Нижегородский государственный университет им. Н.И. Лобачевского

Факультет вычислительной математики и кибернетики

Учебно-исследовательская лаборатория
«Математические и программные технологии для современных компьютерных систем (Информационные технологии)»

Линев А.В., Свистунов А.Н.

Лабораторный практикум по курсу "Операционные системы"

Нижний Новгород
2004

Оглавление

Содержание

Введение

Современный уровень развития информационных технологий характеризуется наличием большого числа операционных систем, решающих самые разнообразные задачи и обладающих различными специфическими свойствами. Тем не менее, многолетний опыт использования различных ОС позволяет выделить типичные составляющие операционной системы и определить ряд моделей, лежащих в основе их функционирования. Изучение данных моделей и типовых архитектур современных ОС является совершенно необходимой составляющей подготовки высококвалифицированных специалистов в области системного программирования и администрирования, и позволяет прикладному программисту эффективно использовать возможности ОС.

Необходимо отметить постоянно увеличивающуюся сложность современного программного обеспечения и связанную с этим тенденцию использования алгоритмов, изначально разработанных для применения в системном программном обеспечении, при проектировании прикладного ПО. А также обратный процесс - привитие некоторых возможностей прикладного ПО операционным системам.

Данная работа описывает лабораторный практикума для курса "Принципы построения современных ОС", основное предназначение практикума - получение слушателями практических навыков реализации некоторых моделей и алгоритмов, рассматриваемых в курсе лекций, и осуществление ими оценки собственных разработок с учетом положений, изучаемых в теоретической части курса.

Требования к содержанию лекций

Курс по операционным системам читается во множестве учебных заведений, и информация о содержании и методической постановке курсов зачастую доступна через Internet. Однако в большом числе случаев данный курс читается без лабораторного практикума. Еще существует небольшая часть курсов, сопровождаемых практическими занятиями пользовательского уровня (например, обучение профессиональным навыкам работы в конкретной операционной системе). В остальных случаях лабораторный практикум непосредственно связан с разделами лекционного курса.

Мы будем считать, что курс лекций содержит, по крайней мере, следующие разделы.

1. Недетализованные модели объектов аппаратного уровня.
2. Управление ресурсом "память".
3. Управление ресурсом "центральный процессор".
4. Процессы и потоки, операционная среда.
5. Синхронизация выполнения процессов/потоков.
6. Обмен данными между процессами/потоками.
7. Архитектура ввода/вывода.
8. Долгосрочное хранение данных.

Лабораторный практикум иллюстрирует разделы 2,3,5,6,8. Раздел 4 не требует поддержки в виде практических занятий; практикум по вопросам, рассматриваемым в разделах 1 и 7, является привилегией курса по архитектуре вычислительных систем.

В качестве предполагаемой теоретической части использовался курс «Введение в операционные системы» (В.Е. Карпов, К.А. Коньков, В.П. Иванников, МФТИ), содержание которого доступно в Internet (http://cs.mipt.ru/docs/courses/osstud/os.html). В частности, материалы данного курса были использованы при создании теоретического раздела.

Организация лабораторного практикума

Большинство заданий практикума курса по операционным системам обычно рассчитаны на создание слушателями некоторого программного обеспечения, логика работы которого базируется на материалах, предлагаемых в курсе, в частности:

- программное обеспечение, реализующее один из алгоритмов, лежащих в основе функционирования ОС (например, алгоритм диспетчеризации);

- программное обеспечение, использующее реализацию некоторых разделов теории операционных систем в конкретной ОС (например, реализация синхронизации выполнения нескольких потоков одной программы в операционной среде Win32 ОС Windows NT/2000/XP).

Отметим, что предлагаемые к использованию аппаратно-программные платформы и инструментальные средства в различных существующих курсах существенно различаются.

Авторами при разработке принимались во внимание следующие дополнительные факторы:

1.Задания по возможности должны быть рассчитаны на выполнение как в среде Windows, так и UNIX.

2. Отдельные работы должны иметь два типа выполнения, различающиеся по сложности.

3. Для каждой работы необходимо создать несколько вариантов исполнения.

Постановка каждой задачи лабораторного практикума очень коротка, однако для понимания задачи необходимо знать материал соответствующего раздела курса, а для выполнения реализации – обладать знаниями о среде выполнения работы.

Необходимый теоретический материал кратко представлен в первом разделе, содержащем адаптированное изложение материала печатных и электронных источников [1-3,5]. Второй раздел содержит постановки задач и дополнительную информацию, которая может помочь при выполнении работы. Необходимо отметить, что часть материалов изложена очень кратко ввиду объемности рассматриваемых вопросов. При необходимости более глубокого изучения какой-либо темы, обращайтесь к литературе.

Программа лабораторного практикума

Практикум включает пять лабораторных работ.

Лабораторная работа 1. Краткосрочное планирование задач

Цель работы – реализация одного из алгоритмов диспетчеризации.

Данная работа предполагает два варианта исполнения:

а) создание программного блока, реализующего диспетчеризацию, для симулятора;

б) модификация исходных кодов ядра UNIX, относящихся к подсистеме планировщика с последующей их компиляцией и установкой полученного ядра. (Здесь и далее предполагается выполнение работ с использованием Linux RedHat 7.3, версия ядра - 2.4.18)

Постановка задачи для конкретного слушателя включает выбор варианта исполнения и реализуемого алгоритма диспетчеризации.

Лабораторная работа 2. Замещение областей памяти

Цель работы – реализация одного из алгоритмов замещения страниц памяти.

Данная работа предполагает два варианта исполнения:

а) создание программного блока, реализующего замещение страниц, для симулятора;

б) модификация исходных кодов ядра UNIX, относящихся к подсистеме управления памятью с последующей их компиляцией и установкой полученного ядра.

Постановка задачи для конкретного слушателя включает выбор варианта исполнения и реализуемого алгоритма замещения страниц.

Лабораторная работа 3. Синхронизация процессов/потоков

Цель работы - практическое освоение механизмов синхронизации процессов и их посредством механизмов, предоставляемых ОС.

Слушателю предлагается задача и операционная среда (операционных систем Windows или UNIX), в которой требуется выполнить работу. Постановка задачи сходна с одним из классических примеров, для которых имеются решения в виде шаблонов.

Задача слушателя – создать работающую программу (программы).

Лабораторная работа 4. Передача данных между процессами/потоками

Цель работы - практическое освоение механизмов передачи данных, предоставляемых ОС.

Как и в предыдущей работе, слушателю предлагается задача и операционная среда. Задача слушателя – создать работающую программу (программы).

Лабораторная работа 5. Файловые системы

Цель работы – реализация доступа к линейному пространству данных как к иерархической файловой системе.

Данная работа предполагает два варианта исполнения:

а) создание программного блока, обеспечивающего работу с файловой системой, хранящейся в файле;

б) создание модулей для ОС UNIX, обеспечивающих подключение к иерархии файловой системы раздела с файловой системой, хранящегося в файле.

Постановка задачи для конкретного студента включает выбор варианта исполнения и параметры файловой системы.

Литература

В настоящее время в теории операционных систем выделилась некоторая классическая часть являющаяся по большей части устоявшейся (базовые понятия и определения, управление

памятью, диспетчеризация, теоретические основы взаимодействия процессов и др.). Лабораторный практикум полностью опирается на классическую теорию, которая описана в целом ряде книг зарубежных и российских авторов, причем большинство из них либо написаны по материалам читаемого автором курса, либо предлагают набор лекций для составления из них курса по выбору преподавателя, либо просто являются учебником для высших учебных заведений.

Для более подробного изучения вопросов теоретического раздела мы рекомендуем следующую литературу:

- Э. Таненбаум "Современные операционные системы" – в книге подробно излагается классическая теория операционных систем и описываются современные тенденции их развития;

- А.В. Гордеев, А.Ю. Молчанов "Системное программное обеспечение" - книга российских авторов, в которой достаточно компактно освещены некоторые вопросы. Несмотря на то, что данный источник сильно уступает, например, книге Таненбаума, его можно рекомендовать благодаря его относительно небольшому размеру и нестрогому и понятному изложению;

- Silberschatz and Galvin "Operating System Concepts (6th Edition)" - рекомендуется в качестве книги для чтения в большом числе иностранных ВУЗов (авторам данная книга недоступна);

- Карпов и др. «Операционные системы» - содержание лекционного курса МФТИ.

Существуют множество книг, представляющих собой описания отдельных аспектов функционирования операционных систем (например, В.Г. Олифер, Н.А. Олифер "Сетевые операционные системы"). В настоящее время операционные системы представляют собой сложнейшие комплексы, изучить все подсистемы которых не представляется возможным. Книги подобного типа помогают понять структуру и алгоритмы функционирования конкретных подсистем ОС. Они могут быть рекомендованы для более подробного изучения некоторых разделов курса.

Для успешного выполнения лабораторного практикума также необходима информация об операционных средах, в которых он выполняется. Для этого необходимо использовать техническую документацию, которая в настоящее время имеется в избытке как в печатном, так и в электронном виде (например, The Linux Kernel by David Rusling http://www.linuxdoc.org/LDP/tlk/). Для конкретных лабораторных работ указаны списки рекомендуемой технической литературы.

Мы надеемся, что выполнение предложенных работ и изучение представленных материалов позволит читателям полнее рассмотреть и понять методы и алгоритмы, лежащие в основе функционирования операционных систем, а преподавателям объективно оценить знания слушателей курса.

Обзор теоретических положений, необходимых для выполнения практикума

Основные понятия и определения

В начале изложения теоретического материала дадим несколько первоначальных определений часто используемых терминов. Предлагаемые определения не являются точными или исчерпывающими, поскольку с нашей точки зрения, для ниже перечисленных понятий дать краткое и исчерпывающее определение не представляется возможным.

Операционная система – комплекс управляющих и обрабатывающих программ, выполняющий задачи управления ресурсами системы и предоставляющий прикладным программам операционную среду для их исполнения. Две основные функции операционной системы – расширение возможностей ЭВМ и управление ее ресурсами.

Операционная среда – среда исполнения прикладных программ. В частности, операционная среда определяет для прикладных программ множество команд процессора, которые они могут использовать, модель адресации и логическую структуры адресного пространства процесса, множество системных вызовов, доступных процессу и т.д. Отметим, что операционная система может осуществлять поддержу нескольких различных операционных сред.

Ресурсы – обычно, повторно используемые, относительно стабильные и часто недостающие объекты, которые запрашиваются, используются и освобождаются процессами в период их активности. Ресурс может быть разделяемым, в этом случае несколько процессов могут его использовать одновременно (в один и тот же момент времени) или параллельно (в течение некоторого интервала времени процессы используют ресурс попеременно); ресурс может быть неделимым.

Существуют аппаратные ресурсы, такие как процессорное время, оперативная память и дисковое пространство; программные ресурсы, например, библиотеки функций; информационные ресурсы – содержимое файлов и баз данных; ресурсы операционной среды – структуры, используемые при выполнении системных вызовов, например, структура сообщения; другие типы ресурсов. С точки зрения системы управления ресурсами, данное понятие нивелировалось до уровня абстрактной структуры с набором атрибутов, характеризующим методы доступа к этой структуре и ее физическое представление в системе.

Одним из основных понятий, связанных с операционными системами, является понятие процесса.

Процесс - абстракция, представляющая программу во время ее выполнения. Процесс является потребителем различных ресурсов операционной системы, например:

- адресное пространство процесса содержит его программный код, данные и стек (или стеки);
- файлы используются процессом для чтения входных данных и записи выходных;
- устройства ввода-вывода используются в соответствии с их назначением.

Необходимо отметить, что множество доступных процессу ресурсов и порядок их использования определяются архитектурой операционной системы. В частности, адресное пространство процесса создается в момент запуска программы на основании информации, получаемой операционной системой из содержимого программы и параметров задания/запуска (если они есть). В зависимости от архитектуры вычислительного устройства и операционной системы, предоставляемое процессу адресное пространство будет обладать различными параметрами (количество адресных пространств, их размер, начальные и конечные адреса, способы адресации команд и данных и т.д.).

Поток исполнения или просто **поток** – абстракция, представляющая выполнение программы, развертывающееся во времени. Каждый процесс имеет по крайней мере один поток. Потоки процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждый поток имеет свой собственный программный счетчик, свое содержимое регистров и свой собственный стек. Процесс представляет собой совокупность взаимодействующих потоков и выделенных ему ресурсов.

Распределение ресурса «центральный процессор»

Процесс и поток – типы ресурсов операционной системы

С точки зрения операционной системы, процессы и потоки представляет собой типы ресурсов, которыми она должна управлять. Для этого ей необходимо хранить информацию о совокупности существующих процессов и потоков в системе и описание каждого из них.

Что представляют собой полное описание процесса и полное описание потока? Это множество информации, достаточное для восстановления процесса или потока в случае, если он был прерван или уничтожен во время выполнения команд программы (за исключением случаев, когда выполнение процесса или потока было прервано в процессе обработки неперезапускаемого системного вызова ядром ОС). Для процесса это:

1. Множество внешней по отношению к процессу информации, используемое операционной системой для управления ресурсом типа «процесс». Состав данной информации зависит от операционной системы.

2. Структура и содержимое пользовательской части виртуального адресного пространства процесса (части адресного пространства, доступной потокам процесса во время выполнения команд программы).

3. Множество ресурсов, принадлежащих процессу или используемых процессом и их состояния. Для разных типов ресурсов состояния описываются различными способами.

Одним из типов ресурсов, принадлежащих процессу, являются потоки. Каждому процессу принадлежит по крайней мере один поток. Описание потока представляет собой:

1. Множество информации, используемое операционной системой для управления ресурсом типа «поток».

2. Аппаратный контекст исполнения потока, полный состав которого зависит от аппаратной платформы. Например, в аппаратный контекст входят:

- состояние процессора с точки зрения предоставляемых потоку прав его использования в конкретной операционной системе, которое обычно представляется множеством доступным потоку регистров процессора (общего назначения, арифметических, адресных, сегментных, флагов, управляющих и т.д.) и их текущими значениями;

- состояние других устройств в случае, если управление ими осуществляется непосредственно на уровне команд программы, а не через интерфейс доступа к устройствам через выполнение системных вызовов операционной системы. Такая ситуация возможна в случае, если операционная система предоставляет пользовательским программам непосредственный доступ к аппаратным ресурсам. В настоящее время пользовательским программам обычно предоставляется доступ к некоторому объекту операционной системы, представляющему физическое устройство, и множеству системных вызовов для работы с этим объектом, а непосредственное взаимодействия с этим устройством осуществляет операционная система. При организации работе с таким объектом ОС, его владение обычно закрепляется за процессом, а не за потоком.

3. Множество ресурсов, используемых потоком или принадлежащих потоку, и их состояния. В большинстве случаев пользователем (владельцем) ресурсов является процесс, но существую исключения, например, мьютексы.

В случае, если выполнение потока какого-либо процесса было прервано в ходе обработки системного вызова, то во многих случаях для продолжения выполнения необходимо также полностью восстановить состояние подсистемы ядра, обслуживавшей запрос (что может потребовать восстановления состояния всего ядра в целом).

Таким образом, мы видим, что возможность восстановление процесса или потока после того как он был прерван или уничтожен, зависит от возможности восстановления выше перечисленных параметров. Наибольшую сложность в данном случае представляет собой восстановления состояния аппаратных устройств, поскольку алгоритмы восстановления состояния аппаратуры могут быть чрезвычайно сложны или принципиально не реализуемы (попробуйте составить алгоритм, в результате выполнения которого жесткий диск примет желаемые параметры S.M.A.R.T.)

В описании процесса и потока можно выделить некоторую системную составляющую, которую операционная система хранит для пользовательских процессов/потоков вне зависимости от того, какой процесс или поток является активным (исполняется) в настоящий момент. А также составляющую, состояние которой может быть утеряно при потере процессом/потоком активного состояния и, соответственно, должно впоследствии восстанавливаться. Эту составляющую мы будем называть **контекстом процесса** или **контекстом потока**. Например, в контекст процесса может входить описание его виртуального адресного пространства, а в контекст потока всегда входят значения доступных ему регистров процессора.

Ядро операционной системы также имеет свое описание состояния, аналогичное описанию процесса, свой контекст. Однако ядру никто не предоставляет виртуализованное представление ресурсов, оно работает с ними напрямую. Поэтому описание состояния ядра существенно зависит от аппаратной среды. С другой стороны, ядро может не иметь постоянно исполняемых потоков и заниматься только обслуживанием системных вызовов (в этом случае используется поток, осуществивший вызов) и обработкой событий по мере их возникновения (в этом случае используется поток, исполнявшийся на процессоре, на котором произошло событие). В зависимости от организации операционной системы, при таком использовании может осуществляться или не осуществляться смена контекста процесса и потока.

Классификация ОС по признаку поддержки процессов и потоков

Существует классификация операционных систем, основанная на их возможностях по работе с процессами и потоками. В соответствии с ней ОС подразделяются следующим образом.

1. Однозадачные ОС

Однозадачные операционные системы рассчитаны на поддержку только одного процесса в каждый момент времени. Этот единственный процесс может иметь только один поток. Программы могут запускаться только последовательно – до завершения выполнения процесса нельзя создать еще один процесс.

Рис. 1 Однозадачная ОС

Однозадачные ОС включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем. В ходе выполнения процесса задача однозадачной ОС практически сводится к поддержке системных вызовов.

2. Многозадачные ОС без поддержки многопоточности

Такие операционные системы поддерживают одновременное существование нескольких процессов, каждый из которых может иметь только один поток.

Рис. 2 Несколько процессов, у каждого – один поток

Многозадачные ОС должны осуществлять управление разделением совместно используемых ресурсов, таких как процессор, оперативная память, файлы и внешние устройства. В них всегда осуществляется переход исполнения между потоками разных процессов, для чего требуется переключение контекста процесса и контекста потока. Дополнительно отметим, что смену исполняемого потока выполняет ядро, то есть при изменении активного потока предварительно происходит переключение в контекст ядра (меняется при этом контекст процесса или контекст потока – зависит от ОС).

3. Многозадачные ОС с поддержкой многопоточности

В таких ОС одному процессу могут принадлежать несколько потоков исполнения команд. Все потоки одного процесса разделяют его ресурсы, например, адресное пространство или открытые файлы, однако характеризуются собственным аппаратным контекстом.

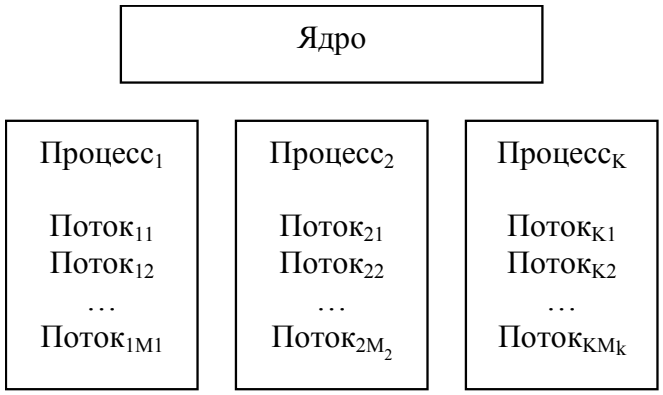


Рис. 3 Несколько процессов, у каждого - несколько потоков

В такой системе может происходить переход исполнения от одного потока процесса к другому потоку того же процесса. В этом случае не требуется переключения контекста процесса, соответственно такое переключение производится быстрее, чем в случае переключения между процессами.

С точки зрения управления ресурсами, владельцем или пользователем каждого типа ресурсов может являться либо процесс, либо поток.

Реализация многопоточности внутри программы пользователя

Даже в операционной системе, не поддерживающей многопоточность, можно создавать многопоточные приложения, используя специальные библиотеки и технику программирования. Размещение потоков целиком организуется внутри процесса. Для этого используется библиотека поддержки исполнения потоков. Библиотека осуществляет хранение данных о множестве существующих в процессе потоках, их аппаратных контекстов исполнения (обычно это значения всех регистров процессора, доступных на пользовательском уровне исполнения), состояниях, и предоставляет набор процедур управления потоками.

