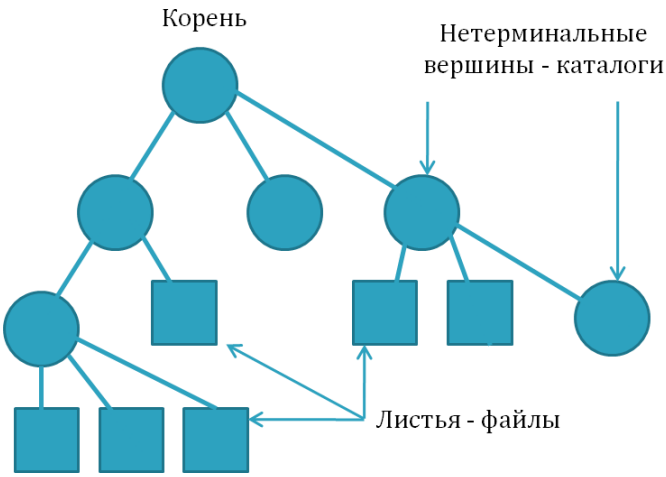

Рис. 11.3.. Древовидная структура файловой системы

Такие древовидные структуры являются графами без циклов. Можно считать, что ребра графа направлены вниз, а корень - вершина, не имеющая входящих ребер. Как мы увидим в следующей лекции, специальная операция "связывание файлов с каталогами", которая используется, в частности в операционной системе Unix, может приводить к образованию циклов в графе. Что, вообще-то, нехорошо. Внутри одного каталога имена листовых файлов уникальны. Имена файлов, которые находятся в разных каталогах, могут совпадать. Для однозначной идентификации файла по его имени (что позволяет избежать коллизии имен) файлы именуются так называемыми **полными именами или путевыми выражениями - pathname**, которые состоят из списка имен вложенных каталогов, по которым можно найти путь от корня к файлу, кроме того, имя файла в каталоге, непосредственно содержащем данный файл. Полное имя включает цепочку имен - путь к файлу. Например, /usr/games/doom: где "/"- это имя корневого каталога; usr - это имя каталога, которое должно содержаться в корневом каталоге; в нем должно содержаться имя – games; в котором должно содержаться имя - doom, которое уже является именем листового объекта, то есть в данном случае - это исполняемый файл игры doom.

Такие путевые выражения задают уникальные имена. Компоненты пути обычно разделяются в Unix прямым слэшем - "/", в Microsoft обратными слэшем, в Multics для этого использовался значок больше - ">". Таким образом использование древовидных каталогов минимизирует сложность назначения уникальных имен. Все делается гораздо проще, потому что за счет вложенных каталогов можно устраивать иерархии именования по разным принципам: можно следовать структуре организации, можно следовать структуре проектов, можно следовать каким-то интересам конкретного пользователя. Но главное, что это сильно облегчает и наименование, и поиск по именам.

Полные имена могут быть длинными, кроме того, не всегда удобно указывать полное имя, если мы хотим именовать файл не в расчете на путь от корня дерева. Поэтому есть другой способ задания имени - это **относительный путь к файлу**. При формировании этого имени используется понятие "рабочего каталога" или "текущего каталога", который обычно входит в состав атрибутов процесса, который работает с данным файлом. Тогда на файлы в таком каталоге можно ссылаться только по имени, а поиск файла будет осуществляться в рабочем или текущем каталоге. Это удобнее, потому что можно использовать короткие имена, которые трансформируются в разные длинные имена в зависимости от того, какой каталог является рабочим. Но по сути для файловой системы - это то же самое, потому что в действительности в этом атрибуте (который задает рабочий или текущий каталог) запоминается именно путь от корня к этому текущему каталогу. Когда процесс задает относительное имя файла, то эти две цепочки: цепочка от корня до рабочего каталога и цепочка от текущего каталога к файлу - конкатенируются, образуя некоторый полный путь. Этот путь используется для того, чтобы искать файл от корня.

Для получения доступа к файлу и локализации его блоков система должна выполнить навигацию по каталогам. Рассмотрим для примера путь /usr/linux/progr.c: сначала в фиксированном месте на диске находится корневая директория, затем находится компонент пути usr; исследуя этот файл, система понимает, что данный файл является каталогом, рассматривает блоки его данных как список файлов и ищет в нем следующий компонент linux. Из строки для linux находится файл, соответствующий компоненту usr/linux/. Затем находится компонент progr.c, который открывается, заносится в таблицу открытых файлов и сохраняется в ней до закрытия файла. В действительности могут быть специальные случаи, если мы искусственным образом меняем структуру дерева за счет использования связывания воедино (монтирование файлов) нескольких отдельных, построенных файловых систем. Этот случай рассмотрим в следующей лекции, которая будет посвящена более тонким и системным аспектам управления файлами.

Многие прикладные программы рассчитаны на то, чтобы работать с файлами текущего каталога или его подкаталогов и не указывать явным образом имя текущего каталога. Это дает пользователям возможность произвольным образом именовать каталоги, которые содержат разные программные пакеты. Для того, чтобы было можно такую возможность реализовать, в Unix в каждом каталоге имеются два специальных элемента с именами: "." - который обозначает сам каталог и ".." - для обозначения его родительского каталога. То есть для того, чтобы явно обозначить, что мы хотим именовать файлы от текущего каталога (того, в котором мы сейчас работаем), можно использовать вместо слэша в качестве первого имени "." (точку). Тогда от неё будут искаться все остальные файлы, как заданные коротким именем.

Организация доступа к архиву файлов

Задание пути к файлу в разных файловых системах отличается тем - с чего начинается эта цепочка имен. В 70-х годах уже было принято разбивать физические дисковые устройства на **логические диски** (это низкоуровневая операция), которые иногда называются разделами - partitions (по-русски партиция). Логический диск - это часть физического диска, которую операционная система позволяет использовать с тем же интерфейсом, как целый и отдельный физический диск. Иногда наоборот - в некоторых операционных системах (как, например, это было возможно в Windows NT) можно объединять несколько физических дисковых устройств в один логический диск. В дальнейшем изложении будем считать, что каждый раздел - это отдельный виртуальный диск. Каждый раз, когда мы будем говорить про диск - это может быть как реальный диск, так и логический диск.

Каждый диск может содержать (если он специальным образом размечен) иерархическую древовидную структуру, состоящую из набора файлов и/или каталогов. То есть файлов, которые содержат перечень других файлов, входящих в состав каталога, необходимых для хранения информации о файлах системы. В некоторых системах управления файлами требуется, чтобы каждый архив файлов целиком располагался на одном диске, то есть разделе диска. В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск. Такой способ именования и организации файловых систем используется, например, в операционных системах компании Digital Equipment (DEC), которая теперь принадлежит Hewlett-Packard, и Microsoft. То есть - сколько есть отдельных дисков, столько может быть отдельных архивов файловых систем.

В других системах (это идет, прежде всего от операционной системы Multics) вся совокупность файлов и каталогов представляет собой единое дерево. То есть сама система в этом случае, выполняя поиск файлов по имени, начиная от корня, требовала в Multics установку необходимых дисков. Такой способ организации в действительности имеет своё преимущество, потому что в этом случае система знает всё про файловые системы. Она может следить за тем, чтобы нужные дисковые устройства (если они съемные) находились на устройствах, чтобы они были доступны в режиме онлайн. Она может отслеживать потребности в резервном копировании и производить его, может поддерживать иерархию памяти, например, перемещать файлы, к которым редко происходят обращения или они вообще престали происходить. Перемещать их, например, на магнитные ленты или на какие-то устройства с большим временем доступа. Есть и свой минус, потому что такая структура кажется пользователю такой, как если бы она была привязана к одному устройству. А на самом деле, если хочется переписать из одной системы в другую какой-то кусок файловой системы (поддерево файловой системы), то получается серьезная "кухня", потому что файлы реально разбросаны по разным дискам, а может быть даже и лентам. В действительности это иногда хорошо, а иногда бывает крайне неудобно.

Есть компромисс, который был реализован в свое время в операционной системе Unix. В этом случае на каждом отдельном диске или разделе может присутствовать отдельная файловая система. То есть на каждом разделе находится свой архив файлов, у каждого из них есть свой корень. После запуска системы можно объединить корневую файловую систему, то есть файловую систему, которая заранее при генерации системы объявлена резидентной, то есть она обязательно должна быть доступна. Система сама знает то дисковое устройство, где находится корневая файловая система, где находится её корневой каталог. Можно привязать к ней ряд изолированных файловых систем с помощью операции mount, логически образовав одну общую файловую систему. Технически это делается с помощью создания в корневой файловой системе специальных пустых каталогов (об этом подробнее в следующей лекции). Системный вызов mount в операционной системе Unix позволяет связать один из этих пустых каталогов с корневым каталогом указанного архива файлов. После выполнения этого системного вызова именование файлов производится точно таким же образом, как если бы вся файловая система с самого начала была централизованной.

Задачей операционной системы является поддержка прохода через точку монтирования при получении доступа к файлу по цепочке имен. То есть в действительности, если говорить грубо, то система реально понимает, что когда мы доходим до этой точки (бывшего пустого каталога корневой файловой системы), то в этот момент необходимо сменить драйвер. То есть мы должны перейти на другой дисковый драйвер и дальше продолжить поиск от корня файловой системы, которая находится на этом диске. Это чистая техника, а если учесть, что обычно монтирование файловой системы производится при запуске системы, то есть когда выполняется командный стартовый файл операционной системы Unix, то её пользователи обычно даже и не задумываются о том, откуда берется общая файловая система, для них она и так общая.

Операции над каталогами

Как и для файлов - система должна обеспечить пользователей набором операций над каталогами, которые реализуются через системные вызовы. Несмотря на то, что каталоги - это файлы, логика работы с ними отличается от логики работы с обычными файлами и определяется природой этих объектов, которые предназначены для поддержки структуры файлового архива. То есть можно сказать, что обычные файлы/ regular files - это файлы, про которые система знает только то, что она предоставляет их пользователю как поток байт. Система совершенно не понимает, как эти файлы структурируют пользовательские приложения. Файловые системы операционной системы совершенно ничего не знают про объектные файлы, про то, как устроены текстовые файлы, для неё совершенно нет разницы между файлами, которые содержат символы, и файлами, которые бинарные. Всё это уходит в приложения. А про каталоги наоборот - файловая система знает, как они устроены. Она знает их структуру, она сама

использует эти файлы. А пользовательские программы и сами пользователи структуру каталогов точно знать не должны, потому что это не их дело (как устроены каталоги). Работать с каталогами как с обычными файлами обычному пользователю запрещается, чтобы не было каких-то специальных побуждений изучать структуру каталогов, которая может меняться в следующей версии операционной системы.

Операции над каталогами являются прерогативой операционной системы, например, пользователь не имеет права выполнить запись в каталог, начиная с текущей позиции (для неё каталог вообще не представляется в виде потока байт). Совокупность системных вызовов для управления каталогами зависит от особенностей конкретных операционных систем.

Для примера рассмотрим некоторые системные вызовы, которые поддерживаются в операционной системе Unix для работы с каталогами:

1. **Создание каталога/create directory.** Отсюда директории и происходят, но directory никогда по-русски директорией не называлась, это каталог. Вновь созданный пустой каталог по умолчанию содержит два имени: "." (точка) и ".." (две точки), это ссылка на самого себя и ссылка на предка. Тем не менее каталог считается пустым.
2. **Удаление каталога.** Удалить каталог можно только в том случае, если оттуда убраны все имена, кроме "." (точка) и ".." (две точки).
3. **Открытие каталога** - для того, чтобы потом его читать. Для того, чтобы вывести список всех имен файлов, которые входят в данный каталог, пользовательский процесс должен открыть каталог и считать имена всех файлов, которые он включает.
4. **Закрытие каталога** после чтения - для того, чтобы освободить место во внутренних системных таблицах.
5. **Поиск** - это операция, которая выполняется файловой системой, то есть отдельной такой операции среди системных вызовов нет. Но практически эта операция возвращает содержимое текущей записи в открытом каталоге. Для этих целей можно использовать системный вызов read, но тогда требуется знание внутренней структуры каталога, и это не разрешается для обычного пользователя.
6. **Получение списка файлов в каталоге** - это распечатка каталога.
7. **Переименование** - это изменение имени каталога. Имена каталогов можно менять, как и имена файлов.
8. **Создание файла**. При создании нового файла выполняется неявная операция добавления необходимого элемента в соответствующий каталог.