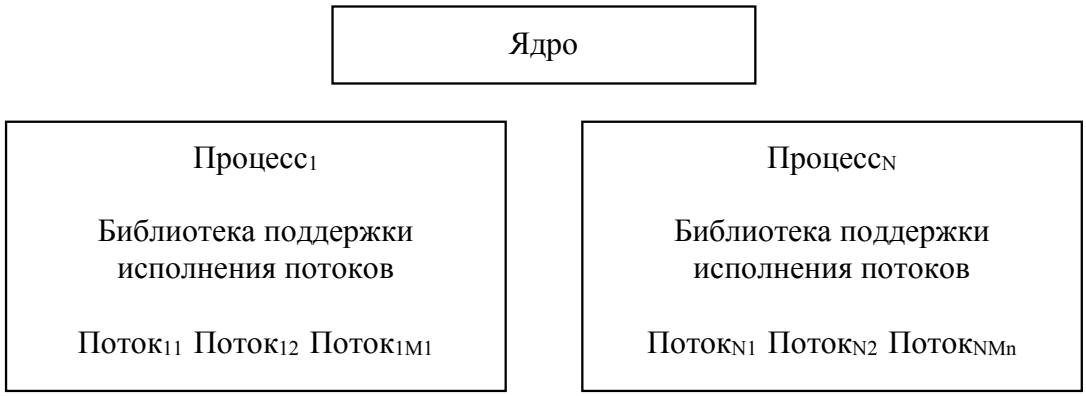


Рис. 4 Несколько процессов, у каждого - несколько пользовательских потоков

Когда поток делает нечто, что может привести к локальной блокировке, он вызывает процедуру библиотека поддержки исполнения потоков. Процедура проверяет необходимость блокирования потока. В случае положительного результата она сохраняет аппаратный контекст потока, ищет поток процесса, готовый к запуску, и загружает его аппаратный контекст. Как только счетчик команд переключен, работа потока возобновляется автоматически.

Преимущество данного подхода состоит в том, что переключение контекста внутри процесса не требует переключения контекстов для перехода в режим ядра, и соответственно, выполняется намного быстрее. К тому же мы можем использовать наиболее подходящий для решения конкретной задачи алгоритм распределения времени центрального процессора между потоками процесса.

Однако, имеются серьезные недостатки: во-первых, при распределении ядром времени центрального процессора оно не будет учитывать многопоточность процессов; во-вторых, в случае выполнения одним из потоков системного вызова, приводящему к переходу в состояние ожидания, процессу не будет выделяться процессорного время до выполнения условия ожидания, то есть выполнение всех потоков процесса будет приостановлено до окончания выполнения системного вызова.

Возможно использование комбинации поддержки многопоточности на уровне ядра и библиотеки поддержки исполнения потоков.

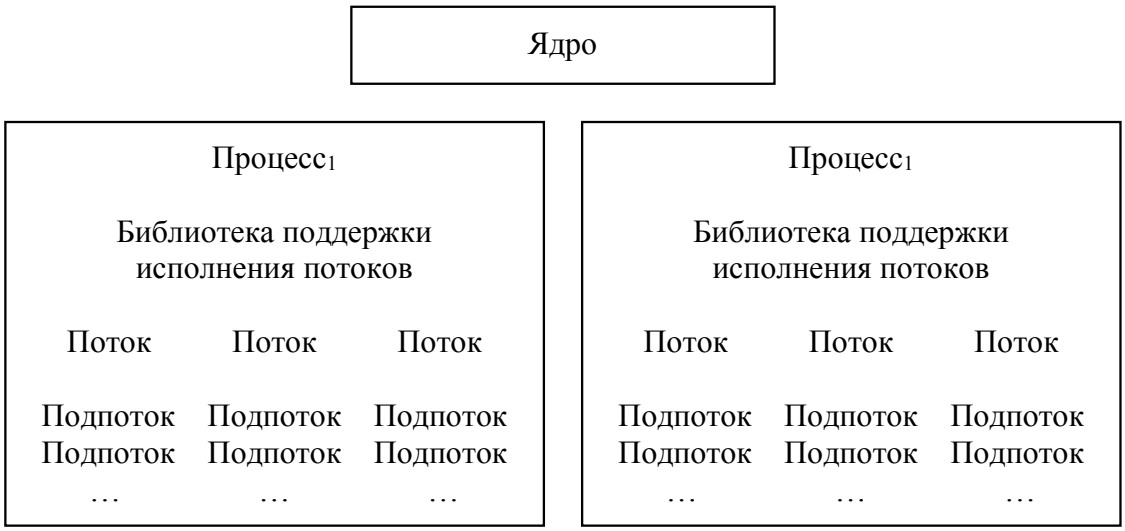


Рис. 5 Несколько процессов, у каждого процесса - несколько потоков, у каждого потока – несколько пользовательских подпотоков

В такой модели ядро знает только о потоках своего уровня и управляет ими. Некоторые из этих потоков могут содержать по несколько мультиплексированных потоков пользовательского уровня.

Состояния потока

В однозадачной среде единственный поток единственного процесса во время от его создания до завершения может находиться в одном из двух состояний – выполнение и ожидание.

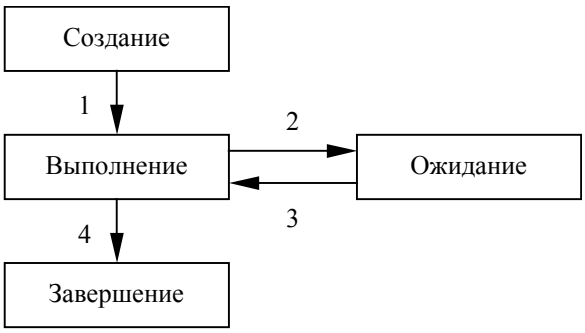


Рис. 6 Диаграмма состояний потока в однозадачной ОС

Поток переходит из состояния выполнения в состояние ожидания (2) посредством выполнения системного вызова, подразумевающего ожидание наступления какого-либо события, например, нажатия клавиши или истечения 10 с. При наступлении этого события происходит возврат из системного вызова и возвращение потока в состояние выполнения (3).

Особенностью однозадачных систем является предоставление единственному активному процессу всех ресурсов системы.

В многозадачных системах существует конкуренция процессов и потоков за ресурсы. Поскольку число существующих в системе центральных процессоров ограничено, то в одновременно может выполняться лишь ограниченное число потоков. Остальные в это время должны ожидать своей очереди. Конкуренция между потоками за ресурс «время центрального процессора» в силу специфики продолжительности традиционно рассматривается отдельно. Диаграмма состояний выглядит следующим образом.

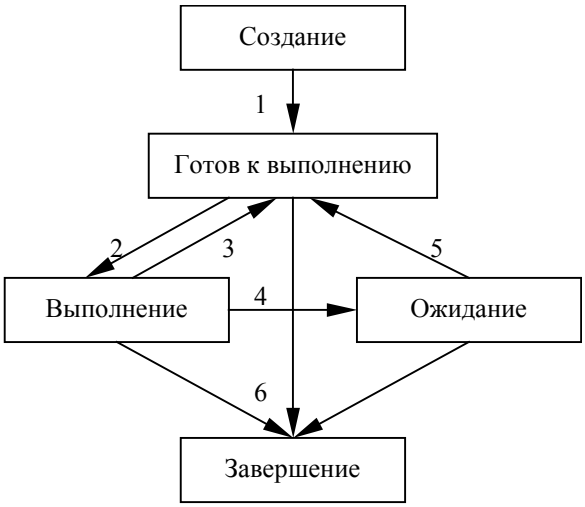


Рис. 7 Диаграмма состояний потока в многозадачной ОС

Выполнение – состояние работающего потока, то есть потока, обладающего всеми необходимыми ресурсами, в том числе возможностью использования центрального процессора.

Готов к выполнению – поток обладает всеми необходимыми для выполнения ресурсами за исключением ресурса «время центрального процессора».

Ожидание (**сон**) – выполнение потока заблокировано до наступления некоторого внешнего по отношению к нему события (например, поступления входных данных или освобождения ресурса).

Перевод потока из состояния «выполнение» в состояние «готов к выполнению» и обратно (2,3) осуществляется ядром операционной системы. Переход из состояния выполнения в состояние ожидания (4) может производиться в следующих случаях:

- поток обнаружил, что продолжение работы невозможно;

- поток обнаружил, что для продолжения работы требуется какого-либо события;

- поток затребовал недоступный в данный момент ресурс;

- поток переводится в состояние ожидания ядром операционной системы во время обработки системного вызова;

- поток заблокирован внешним по отношению к нему вызовом.

В любом из этих случаев переход потока в состояние ожидания производится посредством некоторого системного вызова, при этом для потока сохраняется некоторое условие завершение ожидания (приход события, освобождение ресурса и т.д.)

Переход из состояния ожидания в состояние готовности к выполнению (5) производится ядром ОС в момент выполнения условия ожидания.

Переход потока в состояние завершения может произойти из любого состояния, поскольку завершение потока может быть вызвано как внутренними, так и внешними по отношению к потоку причинами.

В конкретных операционных системах состояния потока могут быть более детализированы, могут появиться некоторые новые варианты переходов из состояния в состояние.

Примечание. Обратите внимание, что подсистема ядра, производящая переключение выполняющихся потоков, не требует наличия специального потока. При работе она использует тот поток, который переходит из состояния выполнения в какое-либо другое состояние.

Дескрипторы процессов и потоков

Каким образом информация о процессах и потоках представлена в операционной системе в период их активности?

В адресном пространстве ядра операционной системы хранится информация о множестве процессов, существующих в системе. Для каждого процесса хранится его описатель – **дескриптор процесса** или **блок управления процесса**. Обычно он содержит данные, на основании которых операционная система осуществляет управление ресурсом «процесс». Перечислим типичные поля этой структуры:

1. Идентификатор процесса - уникален, используется для идентификации процесса.

2. Групповые параметры процесса – родительский процесс, дочерние процессы, процессы, принадлежащие одному заданию т.д.

3. Параметры, используемые в ходе определения приоритета процесса при конкуренции за какой-либо ресурс.

4. Состояние процесса – термин «состояние» обычно относится к диаграмме состояний потока, однако можно выделить состояния, влияющие на все потоки процесса, например, состояние «отсутствует в памяти».

5. Статистические данные – время создания процесса, время центрального процессора, использованное всеми потоками процесса, время ЦПУ, использованное дочерними процессами, и т.д.

6. Описание виртуального адресного пространства процесса – тип организации ВАП, структура ВАП (например, число, типы и размещение сегментов в случае использования сегментной адресации с поддержкой типов сегментов), способ отображения ВАП на физическую память (оперативную память и другие хранилища данных).

7. Контекст ввода-вывода – информация, определяющая возможности процесса по взаимодействию с устройствами ввода-вывода. Например, для процесса может быть указан управляющий терминал, с которого он может получать клавиатурный ввод, и на который он может отправлять вывод текстовых данных.

8. Контекст безопасности – информация, обрабатываемая подсистемой безопасности операционной системы при контроле правомочности функционирования потоков процесса;

например, владелец процесса (от чьего имени был запущен процесс), группа – владелец процесса, привилегии потоков процесса на выполнение определенных операций и т.д. Контекст безопасности также используется при проверке действий, выполняемых над процессом (завершение, приостановка и т.д.)

9. Текущие системные параметры выполнения – например, корневой и рабочий каталог.

10. Код завершения процесса.

Описание множества ресурсов, используемых процессом или принадлежащих процессу, и их состояния обычно хранятся вне его дескриптора во вспомогательных структурах (в адресном пространстве ядра операционной системы) в силу неопределенности размера и различия структур для разных типов ресурсов. В данном случае мы считаем, что процесс имеет дело только с ресурсами, предоставляемыми операционной системой пользовательским приложениям, а не с аппаратным обеспечением.

Способ структурной организации множества дескрипторов процессов в различных операционных системах может быть различным (массив, список или другая структура). В дальнейшем мы будем считать, что у нас имеется в адресном пространстве ядра таблица дескрипторов процессов, и для управления процессами используется только она (очевидно, такой подход не уменьшает общности, позволяя отвлечься от подробностей реализации).

Структура, описывающая поток, называется **дескриптором потока**. В ней хранится следующая информация.

1. Идентификатор потока.

2. Идентификатор процесса – владельца потока.

3. Параметры, используемые в процессе определения приоритета потока при конкуренции за какой-либо ресурс.

5. Статистические данные потока.

6. Аппаратный контекст выполнения потока – данное поле заполняется при остановке выполнения потока и используется при возобновлении исполнения.

7. Код завершения потока.

Чтобы не углубляться в подробности реализации, в дальнейшем мы будем ссылаться на структуру, хранящую описание множества потоков, как на таблицу дескрипторов потоков.

Если ядро операционной системы поддерживает многопоточность, то таблица дескрипторов потоков располагается в адресном пространстве ядра. Если используется библиотека поддержки исполнения потоков, то для каждого процесса, пользующегося этой библиотекой, существует своя локальная таблица дескрипторов потоков.

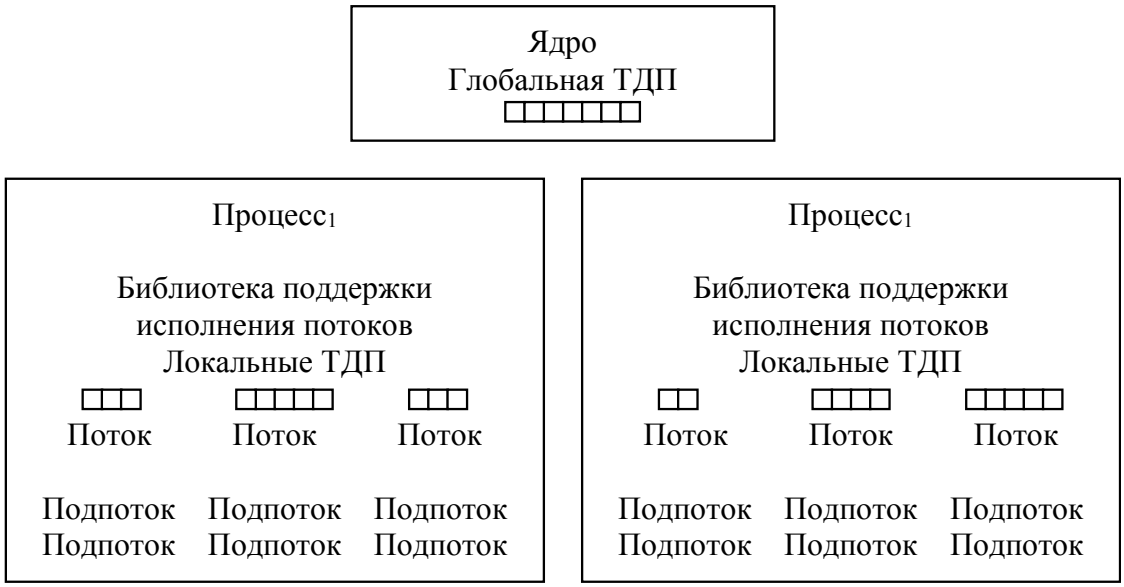


Рис. 8 Для хранения описаний потоков ядра используется таблица дескрипторов потоков в ядре, для хранения описаний пользовательских подпотоков – локальные таблицы дескрипторов потоков

Операции над процессами

1. Создание процесса

В простейших системах можно реализовать ситуацию, когда все процессы, необходимые для работы, присутствуют сразу после загрузки (например, в случае большинства встроенных систем). В универсальных системах должны быть определены правила создания новых процессов. Основными событиями, приводящими к созданию процессов, являются:

- инициализация системы;

- осуществление работающим потоком системного вызова создания нового процесса;

- запрос пользователя на создание нового процесса;

- запуск пакетного задания.

С технической точки зрения все эти действия выполняются одинаково – происходит вызов системной функции создания нового процесса, за исключением инициализации системы, в ходе которого часть процессов создается спонтанно (то есть не посредством выполнения стандартного системного вызова). При этом ядро операционной системы должно выполнить следующие действия:

- создать дескриптор процесса и поместить его в таблицу процессов;

- создать виртуальное адресное пространство процесса и сформировать его структуру (например, скопировав структуру родительского процесса или проанализировав файл запускаемой программы);

- заполнить необходимыми данными виртуальное адресное пространство процесса (разместить в нем код, данные и т.д.);

- выделить процессу ресурсы, которые он может использовать сразу после создания (например, стандартный ввод-вывод) и проинициализировать их состояние;

- проинициализировать значения полей общего назначения дескриптора процесса исходя из параметров вызова или принятых значений по умолчанию;

- оповестить подсистемы, принимающие участие в управлении процессами, о создании нового процесса с целью завершения инициализации его дескриптора;

- создать первичный поток процесса (данный процесс мы рассмотрим чуть позже).

Например, в UNIX существует системный запрос, направленный на создание нового процесса – fork(). Он создает новый процесс, который является почти точной копией родителя. После возвращения из системного вызова оба процесса выполняют инструкции одной и той же программы, имеют одинаковое содержимое адресного пространства и возобновляют выполнение с команды, следующей за системным вызовом (рассматривается случай однопоточного процесса). Однако между родительским и дочерним процессом имеется ряд различий:

- дочернему процессу присваивается уникальный идентификатор процесса PID, отличный от родительского;

- дескриптор процесса в UNIX содержит поле «идентификатор родительского процесса» PPID, соответственно, у дочернего процесса он устанавливается в PID родителя;

- дочерний процесс получает собственную копию u-area - структуру данных, используемую подсистемами ядра для управления процессом и выполнения его системных вызовов; она находится в адресном пространстве ядра и содержит, в частности, информацию об открытых файловых дескрипторах, статистику выполнения процесса и т.д.;

- для дочернего процесса очищаются все ожидающие доставки сигналы;

- временная статистика дочернего процесса обнуляется;

- блокировки памяти и записей, установленные родителем, потомком не наследуются;

- системный вызов fork() в родительском процессе возвращает PID дочернего процесса, а в дочернем – 0.

Для загрузки новой программы, процесс должен выполнить системный вызов семейства exec(). При этом новый процесс не порождается, сегменты кода, данных и стека создаются заново и инициализируются в соответствии с содержимым запускаемой программы, и окружение во многом сохраняется. Например, сохраняются назначения стандартных потоков ввода-вывода и приоритет процесса.

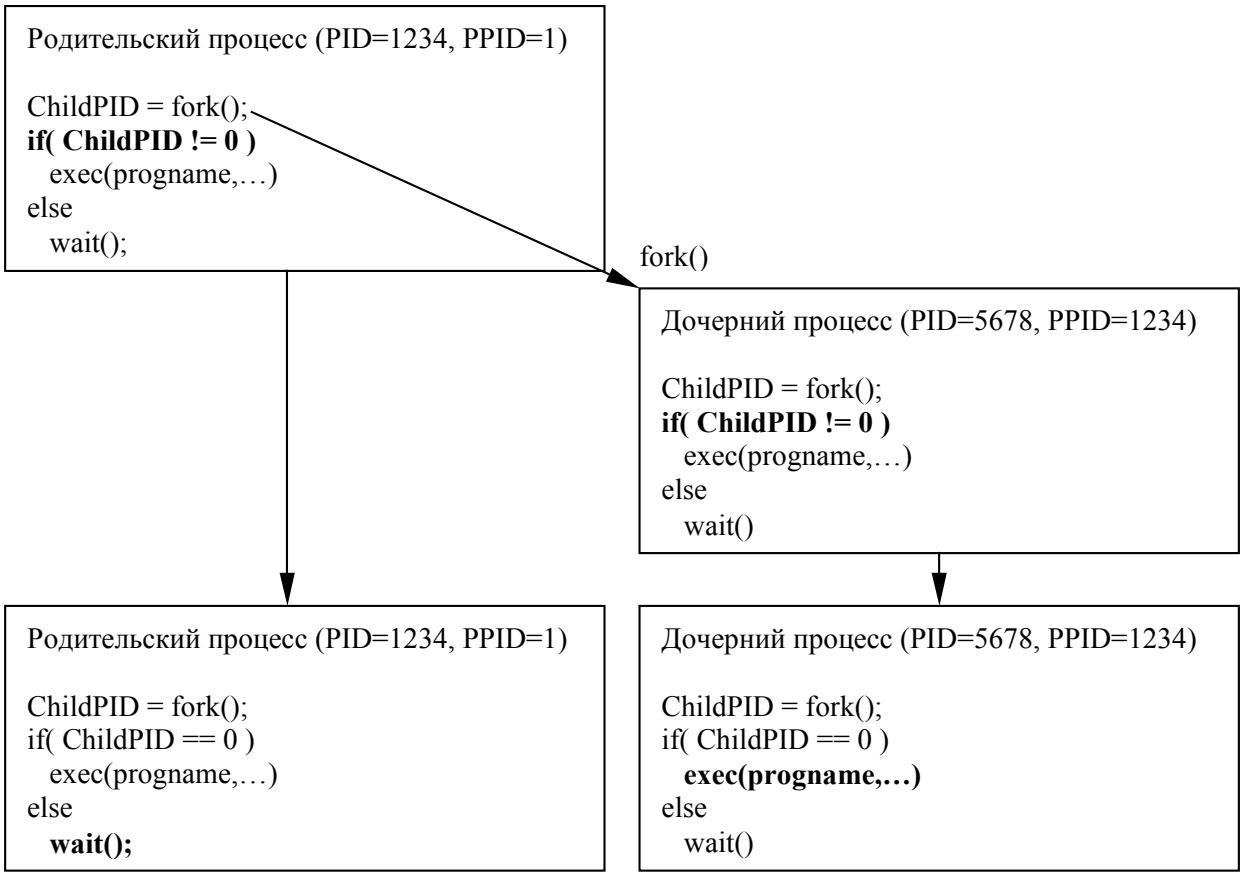


Рис. 9 Создание нового процесса в UNIX

В Windows вызов одной из функции CreateProcess() или CreateProcessEx() интерфейса Win32 управляет и созданием процесса и запуском в нем нужной программы.

2. Завершение процесса

После того как процесс создан, его потоки начинают выполнять свою работу. Однако рано или поздно выполнение процесса будет завершено, чаще всего благодаря одному из следующих событий:

- обычный выход - основной поток процесса совершил обычный выход или один из потоков совершил выход посредством обращения к соответствующему системному вызову;

- один из потоков совершил выход по ошибке (запланированный выход);

- один из потоков допустил неисправимую ошибку (незапланированный выход);

- поток другого процесса выполнил системный вызов завершения данного процесса (незапланированный выход).

В основном процессы завершаются по мере выполнения своей работы. В UNIX для этого может использоваться системный вызов exit() (или _exit()), в Windows – ExitProcess().

Примером выхода по ошибке может служить поведение программы, которой передали неверный параметр командной строки.

Неисправимая ошибка возникает, когда процесс нарушает «правила поведения», предписанные в данной операционной системе пользовательским процессам, например, пытается выполнить операции чтения-записи по недоступным ему адресам памяти.

При наличии у процесса соответствующих привилегий, его потоки могут выполнять системные вызовы, завершающие другие процессы. В UNIX для этого может использоваться системный вызов kill(), в Windows – функция Win32 TerminateProcess().

При завершении процесса операционная система выполняет следующие действия:

- завершает выполнение всех потоков процесса;

- сохраняет статистические данные процесса и код возврата в его дескрипторе;

- переводит все ресурсы, принадлежащие процессу в непротиворечивое и стабильное состояние (например, закрывает все файлы);

- освобождает все ресурсы, принадлежавшие или использовавшиеся процессом;

- освобождает виртуальное адресное пространство процесса и уничтожает его;

- оповещает подсистемы, принимающие участие в управлении процессами, о завершении процесса с целью корректировки ими его дескриптора;

- состояние процесса устанавливает в значение «завершен».

В результате перечисленных действий от процесса остается только дескриптор в таблице дескрипторов ядра. Время его уничтожения в различных операционных системах определяется по-разному. В UNIX уничтожение дескриптора обычно производит родитель процесса с помощью системного вызова семейства wait() (при завершении любого процесса UNIX для его потомков родительским назначается всегда существующий процесс init). В Windows дескриптор процесса удаляется автоматически, когда счетчик его пользователей достигнет нуля.

Операции над потоками

1. Создание потока

Мы будем рассматривать случай, когда потоки поддерживаются непосредственно ядро операционной системы. При создании потока ядро должно выполнить следующие действия;

- создать дескриптор потока и поместить его в таблицу потоков;

- создать информационные структуры, необходимые для функционирования потока в данной аппаратной архитектуре (например, стек потока);

- проинициализировать значения полей общего назначения дескриптора потока исходя из параметров вызова или принятых значений по умолчанию;

- инициализировать поле дескриптора «аппаратный контекст выполнения потока» на основании параметров вызова или в соответствии со значениями по умолчанию (например, установить указатель стека на верхнюю границу стека, а указатель команд – на входную точку потока);

- оповестить подсистемы, принимающие участие в управлении потоками, о создании нового потока с целью завершения инициализации его дескриптора;

- перевести поток в состояние «готов к выполнению».

В UNIX для создания потока может использоваться системный вызов thr_create() (однако, будьте осторожны, в одних UNIX-системах поддерживается многопоточность на уровне ядра, в других - нет); в Windows – функции Win32 CreateThread(), CreateThreadEx().

2. Завершение потока

При завершении потока ядро операционной системы должно выполнить следующие действия:

- сохранить статистические данные потока и код возврата в его дескрипторе;

- перевести все ресурсы, принадлежащие потоку в непротиворечивое и стабильное состояние;

- освободить все ресурсы, принадлежавшие или использовавшиеся потоком;

- оповестить подсистемы, принимающие участие в управлении потоками, о завершении потока с целью корректировки ими его дескриптора;

- состояние потока устанавливает в значение «завершен»;

- если данный поток является последним активным потоком в процессе – завершить процесс.

В UNIX для завершения потока используется системный вызов thr_exit(); в Windows – функция Win32 ExitThread(). Для того, чтобы завершить поток извне, в UNIX используется thr_kill(), в Windows – TerminateThread().

3. Операции, осуществляющие переходы между различными состояниями потока

Множество данных операций было перечислено ранее при рассмотрении диаграммы состояний потока.

Планирование

Когда компьютер работает в многозадачном режиме, на нем может быть несколько потоков, находящихся в состоянии «готов к исполнению». Если доступен только один процессор, то необходимо выбирать, какому из потоков его предоставить. Отвечающая за это часть операционной системы называется планировщиком, а используемый алгоритм выбора – алгоритмом планирования.

Различные методы планирования по-разному подходят к решению задачи и могут использовать как информацию о потоках, так и информацию о процессах.

Модель поведения процесса с одним потоком

Практически все процессы чередуют периоды вычислений с операциями ввода-вывода. Обычно процессор некоторое время работает без остановки, затем происходит системный вызов на выполнение операции ввода-вывода (например, вызов на чтение из файла или записи в файл). После выполнения системного вызова процессор опять считает, пока ему, например, не понадобятся новые данные или не потребуется записать полученные данные и т.д.

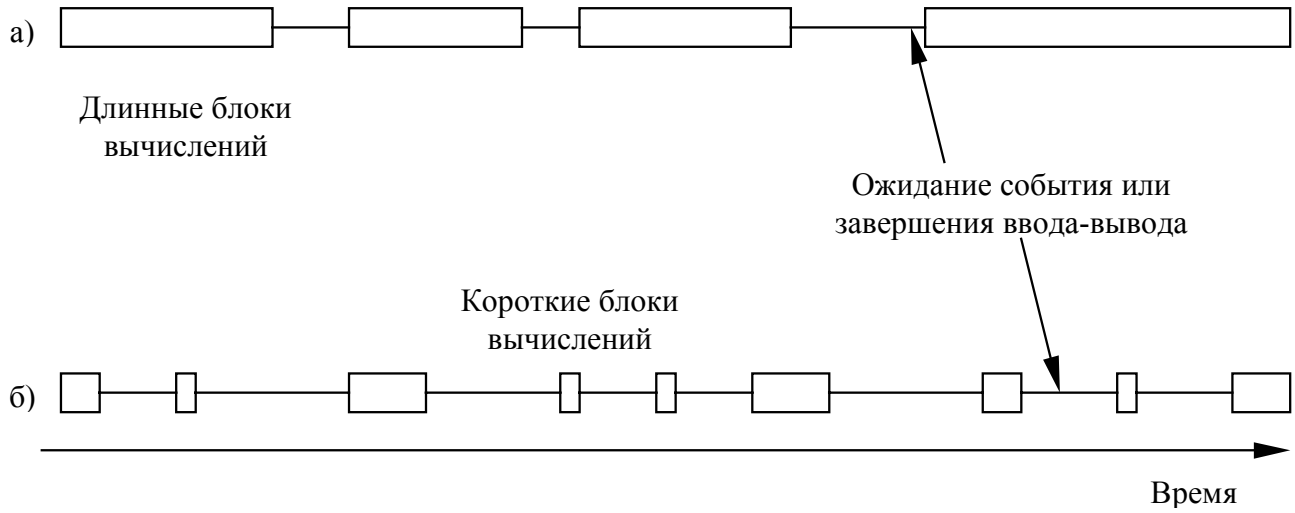


Рис. 10 Периоды использования процессора, чередующиеся с ожиданием ввода-вывода

Важно отметить, что некоторые процессы (а) большую часть времени заняты вычислениями, а другие (б) большую часть времени ожидают ввода-вывода. Первые называются ограниченными возможностями процессора, вторые - ограниченными возможностями устройств ввода-вывода. Основным параметром здесь является не время ожидания, а время вычислений, поскольку, например, время выполнения операции чтения с диска является величиной относительно постоянной (в отличие от времени ожидания внешнего события). Стоит отметить, что с увеличением скорости процессоров процессы становятся все более ограниченными возможностями устройств ввода-вывода.

График, приведенный на рисунке можно назвать временной диаграммой исполнения однопотокового процесса. В случае наличия нескольких процессов и нескольких потоков, каждый поток будет иметь подобную диаграмму, в которой времена выполнения будут отсчитываться исходя из времени выполнения потока на процессоре, а времена ожидания ввода-вывода будут абсолютные. К тому же, в многозадачной системе потоки ожидают освобождения не только аппаратных, но и программных ресурсов, время ожидания которых в общем случае предсказать невозможно.

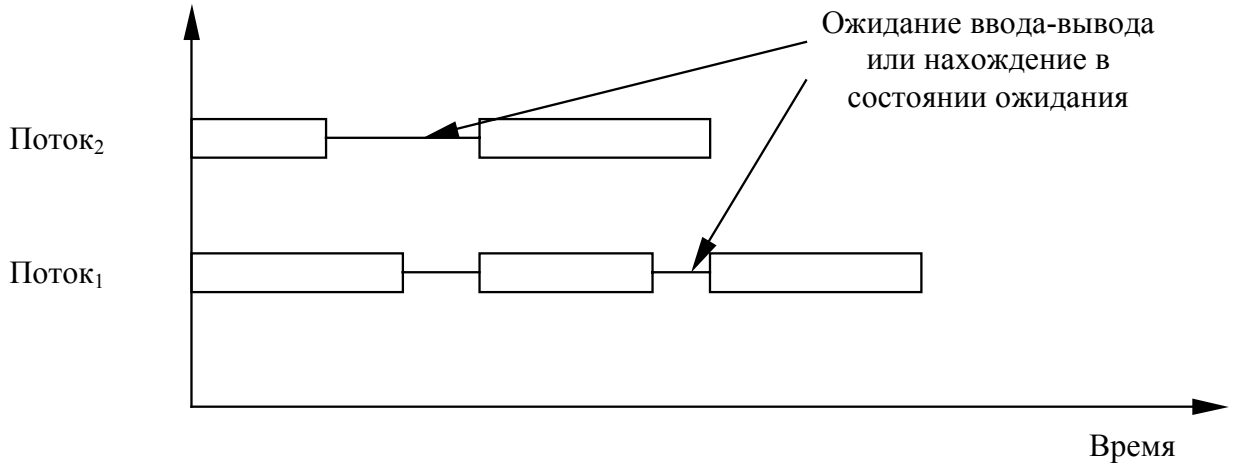


Рис. 11 Временные диаграммы двух потоков одного процесса

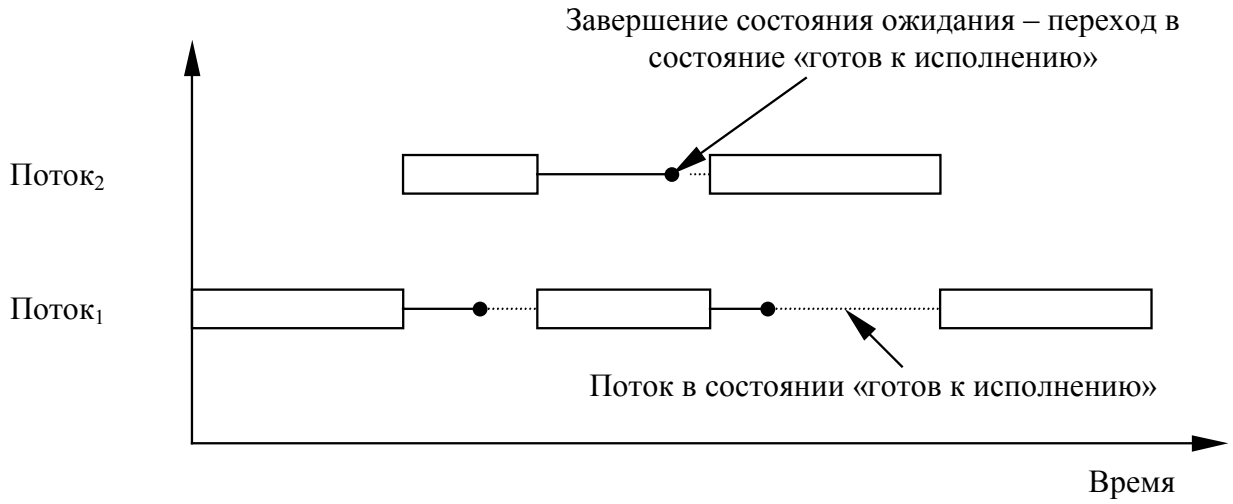


Рис. 12 Временная диаграмма использования центрального процессора

На рисунках видно, что при работе нескольких потоков, ограниченных возможностями процессора, неизбежны длительные пребывания потоков в состоянии «готов к исполнению».

Ключевым вопросом планирования является выбор момента принятия решений. Существует несколько ситуаций, в которых необходимо осуществлять планирование.

1. Создание нового процесса.

Необходимо решить, поток какого процесса будет выполняться – родительского или дочернего.

2. Создание нового потока.

Необходимо решить, какой поток будет выполняться – только что созданный или создавший поток.

3. Завершение потока или процесса.

В данной ситуации один или несколько потоков переходят в состояние «завершен» и перестают конкурировать за центральный процессор. Необходимо выбрать поток на исполнение из оставшихся в системе потоков, готовых к исполнению.

4. Блокирование потока (переход из состояния «выполнение» в состояние «ожидание»).

Если исполнение потока блокируется на операции ввода-вывода, запроса ресурсов или по какой-либо другой причине, необходимо выбрать и запустить другой поток.

5. Разблокирование потока (переход из состояния «ожидание» в состояние «готов к выполнению»).

В случае выполнения условия разблокирования какого-либо потока, например, при завершении операции ввода-вывода или выделении ему ожидаемого ресурса, данный поток переводится в состояние «готов к выполнению». Планировщик должен выбрать, какой поток поставить на выполнение – тот, который выполнялся в момент события разблокировки, или тот, который был разблокирован.

6. Возникновение аппаратного события.

При возникновении аппаратного события может быть инициировано прерывание центрального процессора и осуществлен запуск обработчика прерываний. По окончании обработки планировщик может выбрать на исполнение не тот поток, который был прерван. Например, если аппаратный таймер выполняет периодические прерывания с некоторой

частотой, решения по планированию могут приниматься при каждом прерывании по таймеру (или каждом k-ом).

С последним способом планирования связаны следующие два определения.

Невытесняющие алгоритмы планирования выбирают поток и позволяют ему работать вплоть до блокировки либо до того момента, когда поток сам отдаст центральный процессор.

Вытесняющие алгоритмы планирования выбирают поток и позволяют ему работать некоторое максимально возможное фиксированное время. Если к концу заданного интервала времени поток все еще работает, он приостанавливается, и управление переходит к другому потоку (если существует поток, готовый к исполнению). Вытесняющее планирование требует прерываний по таймеру, происходящих в конце отведенного периода времени. При отсутствии таймера возможна реализация только невытесняющих алгоритмов.

Критерии оценки алгоритмов планирования

Различные операционные системы и различные приложения ориентированы на решение различных задач. Соответственно, критерии оценки алгоритмов планирования в разных системах может быть различно. Традиционно выделяют три среды:

1. Системы пакетной обработки данных.

В системах пакетной обработки данных нет пользователей, сидящих за терминалами и ожидающих ответа. В таких системах приемлемы алгоритмы без переключений или с переключениями, но с большим временем, уделяемом каждому потоку. Такой подход уменьшает число переключений контекста процессов/потоков и повышает эффективность.

2. Интерактивные системы.

В интерактивных системах необходимы алгоритмы с переключениями, чтобы предотвратить захват процессора одним потоком – преднамеренный или как следствие ошибки программирования. Необходимо поддержание разумной степени мультипрограммирования за счет ограничения количества процессов и потоков, которые могут работать в системе одновременно, и понимание человеческой психологии. Если между нажатием на клавишу и появлением символа на экране проходит 20-30 секунд, то многие пользователи предпочтут прекратить работу и продолжить ее, когда система будет менее загружена, либо могут даже сменить операционную систему.

3. Системы реального времени.

В системах с ограничениями реального времени основной задачей всей системы в целом является гарантированная обработка поступающих сигналов в течение заданных временных отрезков.

Существует несколько критериев оценки алгоритмов планирования. Есть критерии, общие для всех сред, и есть критерии, имеющие значение только в какой-либо одно среде.

1. Общие критерии.

- **Справедливость** – гарантированное предоставление каждому потоку справедливой доли процессорного времени. В данном случае «справедливой» не значит «равной», однако сопоставимые потоки и процессы должны получать сопоставимое обслуживание.

- **Баланс** – поддержка занятости всех частей системы. Например, если выбирать на исполнение поочередно потоки, ограниченными возможностями процессора, и потоки, ограниченными возможностями устройств ввода-вывода, то загрузка как процессора, так и устройств ввода-вывода будет максимальной.

- **Накладные расходы** – отношение времени центрального процессора, использованного на работу алгоритма планировщика и переключения контекстов ко времени работы потоков пользовательских процессов.

- **Масштабируемость** – рост накладных расходов при увеличении числа процессов/потоков в системе должен происходить гладко.

2. Системы пакетной обработки данных.

- **Пропускная способность** – количество задач, выполненных за час (среднее число процессов, которые были созданы и завершили свое выполнение за единицу времени).

- **Оборотное время** – статистически усредненное время от момента создания процесса до его завершения. Характеризует время, которое среднестатистический пользователь ожидает выходных данных. Отметим, что прямая связь данного параметра с предыдущим отсутствует.

- **Эффективность** – определяется степенью загрузки центрального процессора. Цель – загрузить центральный процессор работой на все 100% рабочего времени.

3. Интерактивные системы.

- **Время отклика** – время между введением команды и получением результата, иначе, время, которое требуется потоку для ответа на запрос пользователя. Первоочередная обработка всех интерактивных запросов рассматривается как хорошее обслуживание.

- **Время ожидания** – среднестатистическое время, которое проводят потоки в состоянии «готов к исполнению» и задания в очереди для загрузки.

- **Соразмерность** – соответствие реального времени выполнения операций с представлениями пользователя о требуемом для этого времени. Данных параметр является число субъективно-психологическим и зависит от конкретного пользователя, но имеет немаловажное значение при рыночных реалиях.

4. Системы реального времени.

- **Окончание обслуживания к сроку** – строго говоря, это не является параметром оценки системы реального времени, поскольку в случае невыполнения данного требования система не может быть отнесена к рассматриваемому классу.

- **Предсказуемость** – одно и то же задание должно выполняться в течение сопоставимых сроков. На практике достаточно часто для систем реального времени указывают максимальное время выполнения различных внутренних операций ядра системы и исполнения некоторого множества системных вызовов.

Алгоритмы планирования в системах пакетной обработки данных

«Первым пришел – первым обслужен» (First-Come - First-Served, FCFS)

Невытесняющий алгоритм «первым пришел – первым обслужен» является самым простым алгоритмом планирования. Потокам предоставляется доступ к процессору в том порядке, в котором они его запрашивают. Чаще всего формируется единая очередь ждущих потоков. Как только появляется первая задача, она немедленно запускается и ее первичный поток работает столько, сколько необходимо. Остальные появляющиеся потоки образуют очередь в порядке их возникновения. Когда текущий поток блокируется, запускается следующий в очереди, а когда блокировка снимается, поток становится в конец очереди.

Основным преимуществом данного алгоритма является легкость его понимания и программирования. Все потоки в состоянии готовности к выполнению контролируются с использованием одной структурой данных – очереди.

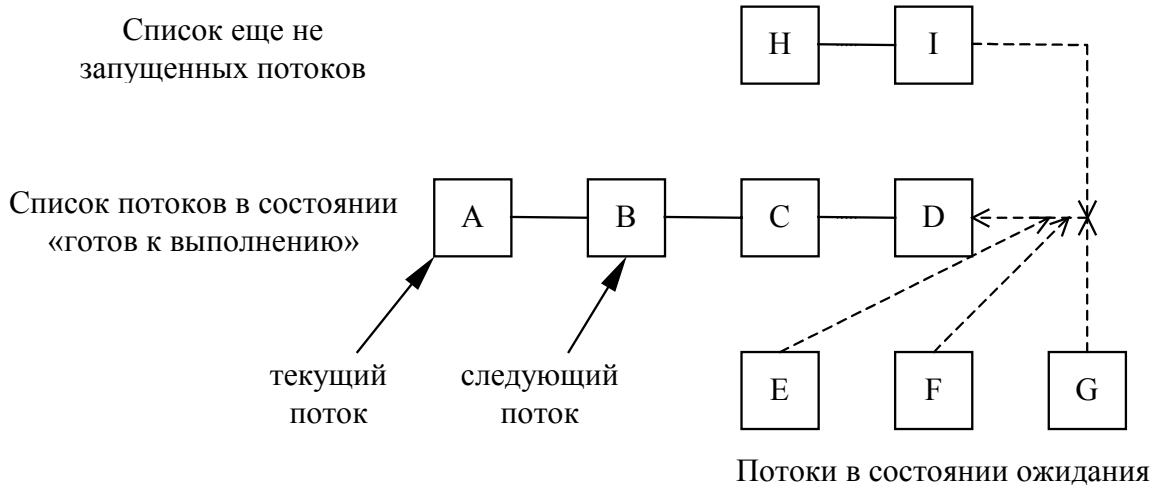


Рис. 13 Очередь потоков при планировании

Однако представим, что у нас есть два потока; первый в цикле производит вычисления в течение 1 с и потом пишет число в файл, второй записывает в цикле в файл последовательность из 1000 небольших блоков. Если позволить сначала выполниться второму потоку, и затем запустить первый, то время работы второго потока будет составлять считанные секунды. Если же запустить потоки вместе, то второй поток после каждой операции записи будет ждать 1 с, пока выполняется первый поток, и его общее время выполнения будет 1000 с.