9. **Удаление файла** - это удаление из каталога необходимого элемента. Если на тот файл, который удаляется, имеется только одна ссылка, то есть он присутствует только в одном каталоге, то он вообще удаляется из файловой системы. Иначе система удаляет только ссылку по тому пути, который указан для удаления файла. Это опять связано с формированием нескольких путей для одного и того же файла, мы рассмотрим подробнее, как это делается в следующей лекции. Создание и удаление файлов, конечно, предполагает также выполнение соответствующих файловых операций.

10. Имеется еще ряд других системных вызовов, которые связанны, например, с защитой информации. Про безопасность отдельного разговора не будет, а именно про принципы авторизации доступа к файлу мы поговорим попозже.

Защита файлов

Информация в компьютерных вычислительных системах должна быть защищена как от физического разрушения - это проблема надежности (reliability), так и от несанкционированного доступа - это именно protection, а не security. То есть сейчас мы не говорим про обеспечение безопасности, мы говорим про защиту файлов.

Контроль доступа к файлам. Если в операционной в системе поддерживается многопользовательский режим, то требуется обеспечение контролируемого доступа к файлам. В этом случае выполнение любой операции над любым файлом должно быть разрешено только тогда, когда у соответствующего пользователя или у процесса, который выполняется от имени данного пользователя, имеются соответствующие привилегии. Обычно контролируются следующие операции: чтение, запись и выполнение. Могут контролироваться и другие операции: копирование файлов или их переименование. Но как правило, для этого достаточно контролировать чтение, запись и выполнение. Например, операцию "копирование" можно представить, как операцию чтения и последующую операцию записи.

Списки прав доступа. Наиболее общим подходом к обеспечению защиты файлов от несанкционированного использования является доступ на основе идентификатора пользователя, то есть связывания с каждым файлом или каталогом **списка прав доступа - access control list**, в котором перечисляются все идентификаторы пользователей, которым разрешен какой-нибудь способ доступа к этому файлу. Для каждого пользователя перечисляются разрешенные для них типы доступа к файлу. Любой запрос на выполнение операции сверяется с таким списком. Этот способ контроля доступа называется мандаторным или мандатным, то есть в действительности у каждого пользователя по отношению к каждому файлу или вообще к любому объекту операционной системы имеется мандат (или не имеется). Если он имеется, то в нем указано, что пользователь может делать с этим файлом или любым другим объектом. Основная проблема реализации такого мандатного способа защиты - это изменчивость и большая потенциальная длина списка. Если мы в действительности

хотим разрешить всем пользователям читать некоторый файл, то необходимо их всех внести в этот список. Значит, нам необходимо потенциально для каждого пользователя, читающего этот файл, прочитывать вхолостую половину списка, пока мы не найдем его собственный идентификатор.

Два основных нежелательных следствия такого подхода: во-первых, построение такого списка может оказаться сложной задачей, особенно если пользователи системы заранее не известны, если они динамически могут появляться и исчезать. Кроме того, из-за этого запись в каталоге, в котором должны храниться списки контроля доступа, получает переменный размер, то есть в действительности необходимо включать в список потенциальных пользователей.

Из-за этих проблем в операционной системе в обычном режиме (когда не требуется слишком большой уровень защиты) используется другой подход, который называется дискреционным. В этом подходе все пользователи для каждого объекта делятся на три группы: владелец (Owner), группа (Group) - это все пользователи, которые совместно используют этот файл и нуждаются в некотором общем способе доступа к нему, остальные (Public). В этом случае можно список прав доступа сделать очень сжатым, то есть необходимо хранить три поля (по одному для каждой группы) для каждой контролируемой операции. В итоге в операционной системе Unix операции чтения, записи и выполнения контролируются с помощью 9 бит (rwxrwxrwx).

Заключение:

- Файловая система представляет собой набор файлов, каталогов и операций над ними.
- Имена, структуры файлов, способы доступа к ним и атрибуты файлов - это важные аспекты организации файловых систем.
- Обычно файлы представляют собой неструктурированные последовательности байт, потоки байт.
- Основной задачей файловых систем является связывание символьного имени файла с данными, которые хранятся во внешней памяти.

Лекция 12. Реализация файловой системы

Общая структура файловой системы

На этой, последней лекции курса про операционные системы мы рассмотрим вопросы реализации файловых систем. Мы будем говорить про общую структуру файловой системы, управление внешней памятью, про реализацию каталогов, операцию связывания или монтирования файловых систем, про связывание файлов и возможность использования разных цепочек имен по отношению к одному и тому же реальному файлу. Рассмотрим вопросы кооперации процессов при работе с файлами и синхронизации доступа, надежность файловой системы, производительность файловой системы, некоторые проблемы реализации операций над файлами и архитектуру файловых систем, которые распространены в последние 10-20 лет.

Реализация файловой системы связана с такими вопросами, как поддержка понятия логического блока диска, связывание имени файлов и блоков, в которых находятся данные этого файла, совместное использование файлов и управление дисковым пространством. Обычно, для того чтобы было можно хранить файлы и организовывать архивы файлов на дисках, сначала необходимо выполнить форматирование дисков. Это программное форматирование - фактически некоторая запись первоначальных данных, которые описывают состояние файловой системы в целом. После этого, уже на пользовательском уровне создается нужная структура каталогов (то есть образуется некоторое дерево), которые фактически являются списками вложенных каталогов и собственно файлов. Наконец, когда всё это подготовлено, дисковое пространство заполняется файлами, которые приписываются, соответственно, одному или другому каталогу.

Операционная система должна обеспечить пользователей или программистов приложений некоторым набором системных вызовов, которые обеспечивают все необходимые для них услуги. Кроме того, файловые службы могут и должны решать проблемы проверки и поддержки целостности файлов, файловых систем, проблемы повышения производительности (потому что это очень важно для файлов, они используются в критически важных приложениях) и ряд других.

Структура хранения данных на дисках, вся программная организация может быть устроена следующим образом (как показано на рис. 12.1.): после обращения к ядру операционной системы через системный вызов процессы пользователей начинают работать в логической подсистеме управления, которая занимается поддержкой иерархической древовидной структуры каталогов и файлов, выполняет системные вызовы, которые работают с путевыми именами (patk name), обеспечивает защиту файлов и, соответственно, авторизацию доступа. Эта подсистема опирается на базисную подсистему управления файлами, которая работает уже с блочными файлами, она отвечает за выделение блоков диска и образование соответствующих структур данных. Система управляет распределением памяти, то есть учитывает свободные и

занятые блоки на дисках, поддерживает коды системных вызовов, которые работают с дескрипторами файлов, поддерживает таблицы открытых файлов, поддерживает выполнение операций системных вызовов монтирования файловых систем, выполняет синхронизацию при параллельном доступе к файлам с разных дисков и т.д.

Рис. 12.1. Блок-схема файловой системы

Базисная подсистема управления файлами работает с логическими дисками (то есть разделами физических дисков) через подсистему, которая обеспечивает кэширование блоков диска в основной памяти ядра операционной системы. Логические диски - это абстракции, которые обеспечиваются программным путем. Операционная система отображается уже на физические диски, с которыми работают драйверы устройств, драйверы дисков, которые, соответственно, занимаются и обработчиками прерываний. В конце концов, естественно, оборудование, то есть драйверы на самом деле работают с контроллерами дисков на уровне уже физических блоковых дисков в терминах секторов, цилиндров и дорожек магнитных дисков.

Оборудование

В этом курсе мы ограничиваемся рассмотрением только традиционных устройств внешней памяти - магнитных дисков с подвижными головками. Видимо, до сих пор - это основные устройства внешней памяти, несмотря на то, что их сильно теснят твердотельные магнитные диски SSD. Магнитные диски с подвижными головками - это пакеты магнитных пластин (поверхностей), между которыми на одном рычаге механически передвигается пакет магнитных головок. Шаг движения пакета головок является дискретным, поэтому каждому возможному положению пакета головок логически соответствует цилиндр магнитного диска (то есть такой цилиндр высекается). Цилиндры, в свою очередь, делятся на дорожки, то есть каждая дорожка соответствует поверхностям какой-то магнитной пластины. Каждая дорожка размечается на одно и то же количество блоков (которые здесь мы называем секторами) таким образом, чтобы в каждый блок можно было записать по максимуму одно и то же число байтов информации.

На уровне аппаратуры для обмена с магнитным диском нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от начала этого блока. Таким образом, диски разбиваются на блоки фиксированного размера, и можно получить непосредственный доступ к любому блоку, то есть организовать прямой доступ к файлам.

Система ввода-вывода

Система ввода-вывода - это система нижнего уровня ядра операционной системы, которая прикрывает от других уровней особенности аппаратных средств (в данном случае внешней памяти). Подсистема предоставляет в распоряжение файловой системы используемое дисковое пространство в виде **непрерывной последовательности блоков фиксированного размера**.

- Система ввода-вывода имеет дело с **физическими блоками** диска, которые характеризуются адресом, например, это второе устройство (диск 2), 75-ый цилиндр, 11-ый сектор на этом цилиндре.
- Файловая система имеет дело с **логическими блоками**, каждый из которых имеет номер от 0 или 1 до некоторого N. Размер логических блоков файла совпадает или кратен размеру физического блока диска и, как правило, он равен размеру страницы виртуальной памяти, которая поддерживается операционной системой совместно с аппаратурой соответствующего компьютера.

Как показано на схеме, в структуре системы управления файлами можно выделить базисную подсистему, которая отвечает за распределение дисковой памяти для конкретных файлов, которая работает именно на уровне логических блоков файлов. А также логическую подсистему, находящуюся на более высоком уровне, которая

организует и использует структуру дерева каталогов для предоставления базисной подсистеме необходимой информации, исходя из символического имени файла. Эта логическая подсистема также отвечает за авторизацию доступа к файлам.

Интерфейс

Стандартный запрос на открытие - open или создание - create файла поступает от приложения через системный вызов к логической подсистеме файловой системы. Логическая подсистема использует структуру каталогов (то есть дерево), проверяет права доступа и обращается к базовой подсистеме для получения доступа к блокам файла. После этого файл считается открытым, то есть его описатель содержится в таблице открытых файлов, и приложение в ответ на обращение open или create получает в качестве ответного параметра дескриптор этого файла, который фактически является ссылкой на описатель файла в таблице открытых файлов и используется в запросах приложения: чтение, запись, позиционирование, закрытие файла и т.д.

Запись в таблице открытых файлов через подсистему распределения внешней памяти указывает на блоки данного файла. Если к моменту выполнения системного вызова open файл уже открыт другим процессом (то есть уже используется, то есть тем самым он содержится в таблице открытых файлов), то после проверки прав доступа к файлу может быть организован совместный доступ (если это допускается для данного файла). В этом случае новому процессу также возвращается дескриптор, то есть это ссылка на уже имеющуюся информацию о данном файле в таблице открытых файлов. Рассмотрим работу других важных системных вызовов.

Управление внешней памятью

Прежде чем описывать структуру данных файловой системы на диске - необходимо обсудить алгоритмы выделения дисковой памяти и способы учета свободной и занятой дисковой памяти. Эти два вопроса: выделение дисковой памяти и учет свободной и занятой памяти, естественно, связаны между собой. Наиболее важным является вопрос: как описываются отдельные блоки файла, то есть как связываются файлы с блоками дисков? В разных операционных системах исторически использовалось несколько методов выделения файлу места на диске. Для каждого из этих методов запись в каталоге (которая соответствует символьному имени файла) содержит некоторый указатель (прямой и более сложный), следуя которому, можно найти все блоки данного файла.

Выделение непрерывной последовательностью блоков. Простейший способ - это хранение каждого файла как непрерывную последовательность блоков на диске. В этом случае файл характеризуется адресом (номером первого блока) и длиной (в блоках). Если файл начинается с блока b, то потом он занимает блоки b+1, b+2, ... b+n-1. У этой схемы есть два преимущества: во-первых, она очень легко реализуется, потому что для того, чтобы понять, где же находится файл, достаточно знать - где находится первый блок, во-вторых, этот способ обеспечивает хорошую

производительность, поскольку файл целиком можно прочитать в основную память за одну дисковую операцию. Можно читать меньше, чем блок, а можно читать больше, чем блок, главное - что при выполнении операции чтения не должно быть движения головок, то есть, если блоки физически находятся смежно, то их можно читать за один обмен с внешней памятью. Такой подход использовался в операционной системе CMS компании IBM, в операционной системе RSX-11 компании DEC для выполняемых файлов, для файлов с расширением exe. В действительности этот способ по простой причине распространен мало: в процессе эксплуатации любой магнитный диск представляет собой некоторую совокупность свободных и занятых фрагментов. Если посмотреть, как это будет происходить со временем, то может оказаться так, что, как обычно, есть много свободных фрагментов, но при создании нового файла не всегда удается найти размер, который для него пригоден. На самом деле проблема непрерывного расположения блоков файла может рассматриваться как частный случай более общей проблемы выделения блока или фрагмента необходимого размера из списка свободных блоков.