«Кратчайшая задача - первая» (Shortest Job First, SJF)

Данный алгоритм также является невытесняющим. Он предполагает, что временные отрезки работы потоков известны заранее. Если в очереди готовых к исполнению потоков имеется несколько одинаково важных задач, планировщик выбирает на выполнение самую короткую задачу.

Подобная стратегия приводит к уменьшению среднего оборотного времени, однако, в случае стабильного поступления новых задач для выполнения, часть потоков с большими планируемыми временами исполнения может не получать центральный процессор очень долго.

Отметим также, что для реализации данного алгоритма необходим механизм сообщения планировщику информации о планируемом времени исполнения однопотоковых процессов и всех потоков многопоточных процессов. Обычно это реализуется посредством использования языка управления заданиями.

Еще одним моментом является невозможность в общем случае предсказать точное время выполнения процесса/потока и связанное с этим несоответствие планируемого и реального времен исполнения, а также намеренное уменьшение планируемого времени исполнения. Практически с этим можно бороться следующим способом – поток, израсходовавший запланированное время всегда находится в конце очереди готовых к исполнению.

Для реализации данного алгоритма также достаточно одного связного списка, с выполнением операции вставки в его середину.

«Наименьшее оставшееся время выполнения»

Этот невытесняющий алгоритм планирования является разновидностью предыдущего, только он учитывает не общее планируемое время выполнения потока, а планируемое время до окончания выполнения.

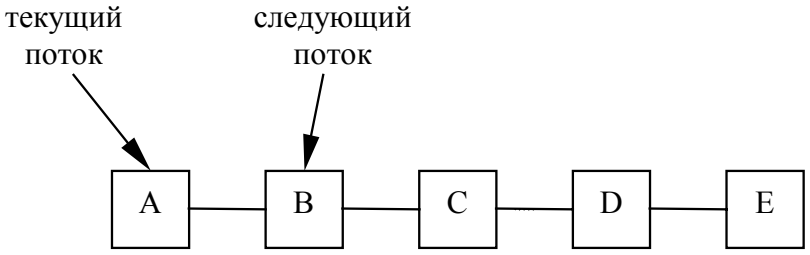
Такая схема позволяет быстро обслуживать короткие и близкие к завершению запросы.

Алгоритмы планирования в интерактивных системах

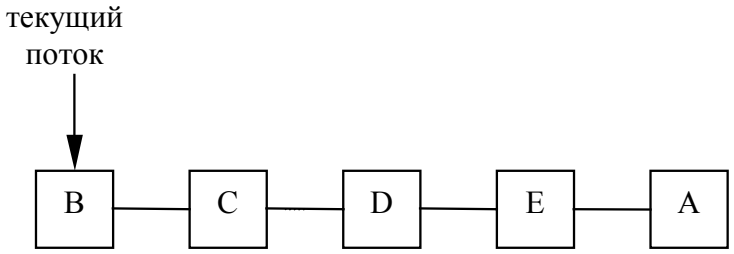
«Циклическое планирование» (Round Robin, RR)

Одним из наиболее старых, простых и справедливых алгоритмов планирования является вытесняющий алгоритм циклического планирования. Процессорное время делится на некоторые интервалы, так называемые кванты времени, и готовом к исполнению потокам в порядке очереди предоставляется такой квант. Если к концу кванта времени поток все еще работает, он прерывается, помещается в конец очереди готовых к исполнению, а управление передается другому потоку. Если поток блокируется или завершает работу до истечения предоставленного ему кванта, переход управления происходит немедленно.

Реализация алгоритма проста – необходимо лишь поддерживать очередь потоков в состоянии «готов к выполнению».



Список потоков в состоянии «готов к выполнению»



Список потоков в состоянии «готов к выполнению» после исчерпания потоком A своего кванта времени

Рис. 14 Очередь готовых к исполнению потоков при циклическом планировании

Интересным моментом данного алгоритма является выбор размера кванта времени. Если выбрать слишком маленький размер кванта, очень много времени будет уходить на переключения контекстов, если выбрать слишком большой – при большом числе потоков

сильно увеличится время отклика. Значение кванта обычно принимается в пределах 20-50 мс.

«Приоритетное планирование»

В циклическом алгоритме планирования есть важное допущение о том, что все потоки равнозначны. В действительности обычно это не так. Во-первых, возможно одновременное существование служебных процессов и процессов, запущенных пользователями. При этом важность выполнения служебных процессов может быть как достаточно высокой, так и достаточно низкой. Во-вторых, возможно ранжирование пользователей по статусу (например, декан-преподаватели-студенты). Необходимость принимать во внимание такие факторы приводит к приоритетному планированию.

Основная идея проста – каждому потоку присваивается приоритет, и управление передается готовому к выполнению потоку с самым высоким приоритетом.

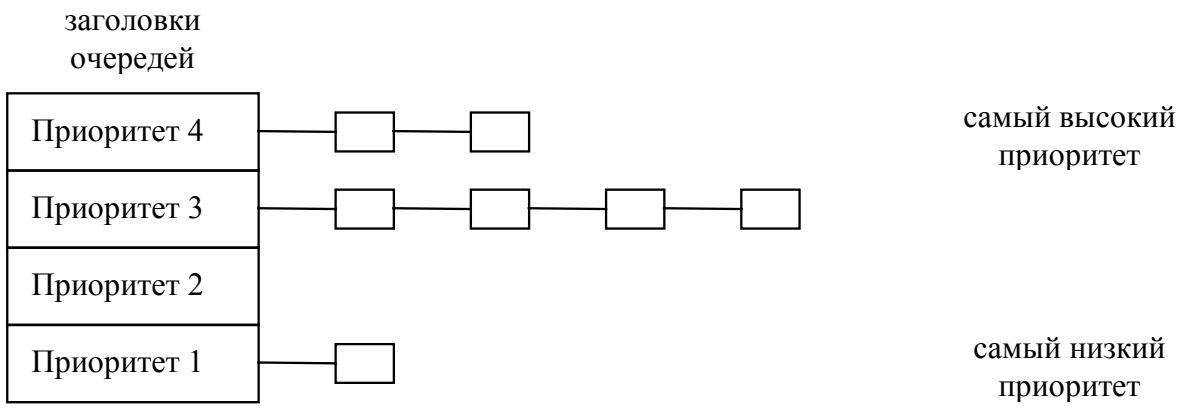


Рис. 15 Приоритетный алгоитм планирования с четырьмя уровнями приоритетов

Пока в очереди потоков с наивысшим приоритетов есть хотя бы один поток, они запускаются один за другим в порядке циклического планирования. Если в классе 4 нет готовых к исполнению потоков, обслуживаются потоки класса 3 и т.д.

При использовании приоритетного планирования, если в системе всегда готов к исполнению хотя бы один из высокоприоритетных потоков, то потоки с низкими приоритетами никогда не будут выполнены. Поэтому всегда используются некоторые модификации этого алгоритма.

В этих модификациях иногда разделяют базовый приоритет потока и эффективный приоритет потока. Базовый приоритет назначается потоку посредством некоторого системного вызова, например, инициированного выполнением команды пользователя. Эффективный приоритет используется планировщиком при выборе потоков на исполнение и может им изменяться в соответствии с реализованным алгоритмом планирования.

1. Для предотвращения бесконечной работы высокоприоритетных потоков планировщик может изменять с каждым тактом таймера эффективный приоритет выполняющегося потока, уменьшая его. Если в системе еще существуют готовые к исполнению потоки, то рано или поздно их приоритет станет выше, и один из них получит центральный процессор. Эффективный приоритет прерванного потока через некоторое время восстановится до первоначального значения.

2. Для предоставления низкоприоритетным потокам центрального процессора планировщик может проверять время, в течение которого поток находился в состоянии готовности к исполнению. В случае достижения некоторого критического значения потоку временно

назначается наивысший эффективный приоритет, который после получения квантов времени постепенно уменьшается до первоначального значения.

3. Динамическое изменение приоритетов.

Система может динамически назначать приоритеты для достижения своих целей. Например, если какой-то поток проводит большую часть времени в ожидании завершения операций ввода-вывода, то, очевидно, что когда бы ни потребовался данному потоку центральный процессор, его желательно немедленно предоставить, чтобы поток смог начать следующую операцию ввода-вывода.

Простой алгоритм обслуживания процессов, ограниченных возможностями устройств ввода-вывода, состоит в установке приоритета равным 1/f, где f – часть использованного в последний раз кванта.

При использовании динамических приоритетов в расчетах могут использоваться самые различные параметры, как статические для каждого потока, так и динамические, например:

- Каким пользователем запущен процесс или сформировано задание.

- Насколько важной является поставленная задача, т. е. каков приоритет ее выполнения.

- Сколько процессорного времени запрошено пользователем для решения задачи.

- Каково соотношение потребляемого процессорного времени и времени, необходимого для осуществления операций ввода-вывода.

- Какие ресурсы вычислительной системы (оперативная память, устройства ввода-вывода, специальные библиотеки и системные программы и т. д.) и в каком количестве необходимы заданию.

- Сколько времени прошло со времени выгрузки процесса на диск или его загрузки в оперативную память.

- Сколько оперативной памяти занимает процесс.

- Сколько процессорного времени было уже предоставлено процессу.

Рассмотрим несколько частных случаев.

«Самый короткий поток - следующий»

Алгоритм планирования «Кратчайшая задача - первая» минимизирует среднее оборотное время в системах пакетной обработки. Поскольку интерактивные процессы часто работают по схеме «ожидание команды – исполнение команды - ожидание команды», то одна итерация такого цикла может быть рассмотрена как отдельное задание, время оборота которого желательно минимизировать.

Один из методов оценки длины потока базируется на использовании информации о предыдущем поведении процесса. В ходе планирования ни исполнение выбирается поток, у которого оценочное время самое маленькое.

Проблема состоит в том, что мы можем делать только прогноз длительности следующего временного отрезка, в течение которого поток будет занимать центральный процессор, исходя из предыстории работы процесса.

Пусть t(n) - величина n-го времени использования ЦП, T(n + 1)- предсказываемое значение для n + 1-го времени использования, a - некоторая величина в диапазоне от 0 до 1.

Определим рекуррентное соотношение:

$$T(n+1) = a*t(n) + (1- a)*T(n),$$

T(0) положим произвольной неотрицательной константой.

Первое слагаемое учитывает последнее поведение процесса, тогда как второе слагаемое учитывает его предысторию. При a = 0 мы перестаем следить за последним поведением процесса, фактически полагая T(n) = T(n-1) =…=T(0), т.е. оценивая все времена использования ЦП одинаково, исходя из некоторого начального предположения. Положив a = 1, мы забываем о предыстории процесса. В этом случае мы полагаем, что время очередного использования ЦП будет совпадать со временем последнего использования ЦП: T(n+1) = t(n). При выборе a = 1/2 равноценно учитываются последнее поведение и предыстория. Отметим, что такой выбор a удобен и для быстрой организации вычисления оценки T(n+1). Для подсчета новой оценки нужно взять старую оценку, сложить с последним измеренным временем использования ЦП и полученную сумму разделить на 2, например, с помощью ее сдвига на 1 бит вправо. Полученные оценки T(n+1) применяются как продолжительности очередных промежутков времени непрерывного использования процессора для краткосрочного планирования «Самый короткий поток следующий».

«Гарантированное планирование»

При одновременной работе N пользователей в вычислительной системе можно применить алгоритм планирования, который гарантирует, что каждый из пользователей будет иметь в своем распоряжении примерно 1/N часть процессорного времени.

Пронумеруем всех пользователей от 1 до N. Для каждого пользователя с номером i введем две величины: T_i - время нахождения пользователя в системе, и t_i - суммарное процессорное время уже выделенное всем его потокам в течение сеанса. Справедливым для пользователя было бы получение T_i/N процессорного времени. Если $t << T_i / N$, то i - й пользователь несправедливо обделен процессорным временем. Если же $t >> T_i / N$, то система явно благоволит к пользователю с номером i. Вычислим для каждого пользовательского потока значение коэффициента справедливости $t_i * N / T_i$, и будем предоставлять очередной квант времени процессу с наименьшей величиной этого отношения. Предложенный алгоритм называют алгоритмом гарантированного планирования.

К недостаткам этого алгоритма можно отнести невозможность предугадать поведение пользователей. Если некоторый пользователь отправится на пару часов пообедать и поспать, не прерывая сеанса работы, то по возвращении его процессы будут получать неоправданно много процессорного времени.

Лотерейное планирование

Для простой реализации предсказуемых результатов используется алгоритм лотерейного планирования.

В основе алгоритма лежит раздача потокам лотерейных билетов на доступ к различным ресурсам, в том числе и процессору. Когда планировщику необходимо принять решение, выбирается случайным образом лотерейный билет, и его обладатель получает доступ к ресурсу. Более важным потоком в этом случае выдается большее число билетов. Каждый поток получает количество ресурсов, пропорциональное количеству имеющихся у него билетов. Например, если всего 100 билетов, и 20 из них находятся у одного потока, то в среднем ему будет доставаться около 20% времени центрального процессора.

Лотерейное планирование характеризуется несколькими интересными свойствами. Например, при изменении у потока количества билетов, его шансы увеличиваются уже в следующем розыгрыше, то есть лотерейное планирование обладает высоко отзывчивостью.

Кроме того, взаимодействующие потоки могут при необходимости обмениваться билетами в зависимости от того, кому нужнее в данный момент тот или иной ресурс. И, наконец, с помощью лотерейного планирования можно легко организовать загрузку центрального процессора несколькими постоянно готовыми к исполнению потоками в определенной пропорции, например 50:20:20:2:1.

Планирование с использованием многоуровневых очередей

Для систем, в которых процессы и их потоки могут быть рассортированы на разные группы, был разработан другой класс алгоритмов планирования. Для каждой группы потоков создается своя очередь потоков, находящихся в состоянии готовность. Этим очередям приписываются фиксированные приоритеты. Например, приоритет очереди потоков системных процессов устанавливается больше, чем приоритет очередей потоков пользовательских процессов. А приоритеты очередей пользователей выстраиваются, например, в соответствии с их званиями.

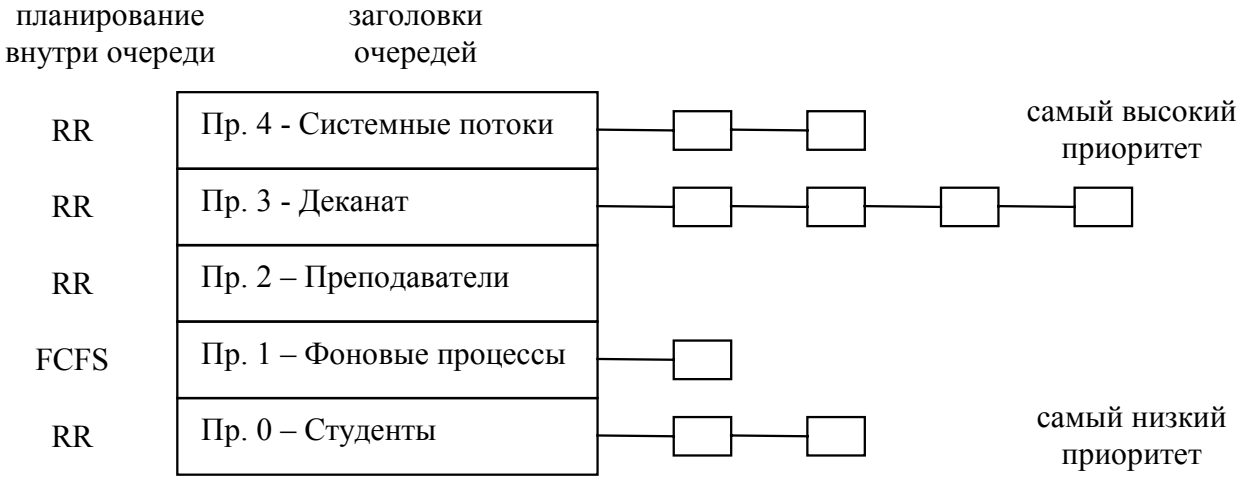


Рис. 16 Приоритетный алгоитм планирования с несколькими очередями

Это значит, что ни один пользовательский поток не будет выбран для исполнения, пока есть хоть один готовый системный поток, и ни один студенческий поток не получит в свое распоряжение процессор, если есть потоки преподавателей, готовые к исполнению. Внутри этих очередей для планирования могут применяться самые разные алгоритмы. Так, например, для больших счетных процессов, не требующих взаимодействия с пользователем (фоновых процессов), может использоваться алгоритм «Первым пришел – первым обслужен» (FCFS), а для интерактивных процессов - алгоритм циклического обслуживания (RR). Подобный подход, получивший название многоуровневых очередей, повышает гибкость планирования: для процессов с различными характеристиками применяется наиболее подходящий им алгоритм.

Планирование с использованием многоуровневых очередей с обратной связью

Дальнейшим развитием алгоритма многоуровневых очередей является добавление к нему механизма обратной связи. Здесь поток не постоянно приписан к определенной очереди, а может мигрировать из очереди в очередь, в зависимости от своего поведения. Рассмотрим пример.

Первичный поток только что созданного процесса поступает в очередь с приоритетом 4. При выборе на исполнение он получает в свое распоряжение квант времени размером 8 единиц. Если он фактически использует время ЦП меньше этого кванта времени, поток

остается в очереди 4. В противном случае, он переходит в очередь 3. Для потоков из очереди 3 квант времени имеет величину 16. Если поток не укладывается в это время, он переходит в очередь 2. Если укладывается - остается в очереди 3. В очереди 2 величина кванта времени составляет 32 единицы. Если и этого мало для непрерывной работы потока, он поступает в очередь 1, для которой квантование времени не применяется, и, при отсутствии готовых потоков в других очередях, он может исполняться столько времени, сколько ему нужно. Чем больше значение продолжительности времени между операциями ввода-вывода или блокировками потока, тем в менее приоритетную очередь он попадает, но тем на большее процессорное время он может рассчитывать для своего выполнения. Таким образом, через некоторое время все потоки, требующие малого времени работы процессора окажутся размещенными в высокоприоритетных очередях, а все потоки, требующие большого счета и с низкими запросами к времени отклика - в низкоприоритетных.

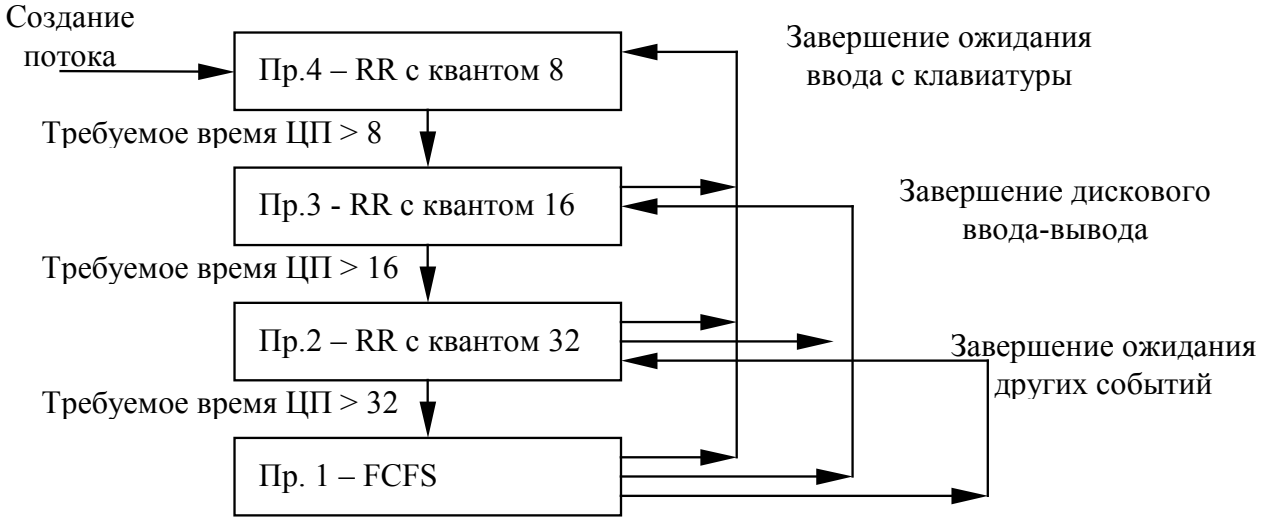


Рис. 17 Приоритетный алгоитм планирования с несколькими очередями и обратной связью

Миграция потоков в обратном направлении может осуществляться по различным принципам. Например, после завершения ожидания ввода с клавиатуры потоки из очередей 1, 2 и 3 могут помещаться в очередь 4, после завершения дисковых операций ввода-вывода потоки из очередей 1 и 2 могут помещаться в очередь 3, а после завершения ожидания всех других событий – из очереди 1 в очередь 2. Перемещение потоков из очередей с низкими приоритетами в очереди с большими приоритетами позволяет более полно учитывать изменение поведения потоков с течением времени.