Как и при распределении основной памяти, типовыми решениями этой задачи являются стратегии первого подходящего, наиболее подходящего и наименее подходящего. Аналогичным образом, как и в схеме с динамическими разделами, этому методу свойственна проблема внешней фрагментации, которая возникает в большей или меньшей степени, в зависимости от объема самого магнитного диска и среднего размера файлов. Кроме того, непрерывное распределение внешней памяти не может быть использовано до тех пор, пока неизвестен хотя бы максимальный размер файла. Иногда размер выходного файла, который создается, легко оценить, например, если мы хотим просто скопировать существующий файл. Но чаще всего это сделать трудно, особенно в тех случаях, когда размер файла меняется. Например, это текстовый файл, который дописывается. Если выделенного места для этого файла не хватает, то пользовательская программу необходимо каким-то образом приостановить, чтобы было можно при последующем её перезапуске выделить дополнительное место для этого файла.

В некоторых операционных системах используются несколько усложненные варианты непрерывного выделения - это основные блоки файла и резервные блоки. Однако, если выделяются файлы из резервных блоков, то возникают те же самые проблемы, то есть приходится решать задачу выделения непрерывной последовательности блоков диска, но уже из совокупности резервных блоков. То есть в действительности, поскольку возникает внешняя фрагментация - единственным приемлемым решением проблем является периодическое уплотнение содержимого внешней памяти (сжатие или подвижка), цель которого состоит в объединении свободных участков в один большой свободный фрагмент. Эта операция дорогостоящая и опасная, потому что на самом деле, если что-нибудь произойдет в процессе подвижки, то можно лишиться множества фалов. Или надо делать очень хитрую подвижку. То есть, если содержимое дисковой памяти постоянно меняется, то

этот способ не является подходящим. Однако для **стационарных** файловых систем, например, для файловых систем, которые организуются на компакт-дисках (на оптических дисках) этот способ вполне пригоден. Особенно, если они без перезаписи, то есть один раз записав - известно, что туда пишется, известно - откуда пишется, известен размер - он никогда больше не меняется.

Связный список блоков

Внешнюю фрагментацию можно устранить, если представлять файл в виде связного списка блоков диска (таким образом, как показано на схеме). В этом случае опять необходимо хранить начало первого блока файла, из него стоит ссылка на следующий блок, и т.д. - до последнего блока данного файла. Понятно, что в этом случае логические блоки файла могут быть произвольным образом разбросаны по внешней памяти.

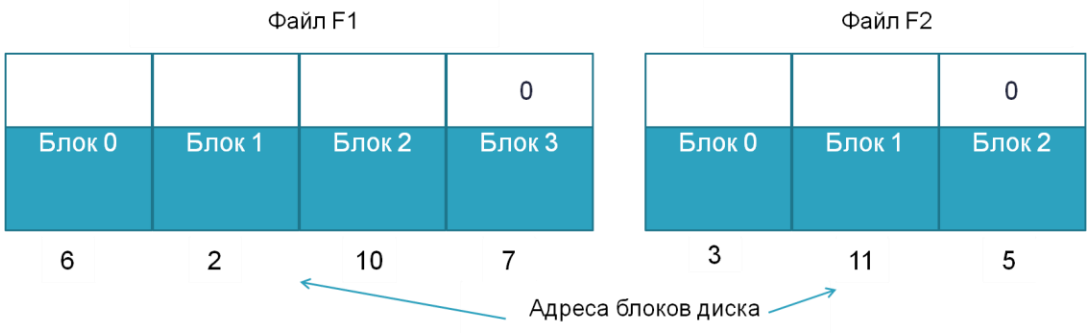

Рис. 12.2. Хранение файла в виде связного списка дисковых блоков

В этом случае запись в каталоге содержит указатель на последний блок файлов (на последний, чтобы файл было можно расширять). Иногда используют специальный знак конца файла в цепочке этих блоков - EOF, тогда можно последний не писать, но будет необходимо ходить по списку файлов, чтобы его найти. Каждый блок содержит указатель на следующий блок. Внешняя фрагментация отсутствует. Для удовлетворения запроса на расширение файла можно использовать любой свободный блок. Нет необходимости объявлять размер файла, когда он создается, файл может расти неограниченно, но имеется ряд существенных недостатков: во-первых, для того, чтобы произвести прямой доступ к i-му блоку - нужно последовательно пройти по блокам с 1-го до i-1, потому что у нас нет прямого указателя на i-й блок. То есть в действительности теряются все преимущества прямого доступа к файлу. Во-вторых, этот способ организации блочных файлов не очень надежен, потому что если вдруг какой-то блок в цепочке блоков перестает читаться, то теряются все данные в оставшейся части файла и потенциально теряется дисковое пространство, которое под этот файл было отведено. Мы просто его больше никогда не найдем. Наконец, для указателя на следующий блок - внутри блока необходимо каким-то образом выделить память. При том, что емкость блока традиционно является степенью двойки, если там

отобрать память под указатель, то эта емкость перестает являться степенью двойки. То есть это уже становится неудобно, поэтому метод связного списка в чистом виде, как правило, не используется.

Вариант - хранить указатели в индексной таблице в памяти, таблице отображения файлов - FAT (file allocation table). Здесь для каждого файла помечается его логические блоки, которые соответствуют физическому блоку. В каталоге, естественно, говорится, где находится первый блок этого файла, в каком месте таблицы FAT.

Номер блоков диска		
1		
2	10	
3	11	Начало файла F1
4		
5	EOF	
6	2	Начало файла F2
7	EOF	
8		
9		
10	7	
11	5	

В этом случае запись в каталоге содержит только ссылку на первый блок, после этого при помощи таблицы FAT можно найти блоки файла независимо от его размера. В тех элементах таблицы FAT, которые соответствуют последним блокам файлов, обычно записывается специальный признак - EOF. Достоинство подхода состоят в том, что по таблице FAT, поскольку она ориентирована на именно хранение данных о физических блоках фалов, можно судить о физическом соседстве блоков, которые располагаются на диске. Когда выделяется новый блок для данного файла, то можно попытаться найти свободный блок диска, который находится поблизости от других блоков данного файла. Минусом схемы является потребность хранения в основной памяти этой таблицы, которая в действительности (с ростом объемов магнитных дисков) становится все больше и больше.

Наконец, метод, который используется в файловых системах операционной системы Unix - это индексные узлы или i-node. Это очень похоже на то, как устроена таблица станиц при организации виртуальной памяти (но, конечно, немного не так). В этом случае для каждого файла заводится специальная область внешней памяти, которая называется индексным узлом - i-node (рис.12.3). В нем содержатся атрибуты файла и дисковые адреса блоков файла. В этом случае запись в каталоге для каждого файла содержит ссылку или адрес на индексный блок. По мере того, как файл заполняется, заполняется и его i-node. Этот подход широко распространен и, соответственно, поддерживает как последовательный, так и прямой доступ к файлу.

Обычно применяется такая комбинация одноуровневого и многоуровневых индексов. В этом случае первые несколько адресов блоков файла хранятся непосредственно в индексном узле, то есть для маленьких файлов после чтения блока i-node, мы получаем таблицу, которая показывает нам на расположение всех блоков данного файла.

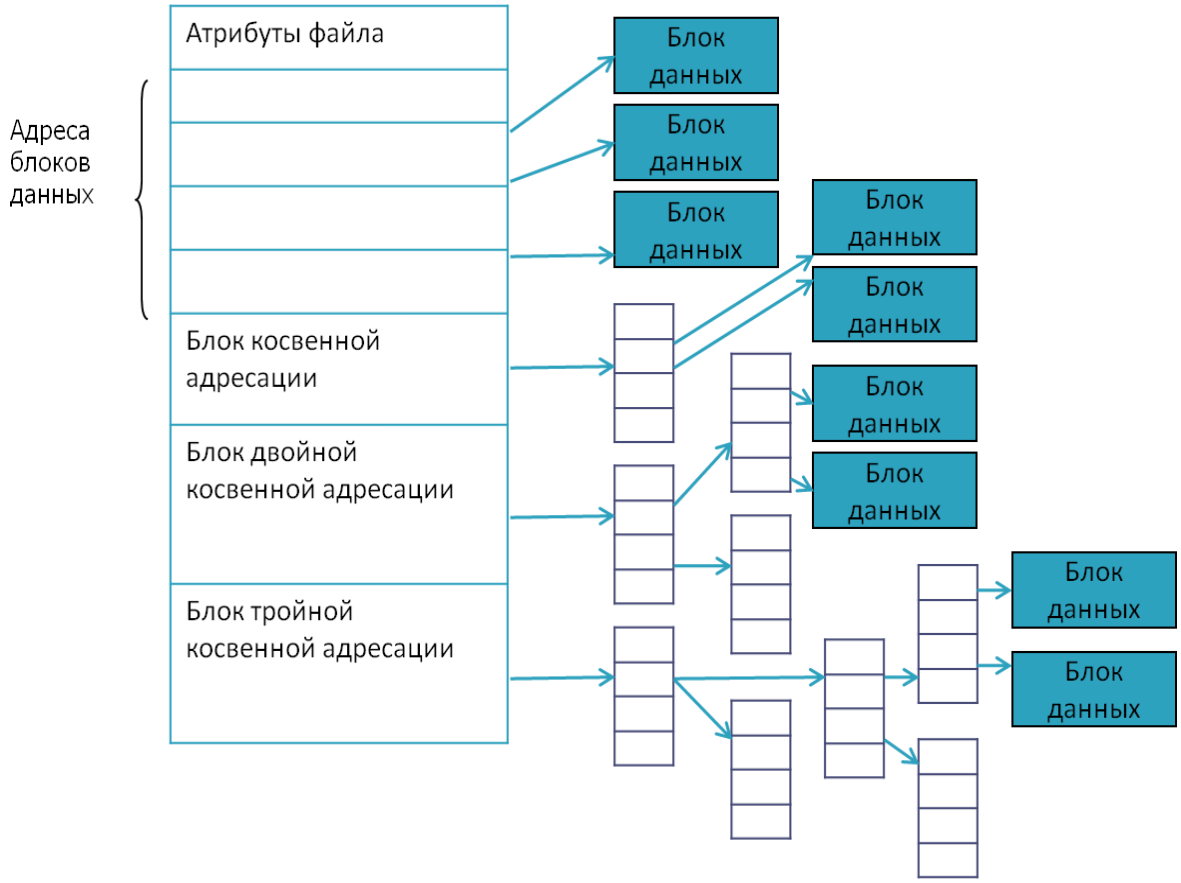

Рис. 12.3. Структура индексного узла

Если файл становится большего размера, то один из адресов (последний адрес i-node) указывает на следующий блок - это блок косвенной адресации. В этом случае каждый элемент каждой записи в этом блоке указывает на отдельный блок, который содержит список физических блоков. Если не хватает и этого - файл ещё больше, то используется блок двойной косвенной адресации, тройной косвенной адресации и т.д. То есть в действительности получается не очень хорошо, потому что чем больше файл, тем дороже добраться до его конца. До начала - легко и быстро добираться, а до конца - чем дальше, тем сложнее и дольше. Есть более равномерные способы, которые как бы выравнивают число обменов, которые необходимо произвести с внешней памятью, чтобы добраться до любого блока файла (независимо от его размера). Но это модификация того же самого подхода. Для больших файлов всегда не всё хорошо.