Многоуровневые очереди с обратной связью представляют собой наиболее общий подход к планированию из числа рассмотренных подходов. Они наиболее трудоемки в реализации, но в то же время они обладают наибольшей гибкостью. Естественно, существует много других разновидностей такого способа планирования помимо варианта, приведенного выше. Для полного описания конкретного воплощения необходимо указать:

- Количество очередей для потоков, находящихся в состоянии готов к выполнению.
- Алгоритм планирования, действующий между очередями.
- Алгоритмы планирования, действующие внутри очередей.
- Правила помещения родившегося потока в одну из очередей.
- Правила перевода потоков из одной очереди в другую.

Изменяя какой-либо из перечисленных пунктов, мы можем существенно менять поведение вычислительной системы.

Алгоритм планирования Windows NT

В Windows NT реализована вытесняющая многозадачность. Планировщик использует для определения порядка выполнения потоков алгоритм, основанный на приоритетах, в соответствии с которым каждому потоку присваивается число - приоритет, и потоки с более высоким приоритетом выполняются раньше потоков с меньшим приоритетом.

Процесс получает базовый приоритет в тот момент, когда его создает подсистема той или иной прикладной среды. Значение базового приоритета присваивается процессу системой по умолчанию, или системным администратором, или указывается при вызове родительским процессом. Поток наследует этот базовый приоритет и может изменить его, немного увеличив или уменьшив. На основании получившегося в результате приоритета, называемого приоритетом планирования, производится планирование выполнения потоков.

Windows NT поддерживает 32 уровня приоритетов, разделенных на два класса - класс реального времени и класс переменных приоритетов. Так называемые потоки реального времени, приоритеты которых находятся в диапазоне от 16 до 31, являются более приоритетными и используются для выполнения задач, критичных ко времени. Каждый раз, когда необходимо выбрать поток для выполнения, диспетчер прежде всего просматривает очередь готовых к выполнению потоков реального времени и обращается к другим потокам, только когда очередь потоков реального времени пуста. Большинство потоков в системе попадают в класс потоков с переменными приоритетами, диапазон значений - от 0 до 15. Этот класс имеет название "переменные приоритеты", потому что диспетчер настраивает систему, изменяя (понижая или повышая) приоритеты потоков этого класса.

Для того чтобы обеспечить хорошее время реакции системы, алгоритм планирования использует концепцию абсолютных приоритетов, в соответствии с которой при появлении в очереди готовых потоков такого, у которого приоритет выше, чем у выполняющегося в данный момент, происходит смена выполняемого потока на поток с самым высоким приоритетом.

Использование динамических приоритетов, изменяющихся во времени, позволяет реализовать адаптивное планирование, при котором не дискриминируются интерактивные задачи, часто выполняющие операции ввода-вывода и недоиспользующие выделенные им кванты. Например, приоритет потока, окно которого владеет фокусом ввода, в зависимости от настроек системы может быть увеличен на 1 или 2; приоритет потока, который начинает обработку события клавиатурного ввода, временно увеличивается на 2. Если поток полностью исчерпал свой квант, то его приоритет понижается на некоторую величину. В то же время приоритет потоков, которые перешли в состояние ожидания, не использовав полностью выделенный им квант, повышается. Приоритет не изменяется, если поток вытеснен более приоритетным потоком.

В многопроцессорных системах при планировании выполнения потоков играет роль их процессорная совместимость: после того, как планировщик выбрал поток с наивысшим приоритетом, он проверяет, какой процессор может выполнить данный поток и, если атрибут потока "процессорная совместимость" не позволяет потоку выполняться ни на одном из свободных процессоров, то выбирается следующий в порядке приоритетов поток.

Алгоритм планирования UNIX

ОС UNIX изначально была многозадачной системой, ее алгоритм планирования с самого начала разрабатывался так, чтобы обеспечить хорошую реакцию в интерактивных процессах. У этого алгоритма два уровня. Низкоуровневый алгоритм выбирает следующий процесс из набора процессов в памяти и готовых к работе. Высокоуровневый алгоритм перемещает

процессы из памяти на диск и обратно, что предоставляет всем процессам возможность попасть в память и быть запущенными.

У каждой версии UNIX свой слегка отличающийся низкоуровневый алгоритм планирования, но у большинства этих алгоритмов есть много общих черт, которые мы опишем. В низкоуровневом алгоритме используется несколько очередей. С каждой очередью связан диапазон непересекающихся значений приоритетов. Процессы, выполняющиеся в режиме пользователя, имеют положительные значения приоритетов. У процессов, выполняющихся в режиме ядра (или обратившихся к системным вызовам), значения приоритетов отрицательные. Отрицательные значения приоритетов считаются наивысшими, а положительные - наоборот, минимальными. В очередях располагаются только процессы, находящиеся в памяти и готовые к работе.

Когда запускается (низкоуровневый) планировщик, он просматривает очереди, начиная с самого высокого приоритета, пока не находит очередь, в которой есть хотя бы один процесс. После этого из этой очереди он выбирает и запускает первый процесс. Процессу разрешается работать в течение некоего максимального кванта времени (обычно 100 мс) или пока он не заблокируется. Если процесс использует весь свой квант времени, он помещается обратно, в конец очереди, а алгоритм планирования запускается снова. Таким образом, процессы, входящие в одну группу приоритетов, используют центральный процессор в порядке циклической очереди.

Раз в секунду приоритет каждого процесса пересчитывается по следующей формуле:

$$priority = CPU\_usage + nice + base$$

На основе сосчитанного нового приоритета каждый процесс прикрепляется к соответствующей очереди. Для получения номера очереди приоритет, как правило, делится на некую константу. Рассмотрим компоненты этой формулы.

CPU_usage – использование центрального процессора – представляет собой среднее значение тиков таймера в секунду, которые процесс работал в течение нескольких последних секунд. При каждом тике таймера счетчик активного процесса увеличивается на единицу.

Для того, чтобы избежать неограниченного накапливания слагаемого CPU_usage, его величина со временем уменьшается. В различных версиях это выполнено по-разному. Один из способов состоит в делении CPU_usage в конце каждой секунды на 2, другой – в выполнении в конце секунды операции CPU_usage = CPU_usage * 2*n/(2*n+1), где n – число процессов в системе.

nice – показатель «любезности» процесса, может принимать значения, например, от -20 до +19. Пользователь может назначить показатель nice в диапазоне от 0 до +19. Только системный администратор может назначить процессу показатель nice от -1 до-20.

base – если процесс эмулирует прерывания для выполнения системного вызова, он переходит в состояние ожидания. Когда процесс блокируется, он удаляется из структуры очереди до тех пор, пока он не будет снова готов к работе. Параметр base представляет собой жестко прошитое в операционной системе повышение приоритета процесса, выполнившего системный вызов. В основе такой схемы лежит идея как можно более быстрого удаления процессов из ядра.

Алгоритм планирования UNIX System V Release 4

В системе UNIX System V Release 4 реализована вытесняющая многозадачность, основанная на использовании приоритетов и квантования. В ней нет поддержки многопоточной

организации процессов на уровне ядра, хотя и есть два системных вызова для организации потоков в пользовательском режиме.

Все процессы разбиты на несколько групп, называемых классами приоритетов. Каждая группа имеет свои характеристики планирования процессов.

Созданный процесс наследует характеристики планирования процесса-родителя, которые включают класс приоритета и величину приоритета в этом классе. Процесс остается в данном классе до тех пор, пока не будет выполнен системный вызов, изменяющий его класс.

При инсталляции системы возможно включение новых классов приоритетов. По умолчанию имеется три приоритетных класса: класс реального времени, класс системных процессов и класс процессов разделения времени. Приоритетность (привилегии) процесса тем выше, чем больше число, выражающее приоритет. На рисунке показаны диапазоны изменения приоритетов для разных классов. Значения приоритетов определяются для разных классов по-разному.

Приоритетный класс	Выбор планировщика	Глобальное значение приоритета
Реального времени (real time)	первый	100-159
Системные процессы (system)	...	60-99
Процессы разделения времени (time-shared)	последний	0-59
Возможно добавление новых классов		

Рис. 18 Приоритетные классы процессов UNIX System V Release 4

Процессы системного класса используют стратегию фиксированных приоритетов. Системный класс зарезервирован для процессов ядра. Уровень приоритета процессу назначается ядром и никогда не изменяется. Заметим, что пользовательский процесс, перешедший в системную фазу, не переходит при этом в системный класс приоритетов.

Процессы реального времени также используют стратегию фиксированных приоритетов, но пользователь может их изменять. Так как при наличии готовых к выполнению процессов реального времени другие процессы не рассматриваются, то процессы реального времени надо тщательно проектировать, чтобы они не захватывали процессор на слишком долгое время. Характеристики планирования процессов реального времени включают две величины: уровень глобального приоритета и квант времени. Для каждого уровня приоритета имеется по умолчанию своя величина кванта времени. Процессу разрешается захватывать процессор на указанный квант времени, а по его истечении планировщик снимает процесс с выполнения.

Для справедливого распределения времени процессора между процессами разделения времени, в этом классе используется стратегия динамических приоритетов, которая адаптируется к операционным характеристикам процесса.

Величина приоритета, назначаемого процессам разделения времени, вычисляется пропорционально значениям двух составляющих: пользовательской части и системной части. Пользовательская часть приоритета может быть изменена суперпользователем и владельцем процесса, но в последнем случае только в сторону его снижения.

Системная составляющая позволяет планировщику управлять процессами в зависимости от того, как долго они используют процессор, не уходя в состояние ожидания. Тем процессам, которые потребляют большие периоды времени без ухода в состояние ожидания, приоритет

снижается, а тем процессам, которые часто уходят в состояние ожидания после короткого периода использования процессора, приоритет повышается. Таким образом, процессам, ведущим себя не по-джентельменски, дается низкий приоритет, что означает, что они реже выбираются на выполнение. Но процессам с низким приоритетом даются большие кванты времени, чем процессам с высокими приоритетами. Таким образом, хотя низкоприоритетный процесс и не работает так часто, как высокоприоритетный, но зато, когда он наконец выбирается на выполнение, ему отводится больше времени.

Планировщик использует, в частности, следующие характеристики для процессов разделения времени:

- величина глобального приоритета;

- количество тиков таймера, которые отводятся процессу до его вытеснения (размер кванта);

- число тиков таймера, оставшихся в кванте процесса;

- системная составляющая приоритета процесса (базовое значение);

- системная составляющая приоритета, назначаемая процессу при истечении его кванта времени;

- системная составляющая приоритета, назначаемая процессу после выхода его из состояния ожидания; ожидающим процессам дается высокий приоритет, так что они быстро получают доступ к процессору после освобождения ресурса;

- величина системной составляющей приоритета, назначаемая процессу, если он находится в состоянии «готов к выполнению» длительное время.

- системная часть приоритета процесса;

- верхний предел и текущее значение пользовательской части приоритета (эти два атрибута могут модифицироваться пользователем);

- nice - используется для обратной совместимости с системным вызовом nice; содержит текущее значение величины nice, которая влияет на результирующую величину приоритета. Чем выше эта величина, тем меньше приоритет.

Алгоритм планирования Linux (версия ядра 2.2 и ниже)

Потоки в системе Linux реализованы в ядре, поэтому планирование основано на потоках, а не на процессах. В операционной системе Linux алгоритмом планирования различаются три класса потоков:

1. Потоки реального времени, обслуживаемые по алгоритму FIFO (First In First Out - первым прибыл - первым обслужен).

2. Потоки реального времени, обслуживаемые в порядке циклической очереди.

3. Потоки разделения времени.

Потоки реального времени, обслуживаемые по алгоритму FIFO, имеют наивысшие приоритеты и не могут прерываться другими потоками, за исключением такого же потока реального времени FIFO, перешедшего в состояние готовности.

Потоки реального времени, обслуживаемые в порядке циклической очереди (RR), представляют собой то же самое, что и потоки времени, обслуживаемые по алгоритму FIFO, но с тем отличием, что для них указывается размер кванта, и они могут прерываться таймером. По окончании кванта RR-поток становится в конец очереди потоков своего уровня приоритета. Ни один из классов на самом деле не является классом реального времени. Здесь

нельзя задать предельный срок выполнения задания и предоставить гарантий его выполнения. Эти классы просто имеют более высокий приоритет, чем потоки стандартного класса разделения времени.

У каждого потока есть приоритет планирования (по умолчанию +20), но его значение может быть изменено системным вызовом nice(value), value={-20,..,19}. Значение приоритета всегда находится в диапазоне от 1 до 40.

Помимо приоритета, с каждым процессом связан квант времени, то есть количество тиков таймера, в течение которых процесс может выполняться. По умолчанию каждый тик равен 10 мс. Этот интервал называют "джиффи" (jiffy - мгновение, миг, момент).

Планировщик использует приоритет и квант следующим образом. Сначала он вычисляет называемую в системе Linux "добродетелью" (goodness) величину каждого готового процесса по следующему алгоритму:

if (class == real_time) goodness = 1000 + priority;

if (class == timesharing && quantum > 0) goodness = quantum + priority;

if (class == timesharing && quantum == 0) goodness = 0;

Для обоих классов реального времени выполняется первое условие.

Когда нужно принять решение, выбирается поток с максимальным значением «добродетели». Во время работы процесса его квант уменьшается на единицу с каждым тиком. Центральный процессор отнимается у потока при выполнении одного из следующих условий:

1. Квант потока уменьшился до 0.

2. Поток блокируется на операции ввода-вывода, семафоре и т.п.

3. В состояние готовности перешел ранее заблокированный поток с более высоким значением «добродетели».

Так как кванты постоянно уменьшаются, рано или поздно у всех готовых к выполнению потоков квант станет нулевым (при этом у потоков, находящихся в других состояниях, например, заблокированных, значение goodness может быть ненулевым). В этот момент планировщик пересчитывает значения квантов времени для всех потоков по формуле

$$quantium = (quantium/2) + priority$$

Таким образом, потоки, ограниченные вводом-выводом, получат преимущество при планировании, а потоки, ограниченные производительностью процессора будут его использовать пропорционально отношению их приоритетов.

Заключение

Одним из наиболее ограниченных ресурсов вычислительной системы является процессорное время. Для его распределения между многочисленными процессами в системе приходится применять процедуру планирования процессов.

По степени длительности влияния планирования на поведение вычислительной системы различают краткосрочное, среднесрочное и долгосрочное планирование процессов. Конкретные алгоритмы планирования процессов зависят от поставленных целей, класса решаемых задач и опираются на статические и динамические параметры процессов и компьютерных систем.

Имеется ряд критериев, по которым оценивается эффективность алгоритмов планирования.

Управление памятью

Операционные системы выполняют две основные функции: расширение возможностей машины и управление ее ресурсами. С точки зрения пользователя, операционная система выполняет функцию расширенной машины, в которой проще программировать и легче работать, чем непосредственно с аппаратным обеспечением, составляющим реальный компьютер. Для прикладных программ операционная система предоставляет операционную среду – виртуальную машину, на которой, собственно, и выполняются программы. С другой стороны, операционная система обеспечивает организованное и контролируемое распределение аппаратных ресурсов системы между различными программами, состязающимися за право их использовать.

Оперативная память – один из важнейших аппаратных ресурсов. Она необходима для работы любой программы, но допускает пространственное мультиплексирование, то есть возможно одновременное использование несколькими программами частей оперативной памяти. Например, если памяти достаточно для одновременного хранения нескольких программ, то, используя центральный процессор по очереди (в соответствии с некоторой дисциплиной диспетчеризации), они способны обеспечить более эффективную загрузку аппаратных ресурсов, чем одна программа, если бы она занимала всю оперативную память.

Часть операционной системы, отвечающая за управление памятью, называется модулем управления памятью или менеджером памяти. В общем случае, он имеет две составляющие - аппаратно-зависимую и аппаратно-независимую.

При изучении, рассмотрении и разработке методов управления оперативной памятью необходимо учитывать следующие аспекты:

- целевая архитектура памяти;

- алгоритм формирования адреса в целевой архитектуре и логическая структура адресного пространства;

- способ описания физической памяти;

- что представляет собой виртуальное адресное пространство прикладной программы (ВАП) и каким образом оно отображается на физическую память;

- алгоритм связывания адресов программных модулей с адресами ВАП;

- алгоритм обеспечения пространственного мультиплексирования.

Кратко объясним содержание перечисленных пунктов.

Архитектура оперативной памяти

Вычислительная система содержит несколько видов памяти, отличающихся способом доступа к ней, скоростью доступа и стоимостью.

Рис. 19 Иерархия памяти компьютера

Из представленных на рисунке видов памяти только к двум – регистрам ЦПУ и главной памяти – процессор может получить доступ, используя прямую адресацию (то есть в команде процессора можно явно указать, какой именно регистр или ячейку оперативной памяти мы хотим использовать).

Простейшим вариантом организации главной памяти является непрерывное однородное адресуемое пространство памяти. В этом случае, например, имеется N байт оперативной памяти с адресами от 0 до N-1, к любому из которых процессор может обратиться при выполнении любой операции, требующей работы с памятью, и обладающие одинаковыми характеристиками.

Перечислим несколько причин, порождающих отступления от данного варианта.

- Различают фон-неймановскую (или принстонскую) а гарвардскую архитектуры памяти. В фон-неймановской архитектуре для хранения команд и данных используется одно общее адресное пространство, в гарвардской – два различных.

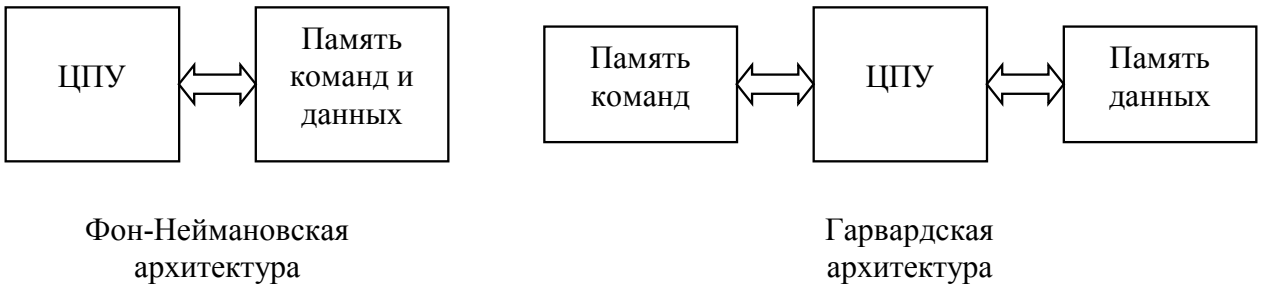


Рис. 20 Фон-Неймановская и Гарвардская архитектуры памяти

- Существует Оперативное Запоминающее Устройство (ОЗУ, RAM – Random Access Memory) , допускающее выполнение команд ЦПУ, производящих запись, и Постоянное Запоминающее Устройство (ПЗУ, ROM – Read Only Memory), допускающее выполнение только команд чтения. При этом ОЗУ и ПЗУ могут быть объединены в одно адресное пространство с целью использования для работы с ними одних и тех же команд процессора. Действия при попытке записи в ПЗУ зависят от реализации, например, это может быть игнорирование команды.

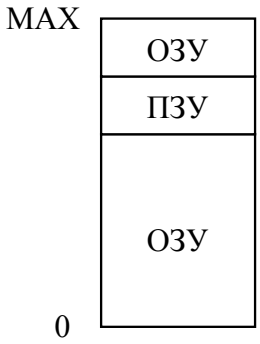


Рис. 21 Физическое адресное пространство, содержащее ОЗУ и ПЗУ

- Некоторые устройства имеют собственную оперативную память для решения специализированных задач, например, память графического адаптера содержит растровое изображение, на основе которого формируется сигнал, посылаемый для отображения на мониторе. Эта память также может быть отображена на одно адресное пространство с ОЗУ с целью обеспечения прозрачного доступа и, возможно, ускорения доступа к ней.