Управление свободным и занятым дисковым пространством

Система должна понимать, где брать блоки для файлов, которые растут (где их необходимо расширять), или для файлов, которые создаются. Необходимо, чтобы система понимала, как управлять дисковым пространством, которое сейчас не занимается ни одним файлом. В современных операционных системах применяется несколько способов учета используемого места во внешней памяти. Рассмотрим наиболее распространенные:

- **Учет при помощи организации битового вектора**. Видимо, самый распространенный способ - это битовый вектор. В этом случае вся внешняя память одного магнитного диска (неважно, логического или физического) представляется в виде битого вектора - bit map или bit vector. Каждый блок представлен одним битом, у которого, например, значение 0, которое означает, что бит занят, а 1 - что он свободен. Этот способ относительно прост и эффективно работает при нахождении первого свободного блока или n последовательных блоков на диске. Во многих компьютерах, во многих процессорах имеются команды манипулирования битовыми строками, их можно использовать для этой цели. Например, в некоторых процессорах Intel или Motorola имелись команды, при помощи которых можно найти первый бит со значением 1 в слове. То есть все равно необходимо длинный вектор перебирать по словам, но это довольно быстрая операция. Такой способ учета свободных блоков используется в разных операционных системах, в том числе в ранних операционных системах компании Apple. В российских операционных системах он тоже использовался. Хотя размер битового вектора - это, конечно, наименьший из всех возможных структур, сам вектор может оказаться достаточно большого размера, а его необходимо хранить в основной памяти, чтобы метод эффективно работал. Значит, это работает для относительно небольших дисков. Иногда, если битовый вектор становится слишком большим, слишком длинным, для ускорения поиска в нем он разбивался на регионы. Организуются такие сводные структуры данных, которые позволяют узнать о количестве свободных блоков для каждого региона битового вектора.

- **Учет при помощи организации связного списка.** Другой способ - связать в список свободные блоки, размещая указатель на первый свободный блок в специально отведенном месте памяти, но кэшируя эту информацию в основной памяти. Вообще-то, эта схема не всегда эффективна, потому что если мы хотим искать блоки среди свободных по каким-то критериям, то нужно выполнить много обращений к диску, потому что - это список. Как правило, правда, требуется только первый свободный блок. Иногда прибегают к модификации такого подхода связного списка, храня в первом свободном блоке адреса n свободных блоков. Тогда первые n-1 из этих блоков действительно используются, а последний блок содержит адреса других n блоков и т. д. Бывают

и другие методы, например, можно рассматривать весь набор свободных блоков диска как файл и вести для него соответствующий i-node, соответствующий индексный узел. Но мы видим, какие там недостатки, то есть в действительности к первому свободному блоку будет легко добраться, а к последнему, если вдруг это потребуется, добраться будет труднее.

Размер блока

Размер логического блока файла играет важную роль. В некоторых системах, в том числе в Unix размер логического блока файла может быть задан при форматировании диска. Если этот размер выбирается небольшим, то в каждом файле содержится много блоков, а поскольку при работе с дисками с подвижными головками чтение блока задерживается операциями передвижения магнитных головок и, соответственно, ожиданием времени подкрутки диска, то файл из многих блоков при прямом доступе будет читаться медленно. Большие блоки обеспечивают более высокую скорость обмена с дисками, но из-за внутренней фрагментации, соответственно, снижается процент полезного использования дискового пространства.

Для систем со страничной организацией памяти характерна сходная проблема с размером страницы. Как показали эмпирические исследования, большинство файлов имеют небольшой размер. Например, в Unix 85% файлов - совсем маленькие, меньше 8 Кбайт, 48% - менее 1 Кбайт. В системах с виртуальной памятью желательно, чтобы единицей пересылки между внешней и основной памятью была страница виртуальной памяти, наиболее распространенный размер страниц памяти - это 4 Кбайт. Поэтому обычный компромиссный выбор размера блока - это 512 байт (как было, например, на заре Unix), 1 Кбайт, 2 Кбайт или 4 Кбайт, то есть степени двойки.

Структура файловой системы на магнитном диске

Структура служебных данных типовой файловой системы на одном из логических дисков состоит из четырех основных частей: суперблок, структуры данных, которые описывают свободную дисковую память и свободный i-node, свободные индексные узлы, сам массив индексных узлов и блоки диска, которые хранят данные файлов.

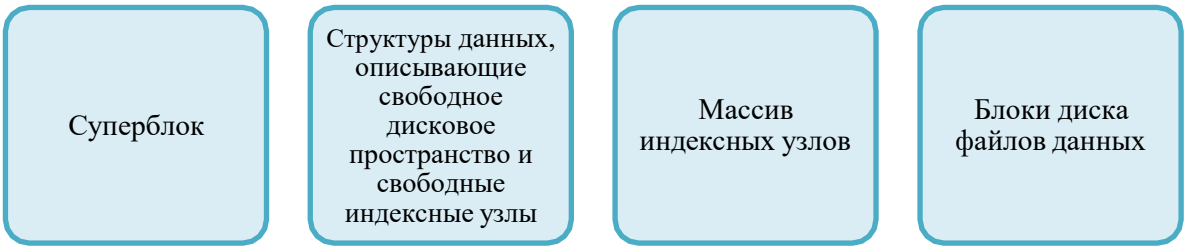

Рис. 12.4. Примерная структура файловой системы на диске

В начале каждого логического диска находится суперблок, который содержит общее описание файловой системы:

- тип файловой системы, как с ней полагается работать - это характеристика, когда можно использовать в одной операционной системе несколько разнотипных видов файловых систем;
- размер файловой системы в блоках;
- размер массива индексных узлов;
- размер логического блока

Эти структуры данных создаются на магнитном диске в результате его **форматирования**. Если они созданы, то система уже может считать, что там имеется файловая система, к ней можно обращаться соответствующими системными вызовами.

В современных файловых системах операционной системы Unix для повышения надежности хранения данных используется несколько копий суперблока. В действительности это повелось с такой замечательной файловой системы, которая появилась внутри операционной системы Unix BSD в 1984 году - это Fast File System, её автор Marshall K. McKusick. Это была файловая система, которая является прообразом большинства файловых систем, которые сегодня существуют. Там впервые использовалось несколько копий суперблока для того, чтобы было можно получать критически важную информацию, даже если диск слегка повредился. В некоторых версиях операционной системы Unix суперблок включал структуры данных, которые управляли распределением дисковой памяти. В этом случае суперблок непрерывно обновлялся, это снижало надежность файловой системы в целом. Более правильным решением явилось выделение тех структур данных, которые описывают свободную и занятую память на магнитных дисках в отдельной части описателя.

Массив индексных узлов (ilist) - это список блоков индексов, которые соответствуют файлам данной файловой системы. Размер массива индексных узлов определяется администратором при генерации системы. То есть в действительности этот размер определяет максимальное число файлов, которые могут быть созданы в файловой системе этого магнитного диска (физического или логического). Блоки данных хранят реальные данные файлов, размер логического блока данных задается при форматировании файловой системы. При заполнении диска с файловой системой содержательной информацией - используются блоки данных для хранения данных каталогов и обычных файлов. При этом изменяется массив индексных узлов и данные, которые описывают пространство диска. Один блок данных может принадлежать одному и только одному файлу в файловой системе, то есть ссылка на него может находиться только в одном i-node, в одном индексном описателе этой файловой системы и только в одном экземпляре.

Реализация каталогов

Каталог - это файл, который имеет вид таблицы и хранит список входящих в него каталогов или файлов. Основная задача каталогов - поддержка древовидной структуры файловой системы. Запись в каталоге для каждой операционной системы имеет свой собственный вид, он зачастую неизвестен пользователю и, вообще-то, и не должен быть ему известен, потому что это его не касается. Поэтому блоки данных файлов-каталогов заполняются не через обычные операции записи, а с помощью специальных системных вызовов, например, запись в каталог производится при выполнении системного вызова "создание файла".

Для доступа к файлу используется длинное путевое имя (pathname), которое сообщается пользователем (рис.12.5.). Каждый элемент каталога, каждая запись этой таблицы связывает имя файла или имя подкаталога (который фигурирует в этом каталоге) с блоками данных на диске.

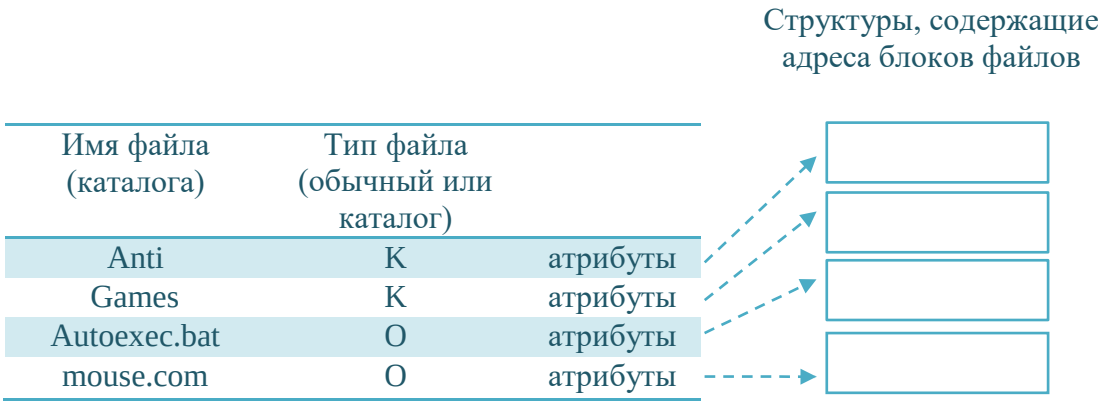

Рис. 12.5. Реализация директорий

Как мы обсуждали ранее, эта ссылка может быть номером первого блока или номером индексного узла, если задание отображения задается с помощью индексных узлов. Когда выполняется системный вызов open, то есть когда по требованию пользователей система открывает файл, прежде всего она ищет его имя в каталоге, но по коротким именам. Сначала в корневом каталоге, потом, соответственно, в каталоге, на который идет ссылка из корневого, и т.д. Затем из записи в каталоге или из структуры, на которую эта запись указывает, извлекаются атрибуты и адреса блоков файла на диске. Эта информация помещается в системную таблицу в основной памяти, именно она потом используется при всех последующих обращениях, где имеются ввиду какие-то действия с этим файлом (который открыт).

Как показано на рисунке 12.5., атрибуты файла можно хранить непосредственно в элементах каталога. Однако, если к файлам обеспечивается совместный доступ, и даже по некоторым другим причинам (когда к файлу обеспечивается несколько

способов доступа, несколько имен) - удобнее хранить атрибуты в индексном узле, как это делается в операционной системе Unix.

Примеры реализации

Каталоги в операционной системе MS-DOS. Конечно, каталоги могут содержать подкаталоги, и это позволяет конструировать произвольное дерево каталогов для того, чтобы использовать его при поиске файла по длинному путевому имени.

Разряды							
8	3	1	10	2	2	2	4
Имя файла	Расширение	Атрибуты (обычный файл или директория)	Резервное поле	Время	Дата	Номер первого блока	Размер

Если используется таблица отображения типа FAT, то номер первого блока, как показано выше, используется в этой таблице в качестве индекса.

Каталоги в операционных системах семейства Unix. Элементы каталога или запись таблицы, которая соответствует каталогу, содержат имя файла и номер его индексного узла. Как мы помним, для каждой файловой системы есть массив индексных узлов, поэтому номер соответствует номеру блока в этом массиве. Вся остальная информация о файле находится в описателе, который называется индексным узлом. Так было в начальных версиях Unix, а в более поздних версиях форма записи претерпела некоторые изменения, но суть осталась той же.

Байты	
2	14
Имя файла	Расширение

Изменения, например, касались того, что для имен файла появилась возможность быть неопределенного размера, поэтому это не такая простая организация, как была вначале, когда имена были фиксированной или ограниченной длины.

Поиск в каталоге. Обычно **с**писок файлов в каталоге по имени файла в файловой системе не упорядочивается, в каждом каталоге приходится производить поиск соответствующего короткого имени. Насколько быстро работает этот поиск - достаточно сильно влияет и на эффективность, и на достоверность, и на надежность файловых систем.

1. Самый простой способ - это **линейный поиск.** В этом случае каждый каталог всегда просматривается с самого начала, пока поиск не дойдет до конца (это означает, что файла с таким именем нет), либо не встретится нужный файл.

Поиск всегда необходимо доводить до конца, в худшем случае мы всегда просматриваем каталог целиком. Это, конечно, наименее эффективный способ поиска, но в большинстве случаев он работает с достаточной производительностью. В частности, когда операционная система Unix была молодой, её авторы утверждали, что линейного поиска (на фоне того, что требуются обмены с дисками) вполне достаточно. Метод линейного поиска простой, но тем самым требуются временные затраты, потому что поиск длится до конца, даже в случае плохого варианта. В частности, при создании нового файла сначала необходимо проверить каталог на наличие такого же имени. То есть в действительности, поскольку это бывает для новых файлов достаточно редко, это будет делаться почти всегда. Потом имя нового файла вставляется в конец каталога, когда мы убедились, что там необходимого имени нет. При удалении файла нужно тоже выполнить поиск его имени в этом списке и пометить запись, соответствующий элемент каталога как неиспользуемый. Настоящим, реальным недостатком этого метода является последовательный поиск файла в каждом каталоге. Из-за того, что информация о структуре каталога используется часто, неэффективный способ поиска может быть заметен пользователям. Можно свести поиск к двоичному поиску, если список файлов отсортировать. В этом случае мы, соответственно, разбиваем пополам и ищем в первой половине, если там нет - то ищем во второй. Смотрим, каким именем кончается первая половина, потом смотрим - где находится файл, разбиваем на четверти и т.д. В результате мы получаем вместо линейного поиска логарифмический поиск, фактически - это то же самое, что бинарное дерево. Однако поддержка упорядоченности по именам файлов усложняет создание и удаление файлов, поскольку каждый раз требуется переписывать достаточно большой объем информации в каждом каталоге.

2. Самый эффективный способ - это **хеш-таблица**. В этом случае имена файлов хранятся в каталоге в виде обычного, то есть последовательного линейного списка. Но при этом дополнительно используется хеш-функция, которая позволяет по имени файла получить указатель на имя файла в списке. Почти всегда мы будем за одно обращение получать правильное имя. Если каталог перегружен, то там могут быть коллизии, но это, как правило, максимум две попытки для того, чтобы найти имя. Но чтобы убедиться, что его нет, будет необходимо просмотреть все элементы с одним значением хеш-функции до конца. В результате, конечно, время поиска может существенно уменьшиться, если каталоги не перегружены. С коллизиями борются следующим образом: дополнительные списки от элемента, который вызвал коллизию, то есть список элементов, которые обладают таким же значением свертки, продолжается дальше. В действительности число коллизий зависит в основном от того, к каким именам склоняются пользователи, выбрана ли хеш-функция, которая более-менее соответствует поведению пользователей файловой системы. Также

это сильно зависит от того, насколько каждый каталог нагружен, то есть на самом деле известно, что хеширование работает хорошо до тех пор, пока хеш-таблица не слишком сильно заполнена. Выбор подходящего алгоритма хеширования позволяет свести к минимуму число коллизий. Конечно, если хеширование вырождается, то есть очень большому числу имен хеш-функция ставит в соответствие одно и тоже значение свертки, то фактически преимущества схемы с хешированием утрачивается (по сравнению с последовательным поиском). По идее, необходимо просто менять функцию хеширования и, соответственно, переделывать все каталоги, которые были сделаны в расчете на предыдущую функцию хеширования.

3. Помимо описанных методов поиска имени файла в каталоге существуют и другие методы. В качестве примера можно привести организацию поиска в каталогах файловой системы NTFS при помощи B-дерева - это ветвистые сбалансированные деревья. В действительности - это спорный вопрос, насколько в каталогах деревья вообще уместны, потому что B-деревья хорошо работают, когда много элементов. Если для каждого каталога сделать отдельное дерево, то эти деревья будут очень маленькие и толку от них может быть не так много, как накладных расходов.

Монтирование файловых систем

Системный вызов **mount** связывает файловую систему из указанного раздела на диске, то есть с указанного физического или логического диска, на котором существует файловая система, с существующей иерархией файловых систем. Есть парный системный вызов "umount", который позволяет отключить ранее примонтированную файловую систему из иерархии. Утверждают, что это необходимо делать, например, когда администратор по какой-то причине хочет администрировать диск, на котором располагается эта файловая система, не выключая систему. Тогда желательно, чтобы пользователи туда не попадали, для этого необходимо отключить его от общей иерархии файловых систем. Mount обеспечивает пользователям возможность обращаться к данным в дисковом разделе как к файловой системе (имеется ввиду, как это сделано в Unix), а не как к последовательности дисковых блоков.

Предположим, что есть корневая файловая система, которая резидентна в операционной системе Unix. Самой операционной системе известно, на каком диске и в каком месте находится корневой каталог файловой системы. Есть другая, отдельная файловая система, которая была создана на отдельном диске. В этой корневой файловой системе есть пустой файловый каталог, который мы используем как точку монтирования. Тогда при выполнении системного вызова "mount" корневой каталог отдельной файловой системы накладывается на пустой каталог корневой файловой системы, и образуется полная иерархия, которая находится на рис. 12.6. справа.

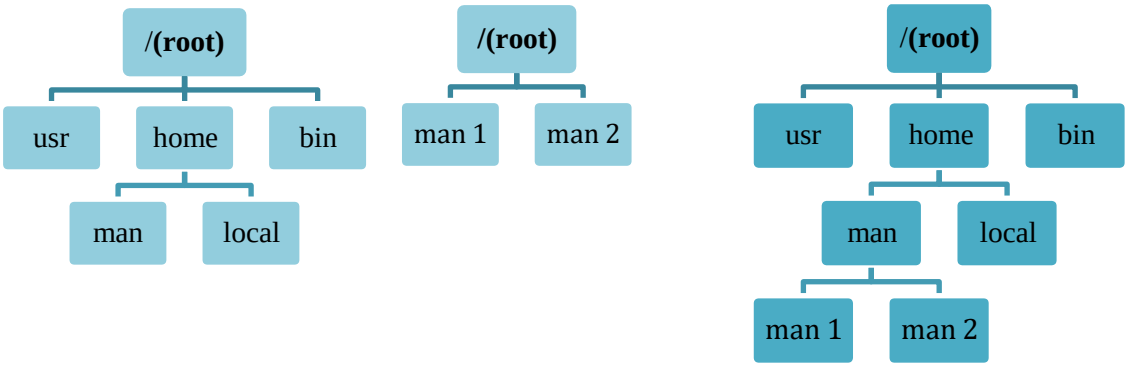

Рис. 12.6. Две файловые системы до монтирования и общая файловая система после монтирования

В операционной системе Unix системный вызов "mount" имеет следующий вид: mount (special pathname, directory pathname,options)

- **special pathname** - это имя специального файла устройства, которое соответствует диску смонтированной файловой системы. Это просто указание - какой драйвер необходимо использовать для доступа к этому диску;
- **directory pathname** - это каталог в существующей файловой системе, который является точкой монтирования, на который мы хотим наложить корневой каталог этой новой файловой системы;
- **options** указывает, как монтировать файловую систему, например, её можно монтировать только для чтения, тогда для файлов, которые находятся в монтированной системе, не будут выполняться системные вызовы, которые что-то меняют (write и creat).

При выполнении системного вызова "mount" операционная система должна убедиться, что устройство (указанный дисковый накопитель) действительно содержит файловую систему ожидаемого формата, что там есть суперблок, массив индексных узлов, заведен корневой каталог. Некоторые операционные системы выполняют системный вызов "mount" автоматически, как только встретят диск, содержащий файловые системы в первый раз, когда видят, что на устройстве такой диск стоит. Тогда жесткие диски монтируются на этапе загрузки, а флэш-накопители - когда они вставляются в соответствующий разъем.

Если файловая система на устройстве имеется, то она монтируется на корневом уровне, при этом к цепочке имен абсолютного имени файла (pathname) добавляется буква раздела. Ядро поддерживает таблицу монтирования с записями о каждой смонтированной файловой системе. В каждой записи содержится информация о вновь смонтированном устройстве, о его суперблоке, о его корневом каталоге и точке монтирования. В таблицу монтирования занесение информации во время выполнения системного вызова производится сразу, иначе между процессами может возникнуть

конфликт. Например, если процесс, в котором выполняется mount - приостановлен для открытия устройства или считывания суперблока файловой системы, а другой процесс тоже попытается смонтировать файловую систему. Наличие в логической структуре файлового архива точек монтирования требует, чтобы алгоритмы, которые осуществляют навигацию по каталогам, были сделан весьма аккуратно и тщательно. То есть точку монтирования можно пересечь двумя способами: можно идти через корневую файловую систему, проходя сверху вниз, или можно идти наоборот - из смонтированной файловой системы вверх по каталогам к корневой файловой системе. Алгоритмы поиска файлов должны предусматривать ситуации, в которых очередной компонент пути к файлу является точкой монтирования и тогда, вместо анализа индексного узла очередного каталога, нужно обрабатывать суперблок монтированной системы. То есть необходимо фактически переходить на другую файловую систему и продолжать поиск по её дереву каталогов.

Связывание файлов

Поскольку базовой организацией архива файлов является дерево, то в ней невозможно связать потомков более, чем с одним предком. Фактически это означает, что один файл нельзя назвать разными именами, то есть у него должен быть один путь в такой иерархии - от корня к любому файлу. Это - негибкость, она частично устраняется за счет возможности связывания файлов с разными именами или организации линков (link). Ядро позволяет пользователю с помощью специальных системных вызовов связывать каталоги, упрощая написание программ, которые требуют прохождения больших путей по файловым системам.

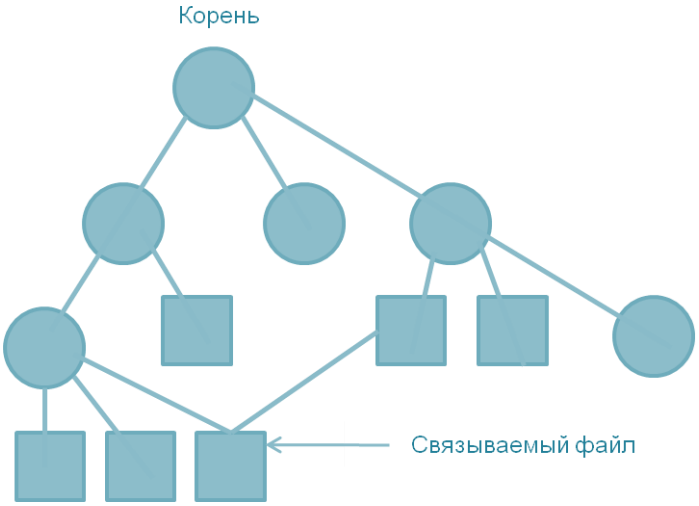

Рис. 12.7. Структура файловой системы с возможностью связывания файла с новым именем

На рисунке 12.7. показано, например, что некоторый пользователь написал программу, которую он разместил в собственном рабочем каталоге. Остальные люди

попробовали эту программу и сказали, что она замечательная, что с ней удобно работать, но именовать её неудобно, потому что для них идти необходимо от корня (через весь путь). Тогда для этого файла можно сделать альтернативное имя. Если есть каталог, в котором находятся какие-то общедоступные программы, тогда для того же самого выполняемого файла можно завести другое имя, которое будет вести по более короткому пути - через каталог, который другим пользователям понятен. То есть часто в действительности имеет смысл хранить под разными именами одну и ту же команду (выполняемый файл). Например, выполняемый файл традиционного текстового редактора операционной системы Unix - vi обычно может вызываться под именами ex, edit, vi, view и vedit файловой системы.

Такое соединение между каталогом и файлом, который именуется разными именами, называется связями или ссылками (link). В этом случае дерево файловой системы превращается в ациклический граф. Это удобно, но создает ряд дополнительных проблем.

Простейший способ реализовать такое связывание файла с разными именами - просто продублировать информацию о его атрибутах и месторасположении в нескольких каталогах. Но проблема совместимости может возникнуть в тех случаях, когда владельцы этих каталогов попытаются независимо друг от друга изменить содержимое файла. Например, в старой операционной системе CP/M запись в каталоге о файле сразу содержит адреса дисковых блоков. Тогда, если мы хотим дать ему ещё и другое имя, необходимо сделать копии тех же самых номеров блоков, которые должны быть помещены в другой каталог, когда выполняется операция link. Тогда, если один из пользователей по какому-то пути что-то добавляет к файлу, новые блоки появятся только у него в каталоге и не будут видны в каталоге другого пользователя.

Такие проблемы можно разрешить двумя разными способами:

- Механизм так называемых **жестких связей - hard link.** Если, как это сделано в Unix, соответствие логических и физических блоков файла производится не в каталоге, а в отдельной структуре данных, связанной собственно с файлом, например, в i-node, то когда устанавливается связь, не надо копировать эти данные. Необходимо просто поставить в каталоге ссылку на тот же самый индексный узел.

- Альтернативное решение - это создание нового имени, которое является новым путем к связываемому файлу. Этот подход называется **символическим** или **мягким связыванием - soft** или **symbolic link**. В этом случае создается как бы новый файл, но новое имя связывается не с индексным описателем, а со старым именем этого файла. Поскольку старое имя - длинное имя от корня, то его необходимо где-то хранить, поэтому в действительности для символьной связи создается отдельный индексный узел, даже может заводиться отдельный блок данных для хранения длинного (исходного) имени файла.