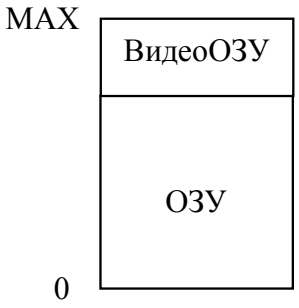


Рис. 22 Физическое адресное пространство, содержащее ОЗУ и ВидеоОЗУ

- Часть устройств, поддерживающих работу в режиме прямого доступа к памяти (выполнение операций чтения/записи без участия ЦПУ), способны работать только с ограниченным набором адресов. Например, ISA-устройства персональных компьютеров способны работать только с физической памятью в диапазоне адресов от 0 до 16Мб.

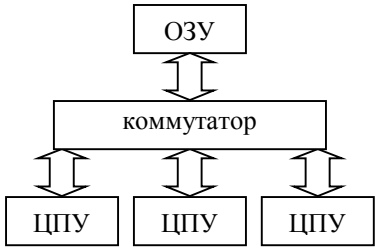
Рис. 23 Некоторые устройства предъявляют требования к адресам при работе в режиме прямого доступа к памяти

- Существуют системы, в которых количество установленной памяти может превышать возможности адресации ЦПУ. Например, если ЦПУ использует 32-разрязрядную адресацию, то он может максимально адресовать 4Гб. При наличие памяти большего объема, требуются специальные механизмы, обеспечивающие доступ ко всей памяти.

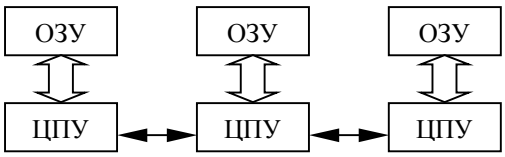
Рис. 24 Объем физической памяти превышает возможности адресации ЦПУ

- В случае многопроцессорных систем существует несколько основных походов к организации памяти: системы с общей памятью, системы с индивидуальной памятью, системы с неоднородной памятью, а также системы, использующие комбинации предыдущих.

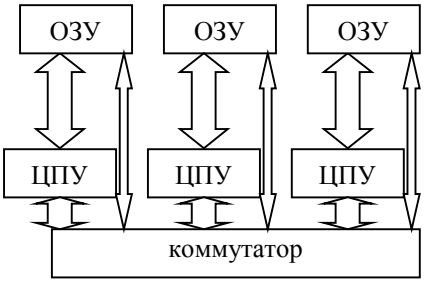
В системах с общей памятью несколько процессоров работают с одним физическим адресным пространством памяти. В системах с индивидуальной памятью каждый процессор имеет свое собственное физическое адресное пространство, недоступное другим процессорам. В системах с неоднородной памятью каждый процессор имеет индивидуальную память и может обращаться к памяти других процессоров, но время доступа к «своей» памяти гораздо меньше.



система с общей памятью



система с индивидуальной памятью



система с неоднородной памятью

Рис. 25 Принципы организации доступа к памяти в многопроцессорных системах

Естественно, в каждой конкретной системе сочетание нескольких факторов может быть уникальным.

Программное обеспечение (и в том числе операционные системы) создают для решения конкретных задач на конкретном аппаратном обеспечении. Соответственно, метод управления существенно будет зависеть как от архитектуры памяти, так и от критериев оценки эффективности, порожденных поставленными задачами.

Алгоритм формирования физического адреса и логическая структура адресного пространства

Если у нас имеется адресное пространство оперативной памяти, то мы можем его рассматривать как множество однородных адресуемых байт данных.

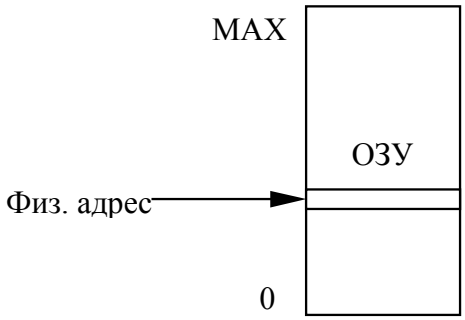


Рис. 26 Однородное множество значений

В данном случае под методом адресации подразумевается способ формирования адреса из одного или нескольких операндов команды ЦПУ. В таблице приведены наиболее употребительные названия методов адресации, хотя при описании архитектуры в документации разные производители используют разные названия для этих методов. Сочетание R1 означает первый регистр, буква M обозначает память (Memory); таким образом, M[R1] обозначает содержимое ячейки памяти, адрес которой определяется содержимым регистра R1.

Метод адресации	Пример команды	Смысл команды метода	Использование
Регистровая	Add R4,R3	R4<--R4+R5	Требуемое значение в регистре.
Непосредственная или литеральная	Add R4,#3	R4<--R4+3	Для задания констант.
Базовая со смещением	Add R4,100(R1)	R4<--R4+M[100+R1]	Для обращения к локальным переменным.
Косвенная регистровая	Add R4,(R1)	R4<--R4+M[R1]	Для обращения по указателю или вычисленному адресу.
Индексная	Add R3,(R1+R2)	R3<--R3+M[R1+R2]	Иногда полезна при работе с массивами: R1 - база, R3 – индекс.
Прямая или абсолютная	Add R1,(1000)	R1<--R1+M[1000]	Иногда полезна для обращения к статическим данным.

Косвенная	Add R1,@(R3)	R1<--R1+M[M[R3]]	Если R3-адрес указателя p, то выбирается значение по этому указателю.
Автоинкрементная	Add R1,(R2)+	R1<--R1+M[R2] R2<--R2+d	Полезна для прохода в цикле по массиву с шагом: R2 - начало массива. В каждом цикле R2 получает приращение d.
Автодекрементная	Add R1,(R2)-	R2<--R2-d R1<--R1+M[R2]	Аналогична предыдущей. Обе могут использоваться для реализации стека.
Базовая индексная со смещением и масштабированием	Add R1,100(R2)[R3]	R1<-- R1+M[100]+R2+R3*d	Для индексации массивов

Однако при решении задач управления памятью приходится манипулировать не отдельными байтами адресного пространства, а их группами, и алгоритмы управления основываются на возможностях работы с блоками адресного пространства, поддерживаемых на аппаратном уровне.

В этом случае часто используются составные адреса. Первая часть такого адреса определяет логический блок, в котором располагается адресуемый элемент, а вторая – смещение в этом блоке. Вышеперечисленные методы адресации используются в таких случаях для формирования второй части, то есть **смещения**.

В настоящее время при создании групп однородной информации часто используется понятие **сегмента** и **сегментной организации памяти**. Сегмент представляет собой часть общего адресного пространства и содержит внутри себя линейную последовательность адресов от 0 до некоторого максимума.

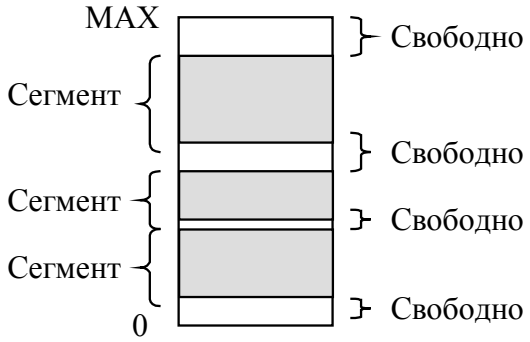


Рис. 27 Выделение нескольких сегментов

Сегмент может иметь атрибуты:

- адрес начала сегмента;
- длина сегмента;
- тип информации, размещенной в сегменте (код, данные, стек и др.);
- уровень привилегированности;
- доступные операции над сегментом (чтение, запись, выполнение и др.);
- другие атрибуты.

Для определения адреса в сегментированной памяти необходимо использовать адрес, состоящий из двух частей: идентификатор (селектор) сегмента и смещение внутри сегмента, указываемое в соответствии с одним из перечисленных выше правил адресации. Физический адрес определяется как сумма адреса начала сегмента и смещения.

В различных реализациях сегмент может не иметь атрибутов или иметь атрибуты по умолчанию, адрес начала сегмента может формироваться из значения идентификатора сегмента на основе некоторого алгоритма. Однако в настоящий момент доминирующим является подход, при котором используются **таблицы дескрипторов сегментов**, в которых содержатся атрибуты сегментов. В этом случае идентификатор сегмента содержит указание на используемую таблицу дескрипторов сегментов и номер дескриптора в этой таблице.

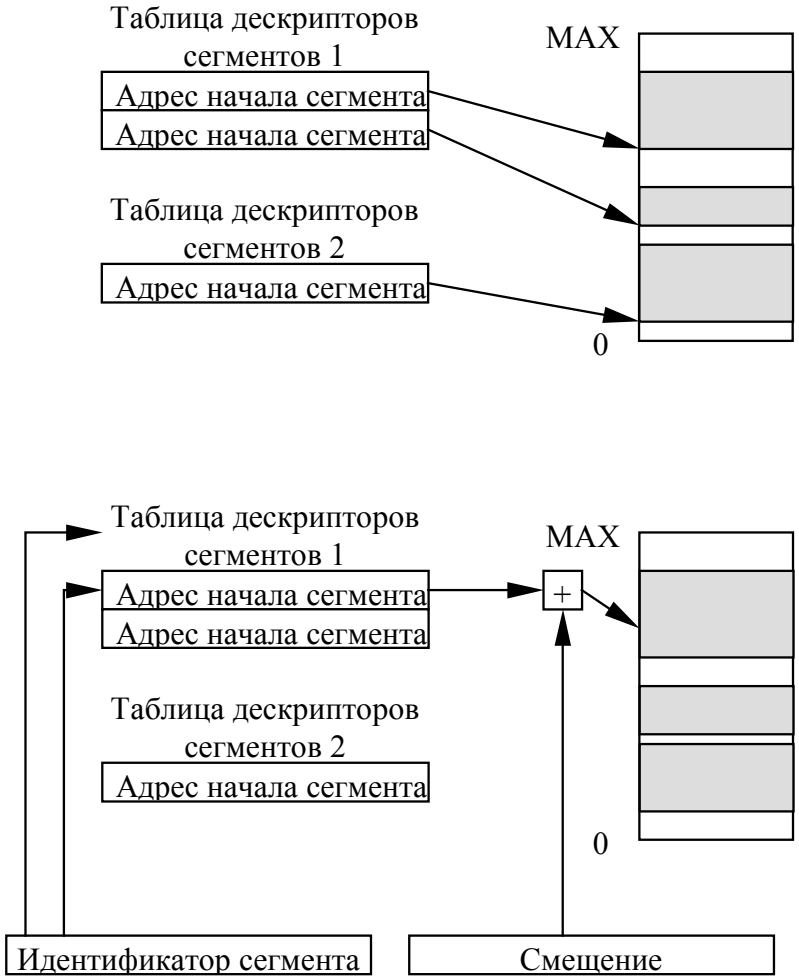


Рис. 28 Таблица дескрипторов сегментов и формирование адреса

При дешифрации команды и формировании физического адреса проводятся проверки на соответствие смещения и заявленной длины сегмента, выполняемой операции и прав доступа к сегменту, текущего уровня привилегированности и указанного в атрибутах сегмента.

Механизм сегментной адресации предоставляет возможность выделять блоки одинаковой целевой направленности и производить управление ими. Однако для решения задач управления необходимо описывать все адресное пространство. Поскольку описать каждый байт невозможно, используется разделение адресного пространства на блоки одинакового размера – страницы. Операционная система может хранить описание каждой страницы физической памяти в виде набора атрибутов (владелец страницы, атрибуты доступа к странице, используется ли страница в операции ввода-вывода и т.д.)

Размер страницы в общем случае может быть выбран произвольно, однако большое число современных архитектур поддерживают страничную организацию памяти, о которой мы скажем ниже - в этом случае размер страницы выбирается в равным размеру аппаратной страницы.

Способ описания физической памяти

Для управления физической памятью операционная система хранит ее описание в виде некоторой структуры данных. Перечислим несколько примеров.

- Массивы. Описание всего адресного пространства может храниться в виде массива описателей страниц. В этом случае легко проводится определение по физическому адресу номера страницы и соответствующего ей элемента массива и наоборот. В случае наличия нескольких адресных пространств может потребоваться несколько массивов.

- Для решения конкретных задач управления могут быть использованы битовые массивы (один бит на одну страницу), например, для учета используемых и свободных блоков. Данный способ имеет существенный недостаток – поиск серии заданной длины в битовом массиве является длительной операцией.

- Связные списки. Например, можно поддерживать список занятых и свободных фрагментов памяти. Каждая запись в списке в этом случае указывает, является ли область памяти свободной или занятой, если она занята процессом – идентификатор процесса, адрес начала области, размер области.

- Деревья. Для организации быстрого поиска по какому-либо количественному признаку может потребоваться организация дерева. Например, для осуществления поиска страницы, которая дольше всего не выделялась процессам.

В одной системе могут одновременно храниться несколько структур данных, используемые для решения различных задач управления памятью. Они могут храниться и обрабатываться независимо или совместно. Например, описатель страницы может иметь дополнительный атрибут ссылочного типа (номер следующей страницы/номер предыдущей страницы), с помощью которого все страницы будут объединены в список.

Виртуальное адресное пространство прикладной программы (ВАП) и способ его отображения на физическую память

Каждому процессу предоставляется виртуальное адресное пространство. Варианты организации ВАП полностью определяются возможностями аппаратной архитектуры. Рассмотрим несколько типичных случаев.

В качестве виртуального адресного пространства выступает адресное пространство физической памяти

В этом случае процесс может использовать все физическое адресное пространство системы. В своей работе он использует физические адреса и его возможность выполнять операции чтения и записи ограничивается только архитектурными ограничениями участка адресного пространства, с которым он работает (например, обычно нельзя писать в ПЗУ).

Естественным недостатком такой организации является неизолированность адресных пространств различных процессов и адресного пространства ядра. Любой процесс может получить доступ к данным другого процесса или операционной системы.

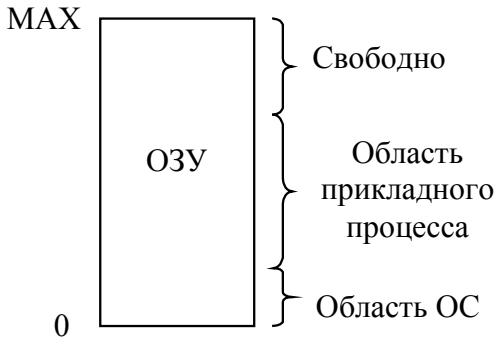


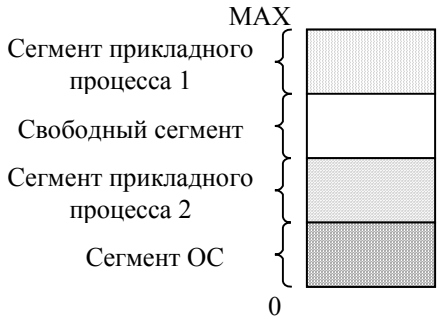
Рис. 29 ВАП процесса совпадает с физической памятью

В данном случае операционная система должна отслеживать только занятость областей памяти.

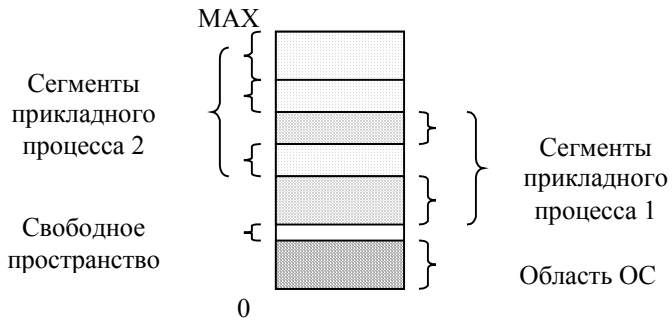
В качестве виртуального адресного пространства выступает множество сегментов

При создании процесса создается множество сегментов (или один сегмент) для хранения его кода, данных и стека.

Если архитектура поддерживает сегментную адресацию, адресное пространство каждого процесса окажется изолированным от пространств других процессов. При адресации в программе используются смещения внутри сегментов, а при исполнении осуществляется сегментное преобразование адресов. В зависимости от архитектуры, могут поддерживаться запросы процессов на выделение дополнительного сегмента или увеличение существующего.



ВАП процесса состоит из одного сегмента фиксированного размера



ВАП процесса состоит из нескольких сегментов различного размера

Рис. 30 ВАП процессов, организованное на основе сегментной адресации

Адресные пространства процессов располагаются в различных сегментах, и если существует механизм ограничения доступа к дескрипторам сегментов, не принадлежащих процессу, или ограничение доступа к самим сегментам, то адресные пространства процессов являются полностью изолированными друг от друга.

В данном случае операционная система должна отслеживать занятость областей памяти, множества сегментов, принадлежащих каждому процессу, обеспечивать разделение адресных пространств (например, поддерживать для каждого процесса собственную таблицу дескрипторов сегментов). Благодаря использованию сегментной адресации становится возможна организация выгрузки сегментов на более медленный носитель информации, например, на диск – при обратном перемещении сегмента в оперативную память достаточно будет изменить значение атрибута «физический адрес начала» в дескрипторе сегмента.

Виртуальное адресное пространство организуется на основе страничной организации памяти

Для каждого процесса создается собственное виртуальное адресное пространство с адресами от 0 до MAX, где MAX зависит от аппаратной архитектуры, но обычно достаточно велико (например, 4Гб). Код процесса оперирует с адресами в рамках своего ВАП. Обычно в ВАП процесса выделяется пространство доступных и недоступных ему адресов (например, 3Гб-1Гб или 2Гб-2Гб). В недоступных прикладному процессу адресах содержится область кода и данных операционной системы.

Пространство виртуальных адресов разделено на блоки одинакового размера, называемые страницами, совпадающие по размеру со страницами физической памяти. Когда программа пытается получить доступ по некоторому адресу, определяется номер страницы, в которой содержится адрес, определяется местоположение страницы в физической памяти, производится расчет физического адреса и осуществляется доступ.

Для отображения виртуальных адресов на физические используется **таблица страниц**, содержащая описания страниц виртуальной памяти процесса. Запись таблицы обычно имеет следующие атрибуты.

- адрес страницы в оперативной памяти или на внешнем носителе;
- признак присутствия в оперативной памяти;
- признак отображения страницы на физическую память;
- признак изменения содержимого страницы;
- признак обращения к странице;
- признак блокирования страницы в оперативной памяти;
- тип информации, размещенной в странице (код, данные, стек и др.);
- уровень привилегированности;
- доступные операции над информацией страницы (чтение, запись, выполнение и др.);
- другие атрибуты.