У каждого из этих методов имеются свои минусы. Если говорить про жесткую связь, то возникает потребность поддерживать счетчик числа ссылок среди атрибутов файла, то есть сколько путей к нему сейчас существует. Это осуществляется для корректной реализации операции удаления файла. Например, в Unix такой счетчик является одним из атрибутов, который хранится в i-node. При выполнении операции remove каким-то пользователем по одному из имен, которые связаны жесткой ссылкой с одним и тем же файлом, в этом атрибуте уменьшается число ссылок на 1, но файл не уничтожается. Реальное удаление файла происходит, когда при выполнении операции remove число ссылок на файл в этом атрибуте становится равным 0.

В случае символического связывания или мягкого связывания такая проблема не возникает, поскольку ссылка на индексный узел файла имеется только у реального владельца файла. В действительности открытие файла по имени, которое создано через мягкое связывание, работает следующим образом: идет поиск по этому имени, пока мы не находим тот каталог, который является последним в этой цепочке; в нем всему этому пути соответствует некоторое альтернативное исходное длинное имя; поиск теперь продолжается снова от корня, но уже по другому имении. В этом случае, если владелец файла его удаляет, то все мягкие связи остаются существовать, но система при попытке открыть файл, когда будет его искать по исходному имени - убедится, что его на самом деле нет. Поэтому все попытки с ним работать не увенчаются успехом. Тем самым удаление символической связи на файл тоже никак на него не влияет, просто пропадает ещё один путь к этому файлу. То есть в действительности проблема организации символической связи - это более длинное выполнение операции open, то есть необходимо дольше искать файл. Тем самым необходимо выполнять дополнительное обращение к внешней памяти, если это потребуется.

Преимущество символической связи состоит в том, что она может использоваться для организации распределенных файловых систем. Вообще говоря, символьная связь допускает образование в архиве файлов циклов, то есть на самом деле можно с помощью символического имени попасть на тот же самый каталог, с которого мы пошли искать альтернативное имя. В действительности обычно такие циклы не ищутся, а обнаруживаются, когда уже производится поиск. То есть это - трудоемкая процедура, в таких графах, которые соответствую файловой системе с символическими связями, необходимы специальные утилиты, необходимо много раз обходить каталоги файловой системы. То есть это довольно опасный механизм. Поскольку здесь нет никаких ограничений на ссылки, то это запросто можно использовать и для организации сквозного именования для файлов, которые находятся на другой машине и связаны через сеть в распределенную файловую систему.

Кооперация процессов при работе с файлами

Как можно совместно работать с файлами нескольким процессам или нескольким пользователям? Файлы - это ресурсы, которые являются разделяемыми, то есть их можно использовать совместно в нескольких процессах. Как и в случае любого

совместно используемого ресурса, в этом случае процессы должны синхронизировать доступ к таким файлам и каталогам. Например, если разрешить нескольким пользователям одновременно редактировать какой-либо файл без синхронизации, то результат будет совершенно непредсказуем. Он будет зависеть от того, в каком порядке производится реальная запись в файл. Если в одном процессе выполняется только операция "чтение", если при этом разрешить другому процессу одновременно модифицировать тот же самый файл, то в результате следующая операция read может дать другой результат, что для многих приложений не является приемлемым.

Системный вызов, который позволяет устанавливать и проверять блокировки файла, входит в обязательный набор системных вызовов современных многопользовательских операционных систем. В принципе, было бы логично связать синхронизацию доступа к файлу как к единому целому с системным вызовом open. Тогда открытие файла в режиме записи или обновления означает его монопольную блокировку соответствующим процессом, а открытие в режиме чтения - совместную блокировку. Так и надо работать, однако в операционной системе Unix это не так, чему имеются исторические причины. Мы нагрузили этот системный вызов ещё и функцией синхронизации доступа. В первой версии системы Unix механизм захвата файла отсутствовал. Система позволяла любому числу процессов одновременно открывать один и тот же файл в любом режиме без синхронизации. Вся ответственность за корректность совместной обработки файла ложилась на использующие его процессы, и система даже не предоставляла каких-либо особых средств для синхронизации доступа процессов к файлу. Впоследствии в версию V системы были включены механизмы захвата файла и записи, базирующиеся на системном вызове fcntl.

Допускаются два варианта синхронизации: с ожиданием, когда требование блокировки может привести к откладыванию процесса до того момента, когда это требование может быть удовлетворено, и без ожидания. В Unix практически для любого системного вызова, выполнение которого может привести к блокировке процесса, есть альтернативный системный вызов с флагом no wait, который не приводит к блокировке процесса, если условие синхронизации не выполнено, а всего-навсего сообщает процессу, что не получилось. Это касается и семафоров, и очередей сообщений, и сокетов. Установленные блокировки относятся только к тому процессу, который их установил, и не наследуются процессами-потомками этого процесса. Естественно, потому что какая иначе могла бы быть синхронизация, если бы они не синхронизировались бы между собой. Если некоторый процесс пользуется возможностями системного вызова fcntl, то другие процессы могут работать с этим файлом без всякой синхронизации. В действительности это сработает только тогда, когда есть межпользовательская договоренность, что с этими файлами будем работать хорошо, но если придет какой-нибудь нехороший человек и напишет соответствующую программу, то он сможет с ним работать без всякой синхронизации.

Более тонкие случаи синхронизации затрагивают блокировку отдельных структур ядра, которые отвечают за работу с файлами. Например, в операционной системе Unix, когда выполняется системный вызов, поддерживающий какую-то операцию с файлом, как правило, происходит блокировка i-node (индексного описателя), который содержит адреса блоков данного файла. На первый взгляд может показаться, что запрет более, чем одному процессу работать с файлом во время выполнения системного вызова - является излишней, поскольку почти всегда выполнение системных вызовов и так не прерывается. Однако в данном случае это не так: операции чтения и записи занимают продолжительное время и только запускаются, инициируются центральным процессором, а происходят они в действительности в основном на уровне обращений к внешней памяти. Поэтому установка блокировки на время системного вызова является гарантией атомарности операций чтения и записи. Достаточно оказывается заблокировать один из буферов кэша, в заголовке которого ведется список процессов, которые ожидают освобождения данного буфера. В соответствии с семантикой операционной системы Unix изменения, которые делаются одним пользователем, становятся видны другим пользователям, которые поддерживают файлы открытыми одновременно с первым пользователем.

Примеры разрешения коллизий и тупиковых ситуаций. Логика работы системы в сложных ситуациях может проиллюстрировать особенности организации мультидоступа. В качестве примера можно рассмотреть **образование потенциального тупика/deadlock**, когда создается жесткая связь (link), при возможности совместного доступа к файлу. Пускай два процесса одновременно выполняют следующие системные вызовы:

- процесс A: link("a/b/c/d","e/f/g"), то есть мы создаем новый путь "a/b/c/d" к файлу, который находится по пути "e/f/g"
- процесс B: link("e/f","a/b/c/d/ee"), создаем новое путевое имя "e/f" к файлу, который находится по пути "a/b/c/d/ee"

Предположим, что процесс A обнаружил индекс файла "a/b/c/d" в тот самый момент, когда процесс B обнаружил индексный узел файла "e/f". То есть системой с помощью чередования каких-то операций достигнуто состояние, при котором каждый процесс получил искомый индексный узел. Теперь, когда теперь процесс A попытается получить доступ к индексному узлу "e/f" - он приостанавливает свое выполнение до тех пор, пока не освободится индексный узел файла "f". Процесс B пытается получить индексный узел каталога "a/b/c/d" и приостанавливается в ожидании освобождения индексного узла "d". В результате процесс A будет удерживать заблокированным индексный узел, нужный процессу B, а процесс B будет удерживать заблокированным индексный узел, необходимый процессу A. Для предотвращения этого классического примера deadlock в файловых системах принято, чтобы ядро освобождало индексный узел исходного файла сразу после увеличения значения счетчика связей. Тогда, поскольку первый из ресурсов (индексный узел) свободен при обращении к

следующему ресурсу, взаимной блокировки, то есть блокировки циклической - не происходит. Deadlock тем самым удается избежать.

Достаточно много бывает поводов для нежелательной **конкуренции** между процессами, особенно **при удалении** имен каталогов. Предположим, что один процесс пытается добраться до файла по его полному символическому имени, последовательно проходя один компонент за другим, а другой процесс удаляет каталог, имя которого входит в путь поиска. Допустим, что первый процесс А ищет файл по имени "a/b/c/d" и приостанавливается для получения доступа к индексному узлу файла каталога "c". Он может приостановиться при попытке заблокировать этот индексный узел или при попытке обратиться к дисковому блоку, где этот индексный узел хранится. Если процессу В нужно удалить связь для каталога с именем "c", он может приостановиться по той же причине, что и процесс А. Предположим, что ядро впоследствии решит возобновить процесс В раньше процесса А. Тогда, прежде чем процесс А продолжит свое выполнение, процесс В завершится, удалив связь каталога "c" и его содержимое по этой связи. Позднее процесс А попытается обратиться к несуществующему индексному узлу, который уже удален. В действительности алгоритм поиска файла, который проверяет, что значение счетчика связей > 0, должен сообщить об этой ошибке. Можно привести и другие примеры, показывающие потребность в тщательном проектировании файловой системы для ее последующей надежной работы, особенно в условиях мультидоступа.

Надежность файловой системы

В жизни, особенно в компьютерной жизни может быть масса неожиданных и неприятных ситуаций, а разрушение файловой системы часто более опасно, чем разрушение компьютера. Просто потому, что тогда пользователь или организация теряют информацию, что действительно обходится и тяжелее, и дороже, это удается с большим трудом каким-то образом преодолевать. Поэтому файловые системы должны разрабатываться с учетом такой возможности. Кроме очевидных решений, таких как, например, своевременное дублирование информации, то есть выполнение резервного копирования - backup, файловые системы современных операционных систем содержат специальные средства для поддержки собственной совместимости.

Целостность файловой системы. Важным аспектом надежной работы файловой системы является контроль ее целостности. В результате файловых операций блоки диска могут считываться в основную память, модифицироваться и потом записываться обратно на диск. Многие файловые операции сразу затрагивают несколько объектов файловой системы, например, копирование файла предполагает выделение ему блоков диска, формирование индексного описателя, изменение содержимого каталога и т.д. Между этими шагами информация в файловой системе может оказываться просто несогласованной. Если из-за непредсказуемого останова системы, например, в следствии выключения электропитания, на диске будут сохранены изменения только части этих объектов, то файловая система на диске может

быть оставлена в нецелостном, несогласованном состоянии. В результате могут возникнуть нарушения логики работы с данными, например, могут появиться потерянные блоки диска, которые не принадлежат ни одному файлу, в то же время помеченные как занятые. Или наоборот - блоки, которые помечены как свободные, но в то же время занятые, то есть на них есть ссылка в индексном блоке. Первая ситуация, когда блоки не заняты, но не помечены как свободные - это обычно в операционных системах страшным не считается. В действительности всё, чем это грозит операционной системе или её пользователям - что будет недостаточно эффективно использоваться внешняя память. Вторая ситуация, когда возникают дважды занятые блоки, когда есть вероятность, что один и тот же блок будет одновременно приписан нескольким файлам, тогда пойдут несанкционированные параллельные изменения - эта ситуация действительно страшная, то есть её быть не должно.

В современных операционных системах предусмотрены меры, которые позволяют свести к минимуму ущерб от порчи файловой системы и полностью или частично позволяют потом восстановить ее целостность:

- Во-первых, это **порядок выполнения операций**. Для правильного функционирования файловой системы отдельные составляющие объектов, которые хранятся в памяти, неравноценны. Например, искажение внутри содержимого пользовательских файлов не приводит к серьезным последствиям с точки зрения файловой системы, в то время как несоответствия в файлах, которые содержат управляющую информацию (такие, как каталоги), могут быть катастрофическими. Нельзя относиться к файлам таким образом: считая, что система гарантирует, что пользователь никогда ничего не потеряет, потому что она этого не гарантирует. В действительности файлы - это рисковые контейнеры. Любой пользователь должен быть готов к тому, что у него любой файл после последнего изменения может каким-то образом утратить это изменение. А каталоги касаются всех файлов, поэтому если в нем что-то будет не так, то мы теряем весь архив файлов, что гораздо более опасно и имеет катастрофические последствия. Поэтому необходимо тщательно продумывать порядок выполнения операций со структурами данных файловой системы. Например, рассмотрим процесс создания жесткой **связи** для файла. Для этого файловой системе необходимо создать новую запись в каталоге, который указывает на уже существующий **индексный узел** файла, и после этого (или одновременно) увеличить счетчик **связей в индексном узле**. Если отказ произойдет между 1-й и 2-й операциями, то в каталогах файловой системы будут иметься два имени файла, которые ссылаются на один и тот же индексный узел со значением счетчика связей, равным 1. Если теперь будет выполнена операция remove по отношению хотя бы к одному из этих имен файлов, то это приведет к удалению файла как такового, то есть он просто исчезнет. Если же порядок операций изменен (сначала увеличивается счетчик связей, а потом создается новый элемент каталога), то у файла будет несуществующая жесткая

связь (помеченная, что она есть, а на самом деле её нет), но существующая запись в каталоге будет правильной. То есть это тоже является ошибкой, но ее последствия менее серьезны, чем в предыдущем случае. Всё, чем это грозит в действительности - за этим файлом будет необходимо следить администратору, так как его нельзя уничтожить на уровне пользователя.