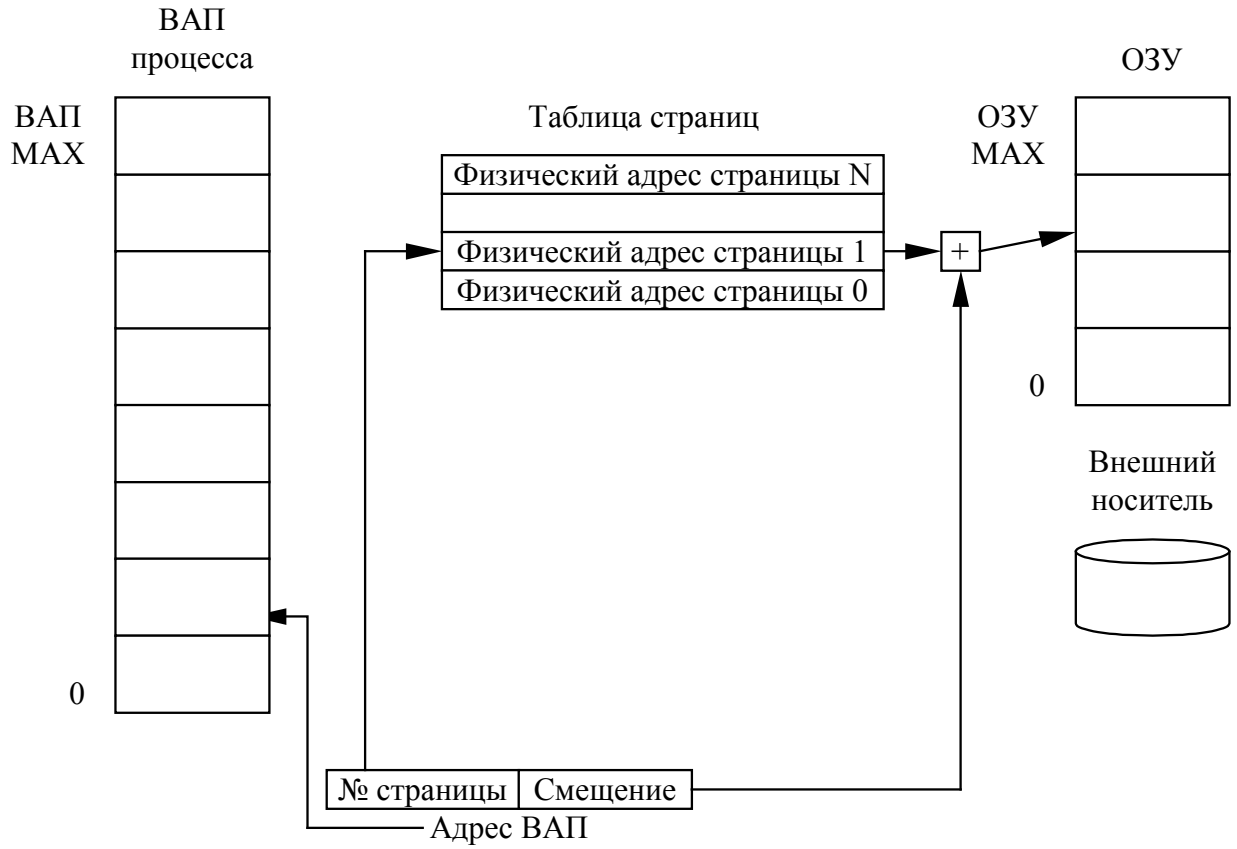


Рис. 31 Связь между виртуальными и физическими адресами через таблицу страниц

Страница ВАП может быть отображена как на оперативную память, так и на внешний носитель. Если программа воспользуется адресом со страницы, не отображенной в оперативную память, будет инициировано прерывание центрального процессора, передающее управление операционной системе. Такое прерывание называется **ошибкой из-за отсутствия страницы** или **ошибкой присутствия в памяти**, **страничным прерыванием** или **страничным сбоем**. Операционная система выбирает малоиспользуемый страничный блок, записывает его на внешний носитель, на освободившееся место считывает страницу, на которую произошла ссылка, изменяет таблицу страниц и запускает заново прерванную команду. Исключение составляет случай, когда программа попыталась выполнить операцию над неиспользуемым участком виртуального адресного пространства (который не отображен ни на оперативную память, ни на диск). Реакция на такую ситуацию зависит от операционной системы, но в большинстве случаев программа, выполнившая подобную операцию, будет аварийно завершена.

Несмотря на достаточно простое описание, при реализации встает важная проблема – таблица страниц может быть слишком большой. Современные компьютеры используют по крайней мере 32-х разрядные виртуальные адреса. При размере страницы в 4Кб, ВАП процесса будет состоять из одного миллиона страниц, и соответственно, таблица страниц будет содержать миллион записей! И таких таблиц страниц должно быть столько, сколько в каждый момент работает процессов.

Для решения этой проблемы во многих архитектурах используется многоуровневая таблица страниц.

На рисунке приведен пример двухуровневой таблицы страниц. При этом адрес ВАП делится на три поля – индекс таблицы таблиц страниц, индекс таблицы страниц, смещение в странице.

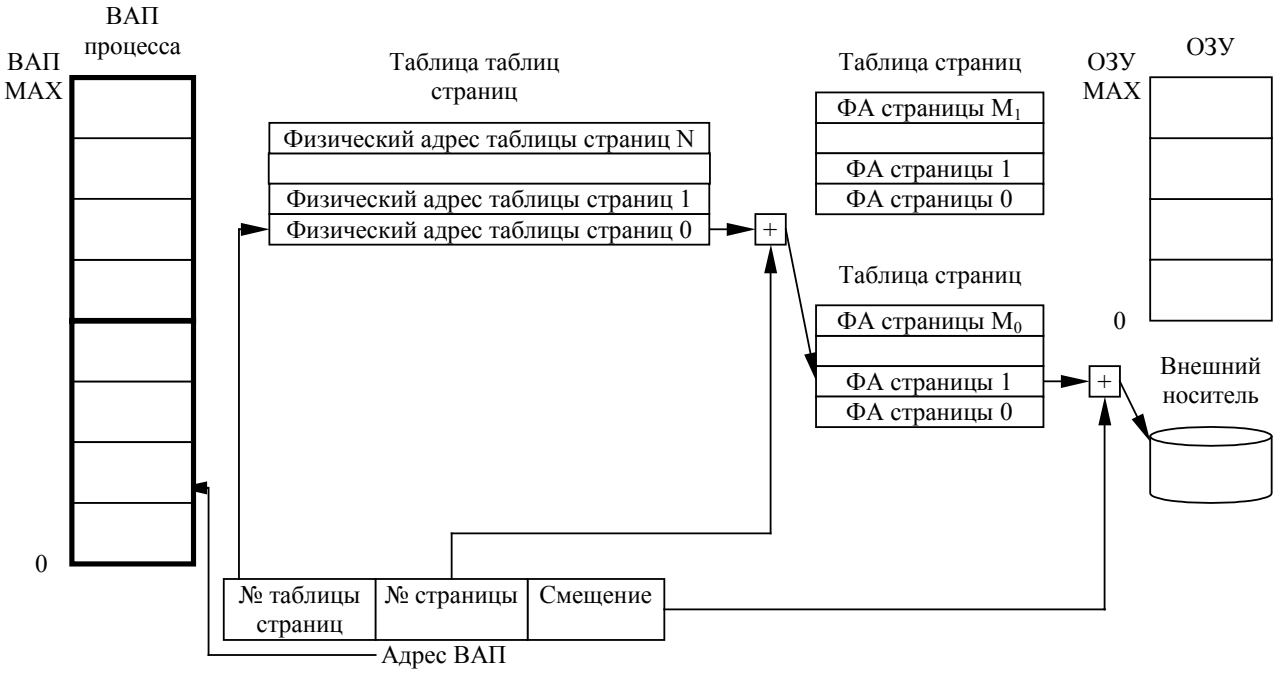


Рис. 32 Использование двух уровней таблиц страниц

Таблица таблиц страниц необходима для организации ВАП, но благодаря тому, что большинство программ используют лишь небольшую часть ВАП, необходимо создавать таблицы страниц только для тех участков адресов, в которых действительно располагаются код или данные программы.

Еще одним важным свойством такого подхода является возможность использования несколькими процессами общих таблиц страниц, что приводит к созданию для таких процессов общей памяти.

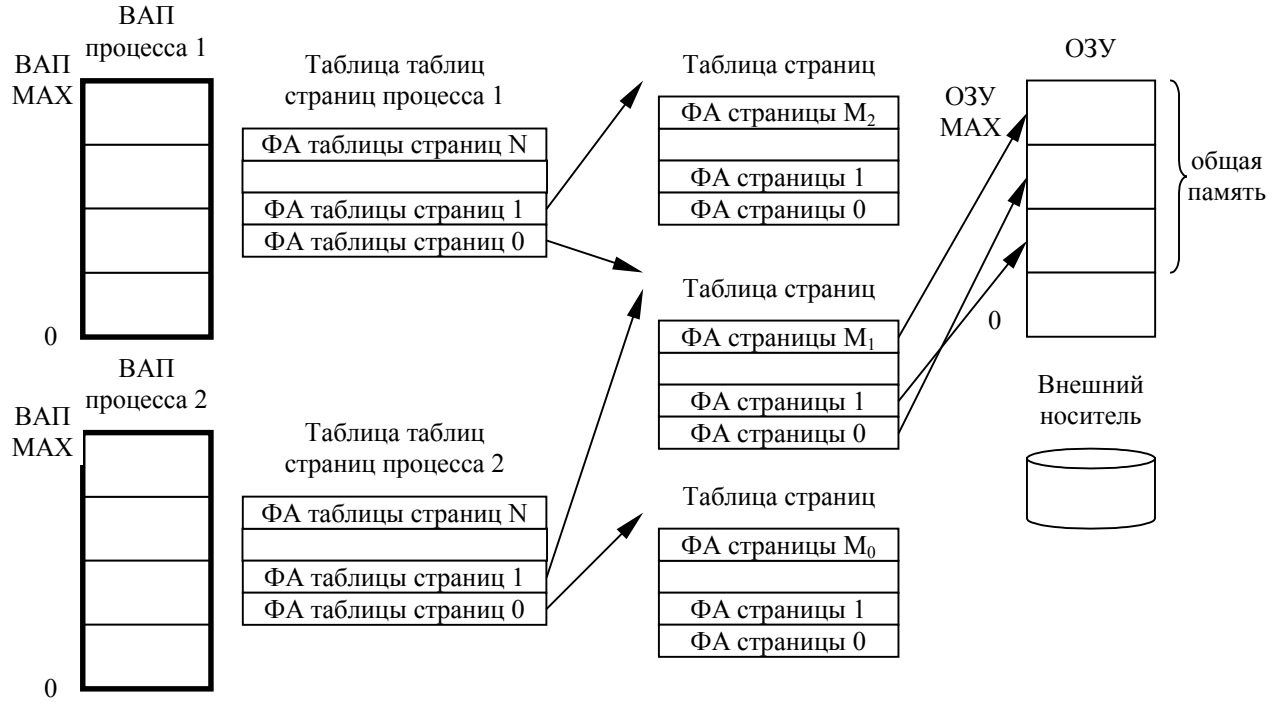


Рис. 33 Организация разделяемой памяти для нескольких процессов

Алгоритм связывания адресов программных модулей с адресами ВАП

На пути к выполнению программа обычно проходит нескольких шагов:

- написание текста на некотором языке программирования;
- компиляция текста в объектный модуль;
- сборка объектных модулей в загрузочный модуль;
- создание из загрузочного модуля бинарного образа в памяти.

На каждом шаге используемые программой адреса могут быть представлены различными способами. Например, адреса в исходных текстах обычно символические. Компилятор связывает эти символические адреса с перемещаемыми адресами (такими как n байт от начала модуля или сегмента). Линкер, в свою очередь, связывают эти перемещаемые адреса с адресами виртуального адресного пространства. Загрузчик осуществляет подстройку адресов под параметры ВАП, созданного для процесса. Каждое связывание - отображение одного адресного пространства в другое.

Привязка инструкций и данных к памяти в принципе может быть, таким образом, сделана на следующих шагах:

- Этап компиляции (Compile time). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда генерируются абсолютные адреса. Если стартовый адрес программы меняется, необходимо перекомпилировать код. В качестве примера можно привести .com программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.

- Этап загрузки (Load time). Если на стадии компиляции не известно где процесс будет размещен в памяти, компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.

- Этап выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одного адресуемого блока памяти в другой, связывание откладывается до времени выполнения. Здесь желательно специализированное оборудование, например регистры перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Например, в MS-DOS для этой цели используются четыре сегментных регистра.

Алгоритм обеспечения пространственного мультиплексирования

Под алгоритмом обеспечения пространственного мультиплексирования мы имеем в виду совокупность алгоритмов поддержки адресного пространства процессов, обслуживания запросов процессов на выделение им памяти, разрешения коллизий между процессами при конкуренции за оперативную память, защиты адресных пространств процессов и организации выгрузки блоков ВАП процессов на диск.

В каждой конкретной архитектуре используются различные сочетания принципов организации адресного пространства, рассмотренных выше, поэтому все множество возможных алгоритмов рассмотреть невозможно. Мы рассмотрим несколько частных случаев архитектур и алгоритмы, применимые для них.

Схема с фиксированными разделами

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько

разделов фиксированной величины. По мере прибытия процесс помещается в тот или иной раздел.

Как правило, происходит условное разбиение физического адресного пространства. Связывание логических адресов процесса и физических происходит на этапе его загрузки в конкретный раздел. Каждый раздел может иметь свою очередь или может существовать глобальная очередь для всех разделов. Когда поступает задание, оно встает в общую очередь или в очередь к одному из подходящих для нее разделов.

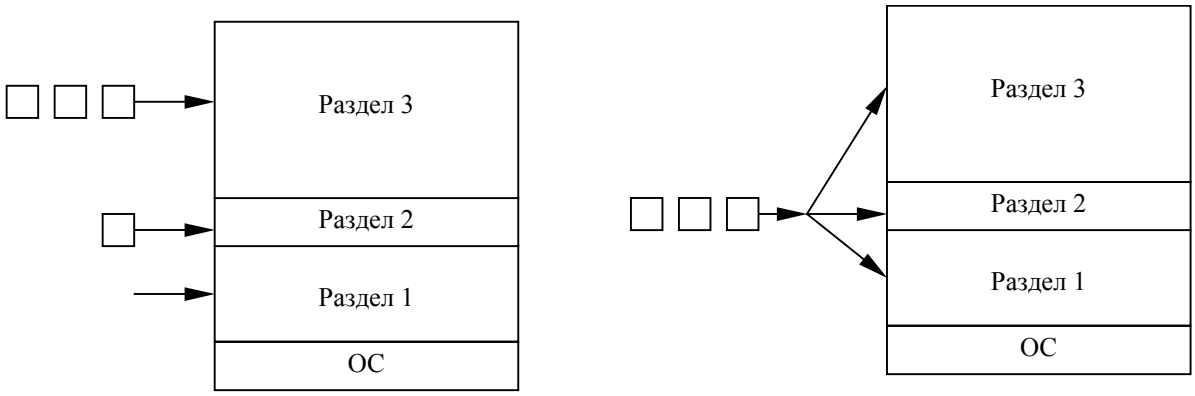


Рис. 34 Схема с фиксированными разделами: с отдельными очередями процессов, с общей очередью процессов

Подсистема управления памятью сравнивает размер программы, поступившей на выполнение, выбирает подходящий раздел, осуществляет загрузку программы и настройку адресов. В какой раздел помещать программу? Распространены три стратегии:

- Стратегия первого подходящего (First fit). Задание помещается в первый подходящий по размеру раздел.

- Стратегия наиболее подходящего (Best fit). Задание помещается в тот раздел, где ему наиболее тесно.

- Стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что с точки зрения утилизации памяти и уменьшения времени первые два способа лучше. С точки зрения утилизации первые два примерно одинаковы, но первый способ быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например, для размещения файлов на диске.

Связывание (настройка) адресов для данной схемы возможны как на этапе компиляции, так и на этапе загрузки.

Очевидный недостаток этой схемы - число одновременно выполняемых процессов ограничено числом разделов. Другим существенным недостатком является то, что предлагаемая схема сильно страдает от фрагментации свободной памяти, которая не может быть использована ни одним процессом. Внешняя фрагментация возникает, потому что процесс не полностью занимает выделенный ему раздел. Также возможна ситуация, когда имеются свободные разделы, но они не могут быть использованы, поскольку слишком малы для программ, поставленных в очередь на исполнение.

Частным случаем схемы с фиксированными разделами является работа менеджера памяти однозадачной ОС.

В памяти размещается один пользовательский процесс. Остается определить, где располагается пользовательская программа по отношению к ОС - сверху, снизу или

посередине. Причем часть ОС может быть в ПЗУ (например, BIOS, драйверы устройств). Главный фактор, влияющий на это решение - расположение вектора прерываний, который обычно локализован в нижней части памяти, поэтому ОС также размещают в нижней. Примером такой организации может служить ОС MS-DOS. Чтобы пользовательская программа не портила кода ОС, требуется защита ОС, которая может быть организована при помощи одного граничного регистра, содержащего адрес границы ОС.

Оверлейная структура

Так как размер виртуального адресного пространства процесса может быть больше чем размер выделенного ему раздела (или больше чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay) или организация структуры с перекрытием. Основная идея - держать в памяти только те инструкции программы, которые нужны в данный момент времени. Потребность в таком способе загрузки появляется, если программа относительно велика.

Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для конструирования оверлеев необходимы специальные алгоритмы перемещения и связывания. Для описания оверлейной структуры обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом, описывающим дерево вызовов внутри программы. Синтаксис подобного файла может распознаваться загрузчиком. Привязка к памяти происходит в момент очередной загрузки одной из ветвей программы.

Оверлеи не требуют специальной поддержки со стороны ОС. Они могут быть полностью реализованы на пользовательском уровне с простой файловой структурой. ОС лишь делает несколько больше операций ввода-вывода. Типовое решение - порождение линкером специальных команды, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

Программист должен тщательно проектировать оверлейную структуру. Это требует полного знания структуры программы, кода, данных, языка описания оверлейной структуры. По этой причине применение оверлеев ограничено компьютерами с ограничением на память. Как мы увидим в дальнейшем, проблема оверлейных сегментов, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Свопинг

Рассматриваемая схема допускает организацию выгрузки разделов на диск в случае необходимости.

Имея дело с пакетными системами можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping) - перемещению процессов из главной памяти на диск и обратно целиком. Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти. Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. В системах со свопингом время переключения контекстов лимитируется временем загрузки-выгрузки процессов. Для эффективной утилизации процессора необходимо, чтобы величина кванта времени существенно его превышала. Оптимизация свопинга может быть связана с выгрузкой лишь реально

используемой памяти или выгрузкой процессов, реально не функционирующих. Кроме того, выгрузка обычно осуществляется в специально отведенное пространство для свопинга, то есть быстрее, чем через стандартный интерфейс файловой системы (пространство выделяется большими блоками, поиск файлов и методы непосредственного выделения не используются).

Схема с переменными разделами

Более эффективной представляется схема с переменными (динамическими) разделами. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется необходимая память. При завершении процесса или выгрузке его на диск в ходе свопинга память временно освобождается. По истечении некоторого времени память представляет собой набор занятых и свободных участков. Смежные свободные участки могут быть объединены в один.

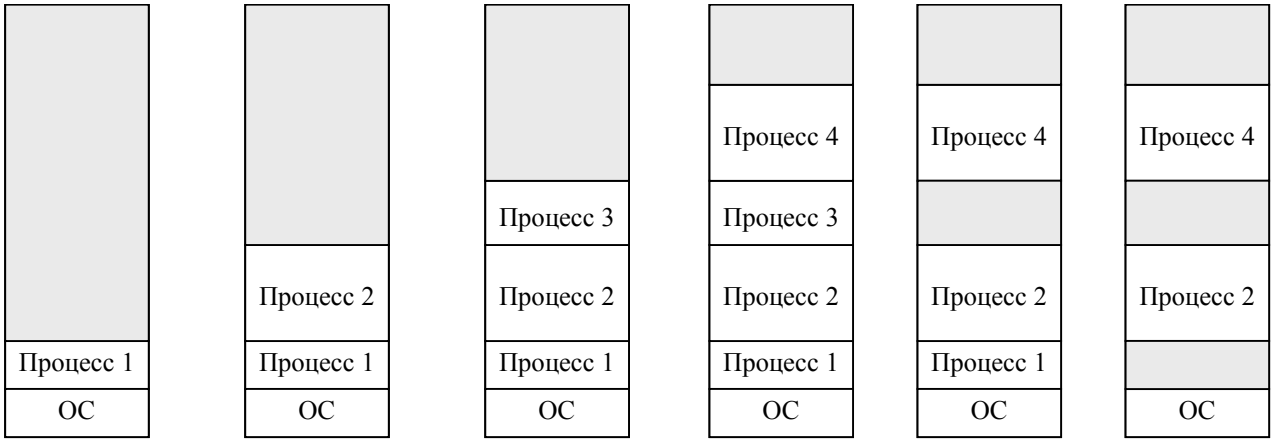


Рис. 35 Динамика распределения памяти между процессами

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборке его среди имеющихся в соответствие с одной из стратегий (first-fit, best-fit, worst-fit), загрузке процесса в выбранный раздел и последующем внесении изменений в таблицы свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может быть осуществлено на этапах загрузки и выполнения.

Этот метод более гибок по сравнению с методом фиксированных разделов. Ему также присуща внешняя фрагментация вследствие наличия большого числа участков свободной памяти. Проблемы фрагментации могут быть различными. В худшем случае мы можем иметь участок свободной (потерянной) памяти между двумя процессами. Если все эти куски объединить в один блок, мы смогли бы разместить больше процессов. Выбор между first-fit и best-fit слабо влияет на величину фрагментации. В зависимости от суммарного размера памяти и среднего размера процесса эта проблема может быть большей или меньшей. Статистический анализ показывает, что при наличии n блоков пропадает n/2 блоков, то есть 1/3 памяти! Это известное 50% правило (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены в один).

Одно из решений проблемы внешней фрагментации - разрешить адресному пространству процесса не быть непрерывным, что позволяет выделять процессу память в любых доступных местах. Один из способов реализации такого решения – организация ВАП на основе страничного преобразования, используемый во многих современных ОС (будет рассмотрен ниже).

Другим способом борьбы с внешней фрагментацией является сжатие, то есть перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память образовала непрерывную область. Этот метод иногда называют схемой с перемещаемыми разделами. В идеале фрагментация после сжатия должна отсутствовать.