- Другим средством поддержки целостности является **журнализация**. В этом случае последовательность действий с объектами во время выполнения файловой операции протоколируется, то есть регистрируется в системном журнале. Если происходит аварийный останов системы, то при в наличии такого журнала изменений можно выполнить откат системы в исходное целостное состояние, в котором она пребывала до начала выполнения этой операции. Такая избыточность поддержки журнала может стоить дорого, но она оправдана, поскольку в случае аварийного отказа позволяет реконструировать потерянные данные. На самом деле это не совсем данные - это метаданные, потому что данными обычно называют то, что хранится в обычных файлах. Всё остальное в файловой системе - это метаданные, которые позволяют добираться к файлам, контролировать правомочность доступа и т.д. То есть это метаданные, описывающие данные. Для того, чтобы было можно выполнить откат, требуется, чтобы для каждой, протоколируемой в журнале операции имелась обратная операция. Протоколируются не все изменения, а лишь изменения метаданных, то есть индексных узлов, записей в каталогах и др. Изменения в файлах пользователя в журнал не заносятся. Если бы в действительности в журнал заносились ещё и пользовательские изменения, то, во-первых, мы действительно бы поднимали файловые системы практически на уровень системы управления базами данных, то есть сильно понижали бы у неё производительность. Кроме того, в журнал записей должны проталкиваться изменения, чтобы можно было в случае сбоя ими воспользоваться. Если все изменения будут проталкиваться во внешнюю память, то кэширование в основной памяти просто теряет смысл. Примерами файловых систем с журнализацией являются NTFS, Ext3FS, ReiserFS и т.д. Чтобы подчеркнуть сложность задачи, необходимо заметить, что имеются не очень тривиальные проблемы, связанные с процедурой откатов. Например, при отмене изменений могут затрагиваться данные, которые уже использованы другими файловыми операциями, то есть такие операции также должны быть отменены. То есть возникает то, что в контексте баз данных называют каскадными откатами транзакций - это очень болезненная вещь, здесь уже необходимо переходить на термины баз данных и рассказывать, что можно сделать при управлении ими, чего пытаются избежать.

- Проверка целостности файловой системы **при помощи утилит**. Если все-таки нарушение целостности произошло, то для устранения проблемы несогласованности метаданных можно прибегать к утилитам (таким, как классические fsck, chkdsk, scandisk и т.д.), которые проверяют целостность

файловой системы. Они могут запускаться сразу после запуска операционной системы или после перезапуска после сбоя. Они проводят многократное сканирование разнообразных структур данных файловой системы в поисках противоречий. Это действительно длинная цепочка, длинная проверка. Возможны эвристические проверки (случайные), например, нахождение индексных узлов, у которых номер превышает доступное их количество на соответствующем разделе диска, или поиск в каталогах пользователей файлов, у которых владельцем является суперпользователь.

К сожалению, средств, которые гарантировали бы абсолютную сохранность данных в файлах - нет. Когда нужно гарантировать целостность метаданных – обычно, тем не менее, прибегают к дублированию, то есть к использованию зеркалированных файловых систем и т.д.

Управление "плохими" блоками

Для дисков с подвижными головками, для любых устройств внешней памяти наличие дефектных блоков на диске - дело обычное. Внутри блока, наряду с данными хранится контрольная сумма данных. Наверное - самый частый случай, когда это побайтовое циклическое сложение. То есть примерно так же, как контроль памяти – это побитовое циклическое сложение битов, которое присутствует в основной памяти, контроль дисковой памяти - это побайтовое циклическое сложение. Под "плохими" блоками обычно понимаются такие блоки диска внешней памяти, для которых вычисленная после чтения контрольная сумма считываемых данных, не совпадает с контрольной суммой, которая хранилась на диске после записи. Обычно дефектные блоки появляются на дисках в процессе эксплуатации, иногда они известны ещё на стадии поставки магнитных дисков, поскольку производители магнитных дисков не всегда могут сделать диск полностью свободным от дефектных блоков.

Рассмотрим два решения проблемы дефектных блоков, одно решение на уровне аппаратуры, другое - на уровне ядра операционной системы:

1. **На уровне аппаратуры** - это хранить список плохих блоков в контроллере магнитного диска. Когда контроллер инициализируется, то есть запускается после включения питания, он читает плохие блоки и заменяет каждый дефектный блок резервным, помечая отображение в списке плохих блоков. То есть в действительности в этом случае все реальные запросы, все обмены будут перенаправляться с блока, который был дефектным, к резервному блоку. Следует иметь ввиду, что при этом механизм подъемника - это наиболее распространенный механизм обработки запросов к блокам диска - будет работать неэффективно. Дело в том, что имеется некая стратегия очередности обработки запросов к диску, она диктует направление движения пакета головок магнитного диска к нужному цилиндру. То есть на самом деле оптимизация заключается в том, что если обмен с диском не противоречит один другому, то

выбирается такая их последовательность, которая направляла бы движение пакета магнитных головок, например, к центру дискового пакета. Однако обычно резервные блоки размещаются на внешних цилиндрах, если плохой блок при этом расположен на внутреннем цилиндре (где-то близко к центру) и контроллер делает подстановку прозрачным образом (то есть заменяет, когда с точки зрения операционной системы выбран правильный порядок обменов), то кажущееся движение головок, которое операционная система считает последовательным перемещением к центру, будет в действительности происходить не к внутреннему цилиндру (как думает операционная система), а к внешнему. Это, конечно, является нарушением стратегии, поэтому и минусом этой схемы замены дефектных блоков.

2. **На уровне операционной системы**, на уровне ядра решение может быть следующим: во-первых, необходимо сконструировать файл, который содержит дефектные блоки, то есть файл только из дефектных блоков, тем самым они тогда изымаются из списка свободных блоков, после этого нужно этот файл скрыть от прикладных программ.

Производительность файловой системы

Обращение к диску - это операция относительно медленная, поэтому ключевой задачей всех алгоритмов, которые работают с внешней памятью, является минимизация количества таких обращений. Типичной техникой повышения скорости работы с диском и уменьшения, соответственно, числа обращений к реальным внешним устройствам является **кэширование.**

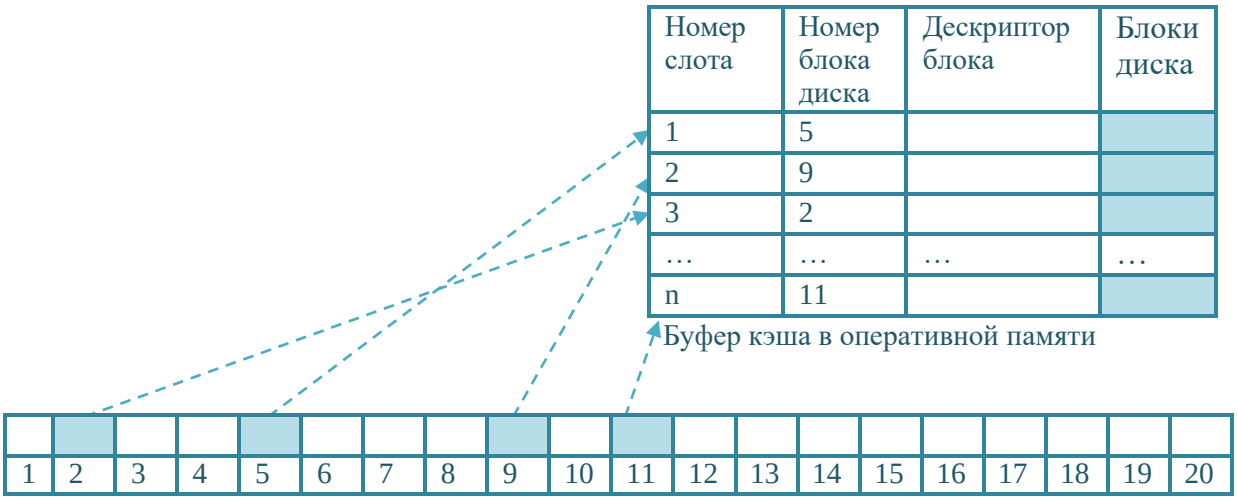

Рис. 12.8. Структура блочного кэша

Кэш диска – это буфер основной памяти, который содержит ряд копий блоков магнитного диска (рис. 12.8). Если от пользовательских программ приходит запрос на

чтение или запись блока диска, то сначала производится проверка на предмет наличия копии этого блока в кэше, если копия необходимого блока в кэше содержится, то эта операция выполняется без обращения к внешней памяти. В противном случае - запрошенный блок считывается в кэш с магнитного диска.
Сокращение числа дисковых операций оказывается возможным из-за присущего операционной системы свойства локальности. В действительности, если у всех процессов есть свойство локальности, то в целом и у кэша операционной системы будет какая-то локальность. Однако для аккуратной реализации кэширования требуется решить нескольких проблем:

- Во-первых, емкость буферного кэша ограничена. Может так оказаться, что при необходимости загрузить копию блока в заполненный буфер кэша, будет необходимо выполнить замещение блоков, то есть удалить оттуда какие-то копии блоков (может быть, во внешнюю память). Здесь используются те же самые стратегии, что и при управлении виртуальной памятью, то есть те же FIFO, Second Chance и LRU-алгоритмы замещения. Но, видимо, в глобальном режиме, потому что кэш - один, здесь про какие-то рабочие наборы говорить трудно. С другой стороны, замещение блоков должно производиться с учетом их важности для файловых систем, поэтому блоки должны быть разделены на категории, например, блоки **индексных узлов** - это основные блоки, где находятся ссылки на первые блоки файла, блоки косвенной адресации - для более крупных файлов, блоки каталогов, заполненные блоки данных и т.д. В зависимости от того, к какой категории принадлежит соответствующая страница, содержащая копию блока, к ним можно применять разную стратегию замещения.

- Во-вторых, при кэшировании используется механизм отложенной записи. Если выполняется операция записи изменения какого-то блока, то это не приводит к тому, что он немедленно записывается на внешнюю память. Из-за этого серьезной проблемой является старение информации в дисковых блоках, копии которых находятся в буферном кэше. Если не вовремя выполнить синхронизацию буфера кэша и диска, то это может привести к серьезным последствиям в случае отказов оборудования или программного обеспечения - мы просто потеряем то, что писалось на диск. Поэтому стратегия и порядок этого выталкивания из кэша на диск должна быть тщательно продумана. Так, например, блоки, которые существенны для согласованности метаданных файловой системы (это блоки индексных узлов, блоки косвенной адресации, блоки каталогов), должны быть переписаны на диск сразу, независимо от того, в какой части LRU-цепочки они находятся. Требуется тщательно выбирать порядок этой переписи. Для этого в Unix поддерживается системный вызов SYNC, который заставляет систему немедленно вытолкнуть во внешнюю память все модифицированные блоки. Для того, чтобы синхронизировать содержимое кэша и диска периодически запускается соответствующий дежурный фоновый

процесс-демон. Кроме того, при работе с отдельными файлами можно организовать синхронный режим работы. Потребность в нем указывается при открытии файла, тогда все изменения в файле немедленно сохраняются на магнитном диске, на внешней памяти.

- Наконец, проблема конкуренции процессов за доступ к блокам кэша решается путем ведения списков блоков, которые находятся в разных состояниях, и отметкой о состоянии блока в его дескрипторе. Например, блок кэша может быть заблокирован для участия в операции ввода-вывода, а также может иметься список процессов, которые ожидают освобождения этого блока.

Кэширование в основной памяти является не единственным способом увеличения производительности файловой системы. Другим важным методом является сокращение количества движений головок чтения записей диска за счет разумной стратегии размещения данных. Например, массив индексных узлов, к которым часто происходит обращение, в операционной системе Unix стараются разместить на средних дорожках. Также имеет смысл размещать индексные узлы неподалеку от блоков данных, на которые они ссылаются и т.д. Кроме того, рекомендуется периодически осуществлять дефрагментацию диска (то есть сборку мусора), поскольку в популярных методах выделения дисковых блоков принцип локальности не работает, и последовательная обработка файла требует обращения к разным участкам диска. На счет этого стоило бы посмотреть статьи Маршалла МакКузика (Marshall Kirk McKusick) по поводу Fast File System. У него есть достаточно интересные рассуждения относительно того, как размещать данные на магнитных дисках.

Реализация некоторых операций над файлами

В прошлой лекции мы рассматривали основные операции над файлами на пользовательском уровне, рассмотрим теперь порядок работы некоторых системных вызовов для работы с файловой системой.

Системные вызовы, работающие с символическим именем файла

Системные вызовы, которые связывают путевое имя/pathname с дескриптором файла. Это функции системного вызова создания и открытия файла. Например, в операционной системе Unix это системные вызовы **fd = creat (pathname, modes) и fd = open (pathname, flags, modes).** Другие операции над файлами, такие как чтение, запись, позиционирование чтения-записи, воспроизведение дескриптора файла, установка параметров ввода-вывода, определение статуса файла и закрытие файла - для всех них в качестве прямого параметра используется значение полученного дескриптора файла.

Рассмотрим, как работает системный вызова open на уровне программирования:

- прежде всего, логическая файловая подсистема в архиве файлов производит поиск файла по его имени;

- проверяются права пользователей или процесса, который работает от его имени, на открытие файла в нужном режиме;
- для открываемого файла выделяется элемент в таблице файлов (запись элемента таблицы файлов содержит указатель на индексный узел открытого файла);
- после этого ядро выделяет элемент в личной (закрытой) таблице в адресном пространстве процесса, которая локальна для этого процесса - это таблица пользовательских дескрипторов открытых файлов, там запоминается указатель на общесистемную запись.

В роли указателя параметров других системных вызовов, с которыми работают пользователи, выступает дескриптор файла, который возвращается пользователю, то есть это индекс пользовательской таблицы файлов его процесса. Как показано на рисунке 12.9., запись в таблице пользовательских файлов указывает на запись в глобальной таблице файлов. При этом первые три пользовательских дескриптора (0, 1 и 2) в каждой таблице пользовательских файлов именуются дескрипторами файлов стандартного ввода, стандартного вывода и стандартного файла ошибок.

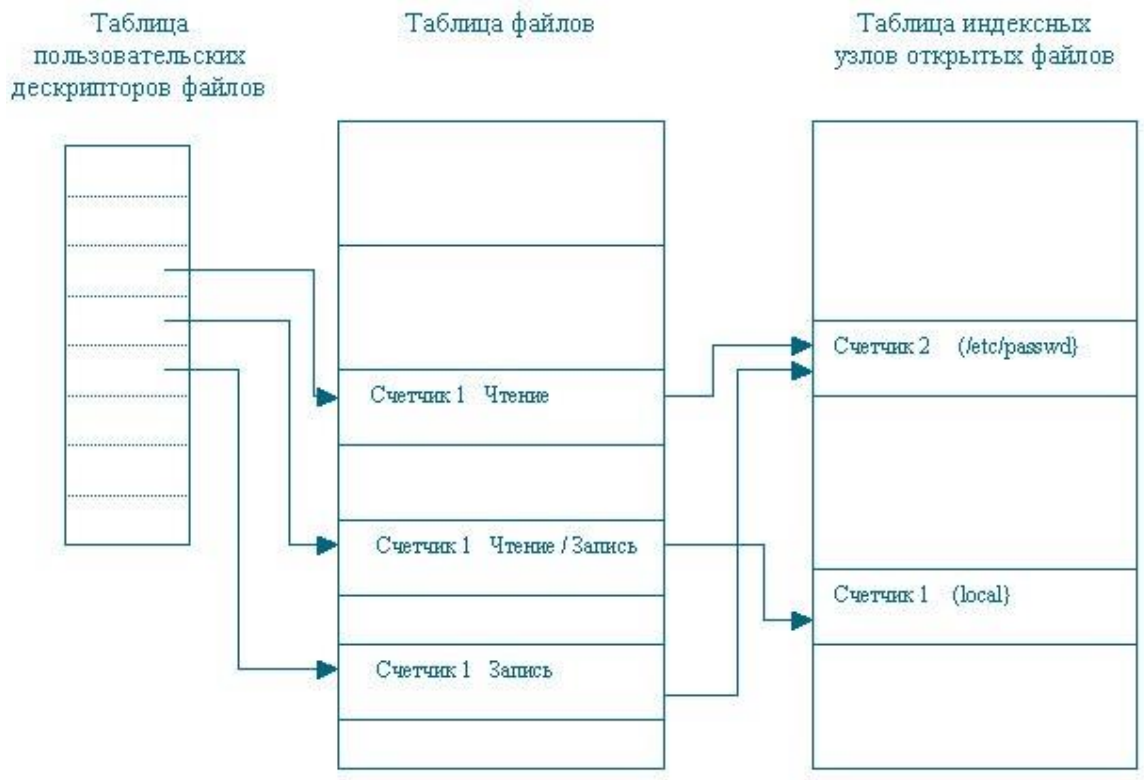

Рис. 12.9. Структуры данных после открытия файлов

Связывание файла. Системный вызов link связывает файл с новым именем в структуре каталогов файловой системы, создавая для существующего индексного узла новую запись в каталоге. Системный вызов функции link имеет следующий вид: link и два параметра - source file name (имя исходного файла), target file name (имя исходного файла), **source file name** - имя существующего файла, а **target file name** - это некоторое новое, дополнительное имя, которое присваивается файлу после выполнения функции link.

Удаление файла. В операционной системе Unix системная функция unlink удаляет из каталога точку входа для файла. То есть синтаксис вызова функции unlink - это unlink (pathname). Если удаляемое имя является последней связью файла с каким-либо каталогом, то ядро в итоге освобождает все блоки файла, то есть файл уничтожается. Однако, если у файла было несколько связей, то он остается доступным под другими именами. Если дисковые блоки таки освобождаются, то для этого ядро в цикле просматривает таблицу содержимого индексного узла, освобождая все блоки прямой адресации немедленно. Что же касается блоков косвенной адресации, то ядро освобождает все блоки, которые возникают на различных уровнях косвенности, рекурсивным образом. Причем в первую очередь освобождаются блоки с меньшим уровнем, то есть те, которые ближе и имеют меньший уровень косвенности.

Системные вызовы, работающие с файловым дескриптором

Открытые файлы могут использоваться для чтения и записи последовательностей байтов, для этого поддерживаются системные вызовы read и write. В качестве их первого, основного параметра выступает дескриптор файла, который в терминологии Microsoft называется **handle**, который получен при ранее выполненных системных вызовах open или creat.

Функции ввода-вывода из файла. Системный вызов read выполняет чтение обычного файла. Системный вызов имеет вид: **number = read (fd, buffer, count)**.

- первый параметр **fd** - дескриптор файла, который был получен при выполнении системного вызова open или create;
- **buffer** - это адрес структуры данных в пользовательском процессе, где будут размещаться считанные данные в случае успешного завершения выполнения функции read;
- **count** - это число количество байтов, которые пользователю нужно прочитать;
- в ответ сообщается значение системного вызова как функция - это **number**, которая означает число фактически прочитанных байтов.

Синтаксис вызова системной функции write (писать): **number = write (fd, buffer, count)**. Здесь переменные fd, buffer, count и number имеют практически тот же смысл, что и для вызова системного вызова функции read:

- **buffer** - это указатель на заполненный буфер пользовательского адресного пространства, где находится последовательность байтов, которые необходимо записать на диск;
- **count** - это размер этого буфера;
- **number** - это сколько реально получилось записать байтов на диск.

Алгоритм записи в обычный файл похож на алгоритм чтения из обычного файла. Однако, если в файле отсутствует блок, соответствующий смещению в байтах до места, куда должна производиться запись, то ядро выделяет блок и присваивает ему номер в соответствии с точным указанием места в таблице содержимого индексного узла.

Обычное использование системных функций read и write обеспечивает последовательный доступ к файлу, однако процессы могут использовать системный вызов lseek для указания позиции в файле, откуда должен производиться ввод или куда должен производиться вывод, что позволяет производить произвольный доступ к файлу. Синтаксис вызова системной функции: **position = lseek (fd, offset, reference).**

- **fd** - это дескриптор открытого файла, идентифицирующий файл;
- **offset** - смещение в байтах;
- **reference** - указывает, что такое offset - это смещение от начала файла или смещение от текущей позиции ввода-вывода, или смещение от конца файла.

Возвращаемое значение position является смещением в байтах до места, где будет начинаться следующая операция чтения или записи.

Современные архитектуры файловых систем

Современные операционные системы позволяют пользователям работать сразу с несколькими файловыми системами. Например, Linux может работать с Ext2fs, FAT и т.д. В этом случае файловая система в традиционном понимании становится частью более общей многоуровневой структуры. Пользовательское приложение напрямую взаимодействует через системные вызовы с диспетчером файловых систем. Диспетчер отправляет на некоторую конкретную файловую систему, которая выходит на одну и ту же систему ввода-вывода. Диспетчер файловых систем связывает запросы прикладной программы с конкретной файловой системой, а каждая файловая система (иногда их называют драйвером файловой системы) на этапе инициализации регистрируется у диспетчера, сообщая ему точки входа для последующих обращений к данной файловой системе. Та же идея поддержки нескольких файловых систем в рамках одной операционной системы может быть реализована по-другому, например, исходя из концепции виртуальной файловой системы.

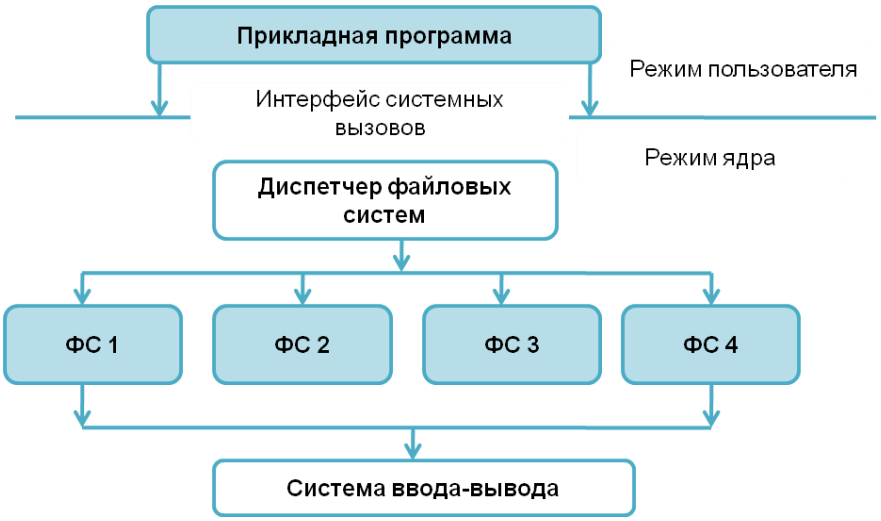

Рис. 12.10. Архитектура современной файловой системы

Виртуальная файловая система - vfs представляет собой независимый от реализации уровень и опирается на реальные файловые системы (s5fs, ufs, FAT, NFS, FFS). При этом возникают структуры данных виртуальной файловой системы типа виртуальных индексных узлов vnode, которые обобщают индексные узлы конкретных файловых систем.

Заключение:

- Реализация файловой системы связана с такими вопросами, как поддержка понятия логического блока диска, связывания имени файла и блоков его данных, проблемами совместного использования файлов и проблемами управления дисковым пространством.
- Наиболее распространенными способами выделения дискового пространства являются: непрерывное выделение, организация связного списка и использование индексных узлов.
- Файловая система часто реализуется в виде слоеной модульной структуры. Нижние слои имеют дело непосредственно с аппаратурой, а верхние - с символическими именами и логическими свойствами файлов.
- Каталоги могут быть организованы разными способами и могут хранить атрибуты файла и адреса блоков файлов, а иногда для этого предназначается специальная структура - индексные узлы.
- Важнейшими аспектами проектирования файловой системы являются надежность и её производительность.