Сжатие, однако, является дорогостоящей процедурой, алгоритм выбора оптимальной стратегии сжатия очень труден, и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

Схема с использованием виртуального адресного пространства на основе страничного преобразования

При помощи виртуальной памяти обычно решают две задачи. Во-первых, виртуальная память позволяет адресовать пространство, гораздо большее, чем емкость физической памяти конкретной вычислительной машины. Во-вторых, обычно для реальных программ действует принцип локальности (в течение какого-то отрезка времени ограниченный фрагмент кода работает с ограниченным набором данных), и нет необходимости в помещении их в физическую память целиком.

Возможность выполнения программы, находящейся в памяти лишь частично имеет ряд вполне очевидных преимуществ.

- Программа не ограничена величиной физической памяти.

- Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о фактическом размере используемой памяти.

- Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы.

- Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге, каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы видимости практически неограниченной (32- или 64-разрядной) адресуемой пользовательской памяти при наличии основной памяти существенно меньших размеров - очень важный аспект. Но введение виртуальной памяти позволяет решать другую не менее важную задачу: обеспечение контроля доступа к отдельным блокам памяти и в частности защиту пользовательских программ друг от друга и защиту ОС от пользовательских программ.

Напомним, что в системах с виртуальной памятью те адреса, которые генерирует программа, называются виртуальными, и они формируют виртуальное адресное пространство, которое посредством страничного преобразования с использованием таблиц страниц отображается на физические устройства хранения информации. Дополнительно отметим, что внутри ВАП может быть организовано как линейная схема адресации, так и сегментная. В последнем случае приложение использует адреса в виде пары (идентификатор/селектор сегмента, смещение), которые сначала проходят через сегментное преобразование, формируя линейный адрес ВАП, который затем проходит через страничное преобразование, формируя физический адрес. Сегментно-страничная организация виртуальной памяти позволяет совместно использовать одни и те же сегменты данных и программного кода в виртуальной памяти разных задач (для каждой виртуальной памяти создается отдельная таблица сегментов, но для совместно используемых сегментов поддерживаются общие таблицы страниц).

Общие вопросы управления страничной памятью

Обычно рассматривают три стратегии управления страничной памятью.

- Стратегия выборки (fetch policy) - в какой момент следует переписать страницу из вторичной памяти в первичную (с жесткого диска в оперативную память). Выборка бывает по запросу и с упреждением. Алгоритм выборки вступает в действие в тот момент, когда процесс обращается к не присутствующей странице, содержимое которой в данный момент находится на диске (в своп файле или файле, отображенном в память), и потому является ключевым. Он обычно заключается в загрузке страницы с диска в свободную физическую страницу и отображении этой физической страницы в то место виртуального адресного пространства, куда было произведено обращение, вызвавшее исключительную ситуацию. Существует модификация алгоритма выборки, которая применяет еще и опережающее чтение (с упреждением), т.е. кроме страницы, вызвавшей исключительную ситуацию, в память также загружается несколько страниц, окружающих ее (так называемый кластер). Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникающих при работе с большими объемами данных или кода, кроме того, оптимизируется и работа с диском, поскольку появляется возможность загрузки нескольких страниц за одно обращение к диску.

- Стратегия размещения (placement policy) - определить в какое место первичной памяти поместить поступающую страницу. В системах со страничной организацией ее можно поместить в любой свободный страничный кадр (в системах с сегментной организацией - нужна стратегия, аналогичная стратегии с переменными разделами). Исключение составляют случаи неоднородной архитектуры памяти.

- Стратегия замещения (replacement policy) - какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место. Разумная стратегия замещения позволяет оптимизировать хранение в памяти самой необходимой информации и тем самым снизить частоту страничных сбоев.

Наиболее ответственным действием страничной системы является выделение страницы основной (оперативной) памяти для удовлетворения требования доступа к отсутствующей в основной памяти виртуальной странице. Напомним, что мы рассматриваем ситуацию, когда размер ВАП каждого процесса может существенно превосходить размер основной памяти. Это означает, что при выделении страницы оперативной памяти с большой вероятностью не удастся найти свободную (не приписанную к виртуальному адресному пространству какого-либо процесса) страницу. В этом случае операционная система должна в соответствии с заложенными в нее критериями найти некоторую занятую страницу оперативной памяти, переместить в случае надобности ее содержимое во внешнюю память, загрузить требуемую страницу из внешней памяти в оперативную, должным образом модифицировать соответствующие элементы соответствующих таблиц страниц и после этого продолжить процесс удовлетворения доступа к странице.

Заметим, что при замещении приходится дважды передавать страницу между основной и вторичной памятью. Процесс замещения может быть оптимизирован за счет использования признака изменения (один из атрибутов страницы). Признак изменения устанавливается компьютером, если хотя бы один байт записан на страницу. При выборе кандидата на замещение, проверяется признак изменения. Если он не установлен, нет необходимости записывать данную страницу на диск, она уже там присутствует. То же относится и к страницам, доступным только для чтения - они никогда не модифицируются. Эта схема уменьшает время обработки ошибки присутствия в памяти (page fault’a).

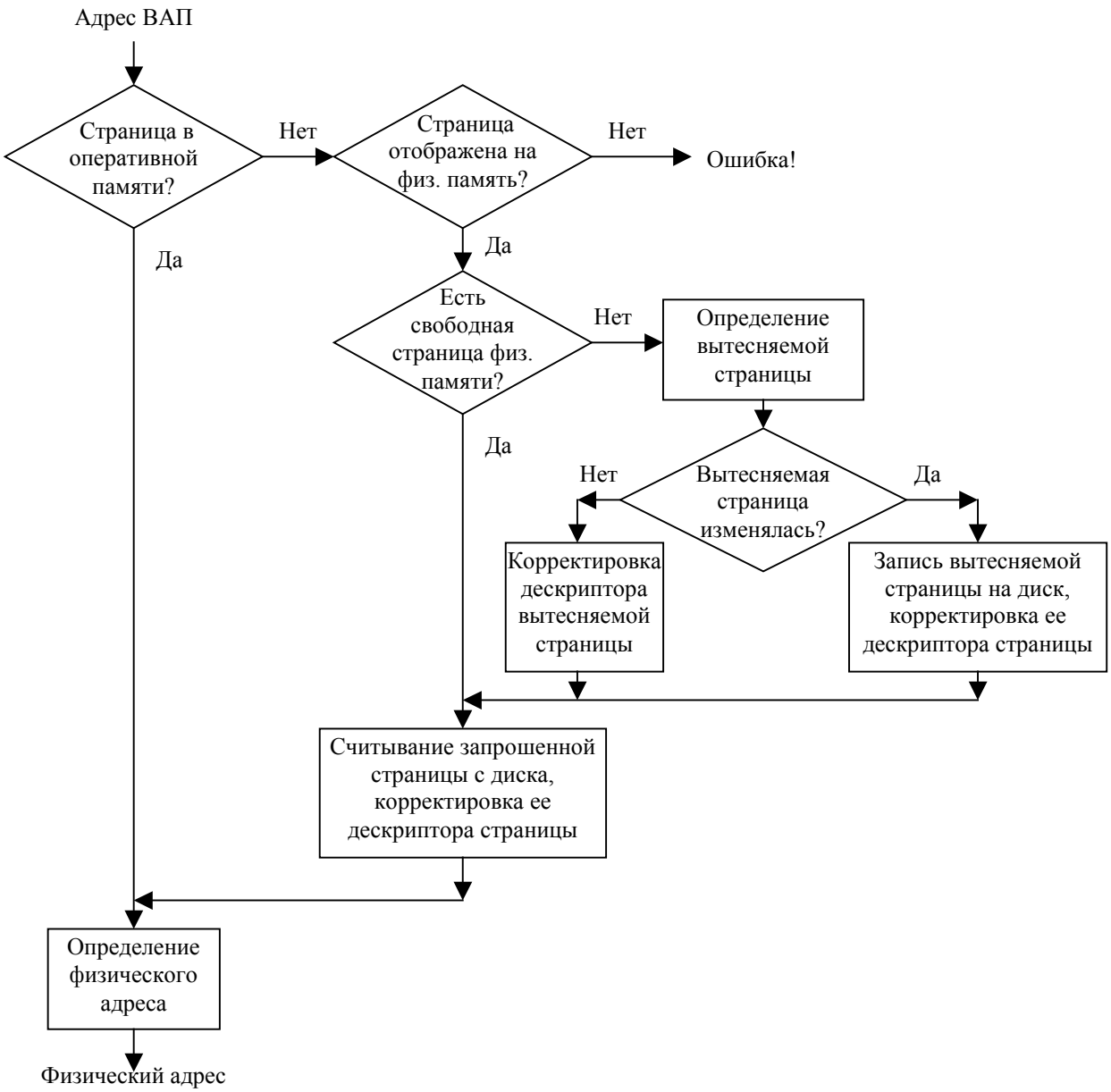


Рис. 36 Блок-схема обработчика страничного сбоя

Существует большое количество разнообразных алгоритмов замещения страниц. Все они делятся на локальные и глобальные. Локальные алгоритмы, в отличие от глобальных, распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процесс израсходует все предназначенные ему страницы, система будет удалять из физической памяти одну из его страниц, а не из страниц других процессов. Глобальный же алгоритм замещения в случае возникновения исключительной ситуации удовлетворится освобождением любой физической страницы, независимо от того, какому процессу она принадлежала.

Глобальные алгоритмы имеют несколько недостатков. Во-первых, они делают одни процессы чувствительными к поведению других процессов. Например, если один процесс в системе использует большое количество памяти, то все остальные приложения будут в результате ощущать сильное замедление из-за недостатка памяти. Во-вторых, некорректно работающее приложение может подорвать работу всей системы (если конечно в системе не предусмотрено ограничение на размер памяти, выделяемой процессу), пытаясь захватить все

больше памяти. Поэтому в многозадачной системе лучше использовать более сложные, но эффективные локальные алгоритмы. Такой подход требует, чтобы система хранила список физических страниц каждого процесса. Этот список страниц иногда называют рабочим множеством процесса.

Алгоритм обычно оценивается на конкретной последовательности ссылок к памяти, для которой подсчитывается число ошибок присутствия в памяти. Эта последовательность называется reference string. Мы можем генерировать reference string искусственным образом при помощи датчика случайных чисел или трассируя конкретную систему. Последний метод дает слишком много ссылок, для уменьшения числа которых можно сделать две вещи:

- для конкретного размера страниц можно запоминать только их номера, а не адреса, на которые идет ссылка;

- если имеется ссылка на страницу, ближайшие последующие ссылки на данную страницу можно не фиксировать.

Как уже говорилось, большинство процессоров имеют простейшие аппаратные средства, позволяющие собирать некоторую статистику обращений к памяти. Эти средства включают два специальных флага на каждый элемент таблицы страниц. Один флаг (флаг использования или обращения, reference бит) автоматически устанавливается, когда происходит любое обращение к этой странице, а второй флаг (флаг модификации или изменения, modify бит) устанавливается, если производится запись в эту страницу. Чтобы использовать эти возможности, операционная система должна периодически сбрасывать эти флаги.

При изучении алгоритмов обратите внимание, что при реализации любого из них в качестве базовой структуры данных можно использовать очередь описателей страниц. Естественно, методы управления очередью в точках принятия решений будут различны.

Алгоритмы замещения страниц

Оптимальный алгоритм

Этот алгоритм имеет минимальную частоту ошибок присутствия в памяти среди всех алгоритмов. Он прост: замещай страницу, которая не будет использоваться в течение наибольшего периода времени. Каждая страница помечается числом инструкций, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка.

Этот алгоритм нереализуем: ОС не знает, к какой странице будет следующее обращение. Зато из этого можно сделать вывод, что для того, чтобы алгоритм замещения был максимально близок к идеальному алгоритму, система должна как можно точнее предсказывать будущие обращения процессов к памяти. Данный алгоритм применяется для оценки качества реализуемых алгоритмов.

Алгоритм FIFO - Выталкивание первой пришедшей страницы

Простейший алгоритм. Каждой странице присваивается временная метка. Реализуется это просто созданием очереди страниц, в конец которой страницы попадают, когда загружаются в физическую память, а из начала берутся, когда требуется освободить память. Для замещения выбирается старейшая страница. К сожалению, эта стратегия с достаточной вероятностью будет приводить к замещению активно используемых страниц, например, страниц разделяемых библиотек. Заметим, что при замещении активных страниц все работает корректно, но ошибка присутствия в памяти происходит немедленно.

Алгоритм LRU (The Least Recently Used) - Выталкивание дольше всего не использовавшейся страницы

Исходим из эвристического правила, что недавнее прошлое - хороший ориентир для прогнозирования ближайшего будущего. Ключевое отличие между FIFO и оптимальным алгоритмом в том, что один смотрит назад, а другой вперед. Если использовать прошлое, для аппроксимации будущего, имеет смысл замещать страницу, которая не использовалась в течение долгого времени. Такой подход называется least recently used (LRU) алгоритм.

LRU часто используется и считается хорошим. Основная проблема - реализация. Необходимо иметь связанный список всех страниц в памяти, в начале которого будут часто используемые страницы. Причем он должен обновляться при каждой ссылке, при этом много времени потребуется на поиск страниц в списке. Есть вариант реализации со специальным устройством. Например, если имеется 64-битный указатель, который автоматически увеличивается на 1 после каждой инструкции и в таблице страниц имеется соответствующее поле, в которое заносится значение указателя при каждой ссылке на страницу, то при возникновении ошибки присутствия в памяти выгружается страница с наименьшим указателем.

Существует класс алгоритмов, называемых стековыми. Это алгоритмы, для которых множество страниц в памяти для ОЗУ, объемом n страниц - всегда подмножество страниц для случая ОЗУ из n+1 страницы. LRU таковым является.

Заметим, что никакая реализация LRU неприемлема без специального оборудования. Если, например, задействовать прерывание для модификации полей, то это будет замедлять ссылку к памяти в несколько раз.

Алгоритм NFU (Not Frequently Used) - Выталкивание редко используемой страницы

Программная реализация алгоритма, близкого к LRU. Рассмотренные варианты LRU в принципе реализуемы, но, как уже отмечалось, они требуют специальной аппаратной поддержки, которой большинство современных процессоров не предоставляет. Поэтому хотелось бы иметь алгоритм, достаточно близкий к LRU, но не требующий сложной специальной поддержки. Один из таких возможных алгоритмов - это алгоритм NFU.

Для него требуются программные счетчики, по одному на каждую страницу, которые сначала равны нулю. При каждом прерывании по времени (а не после каждой инструкции) операционная система сканирует все страницы в памяти и у каждой страницы с установленным флагом обращения увеличивает на единицу значение счетчика, а флаг обращения сбрасывает.

Таким образом, кандидатом на освобождение оказывается страница с наименьшим значением счетчика, как страница, к которой реже всего обращались. Главным недостатком алгоритма NFU является то, что он никогда ничего не забывает. Например, страница, к которой очень много обращались некоторое время, а потом обращаться перестали, все равно не будет удалена из памяти, потому что ее счетчик содержит большую величину.

Для устранения данного недостатка используется небольшая модификация алгоритма, которая реализует "забывание". Достаточно, чтобы периодически (например, раз в секунду) содержимое каждого счетчика сдвигалось вправо на 1 бит, а уже затем производилось бы его увеличение для страниц с установленным флагом обращения.

Другим, уже не так просто устранимым недостатком алгоритма, является длительность процесса сканирования таблиц страниц.

Алгоритм Second-Chance - Вторая попытка

Модификация алгоритма FIFO, которая позволяет избежать потери часто используемых страниц посредством анализа признака использования для самой старой страницы. Если признак установлен, то страница, в отличие от FIFO, не выталкивается, а очищается бит и страница становится в конец очереди. Если ко всем страницам было осуществлено обращение, он превращается в FIFO.

Алгоритм «часы»

Хотя алгоритм Second-Chance является корректным, он неэффективен, потому что постоянно передвигает страницы по списку. Поэтому лучше хранить описания страничных блоков в виде кольцевого списка и использовать указатель на старейшую страницу.

Дальнейшим развитием алгоритма является использование двух указателей перемещаемых по кольцевому списку. Первый указывает на страницу, дольше всего находящуюся в памяти. Второй – перемещается по списку синхронно с первым и находится всегда на одном расстоянии до/после него; при передвижении второго указателя на следующую страницу, для нее очищается признак использования. Таким образом, интервал времени, за который страница должна быть использована, чтобы ее не вытеснили, уменьшается, и в памяти остаются только действительно часто используемые страницы.

Заключение

Память представляет собой важный ресурс, требующий тщательного управления. Современные компьютеры часто поддерживают некоторую форму виртуальной памяти, соответственно менеджер памяти должен на базе аппаратных средств поддерживать виртуальные адресные пространства процессов, одновременно реализуя алгоритмы пространственного мультиплексирования адресных пространств различных процессов. Для хорошей работы системы необходим выбор наиболее подходящего для ее области использования алгоритма замещения страниц.

Взаимодействие потоков – передача данных и синхронизация

Даже в однопроцессорной многозадачной системе процессы чередуются (разделяют процессор), создавая иллюзию параллельного выполнения. В многопроцессорной системе наличие нескольких параллельно выполняющихся процессов (или потоков), работающих под управлением ОС, является необходимым условием ее функционирования.

Таким образом, концепция параллельных вычислений является одной из ключевых концепций построения современных ОС. Параллельность охватывает множество вопросов разработки, включая вопросы обмена информацией между процессами, разделения ресурсов, синхронизацию работы процессов.

Потоки могут выполняться параллельно (независимо друг от друга, в случае многопроцессорной системы) до тех пор, пока не возникнет потребность в общении между ними. Для обеспечения совместной работы потоков необходимо предпринимать усилия - специальным образом планировать и управлять ими. Во многих ОС присутствуют специальные средства, обеспечивающие их синхронизацию и общение и устанавливающие определенные ограничения на последовательность выполнения взаимосвязанных параллельных потоков. Реализация синхронизирующих правил осуществляется с помощью механизмов синхронизации. Такие механизмы весьма многочисленны по способам реализации, отличаются степенью эффективности и областями использования в различных приложениях.

Особенности каждого конкретного типа взаимодействия между двумя или более параллельными потоками определяются задачей синхронизации. Количество различных задач синхронизации неограниченно. Однако некоторые из них являются типичными. К ним относятся: взаимное исключение, производители-потребители, читатели-писатели, обедающие философы и т.д. Большинство задач в реальных ОС по согласованию параллельных потоков можно решить либо с помощью этих типовых задач, либо с помощью их модификаций.

Взаимодействие потоков

Если приложение реализовано в виде множества процессов (или потоков), то эти процессы (потоки) могут взаимодействовать двумя основными способами:

- посредством разделения памяти (оперативной или внешней)
- посредством передачи сообщений или потоков данных
- посредством использования одних и тех же объектов операционной системы

При взаимодействии через общую память процессы должны синхронизовать свое выполнение. Пренебрежение вопросами синхронизации в многопоточной системе может привести к неправильной работе приложения или даже к его краху. Рассмотрим, например, задачу ведения базы данных клиентов некоторого предприятия. Каждому клиенту отводится отдельная запись в базе данных, в которой среди прочих полей имеются поля Заказ и Оплата. Программа, ведущая базу данных, оформлена как единый процесс, имеющий несколько потоков, в том числе поток А (писатель 1), который заносит в базу данных информацию о заказах, поступивших от клиентов, и поток В (писатель 2), который фиксирует в базе данных сведения об оплате клиентами выставленных счетов. Оба эти потока совместно работают над общим файлом базы данных, используя однотипные алгоритмы, включающие три шага.

1. Считать из файла базы данных в буфер запись о клиенте с заданным идентификатором.

2. Внести новое значение в поле Заказ (для потока A) или Оплата (для потока B).

3. Вернуть модифицированную запись в файл базы данных.

Обозначим соответствующие шаги для потока A как Al, A2 и A3, а для потока B как Bl, B2 и B3. Предположим, что в некоторый момент поток A обновляет поле Заказ записи о клиенте N. Для этого он считывает эту запись в свой буфер (шаг A1), модифицирует значение поля Заказ (шаг A2), но внести запись в базу данных (шаг A3) не успевает, так как его выполнение прерывается, например, вследствие завершения кванта времени.

Предположим также, что потоку B также потребовалось внести сведения об оплате относительно того же клиента N. Когда подходит очередь потока B, он успевает считать запись в свой буфер (шаг B1) и выполнить обновление поля Оплата (шаг B2), а затем прерывается. Заметим, что в буфере у потока B находится запись о клиенте N, в которой поле Заказ имеет прежнее, не измененное значение.

Когда в очередной раз управление будет передано потоку A, то он, продолжая свою работу, запишет запись о клиенте N с модифицированным полем Заказ в базу данных (шаг A3). После прерывания потока A и активизации потока B последний запишет в базу данных поверх только что обновленной записи о клиенте N свой вариант записи, в которой обновлено значение поля Оплата. Таким образом, в базе данных будут зафиксированы сведения о том, что клиент N произвел оплату, но информация о его заказе окажется потерянной[1].

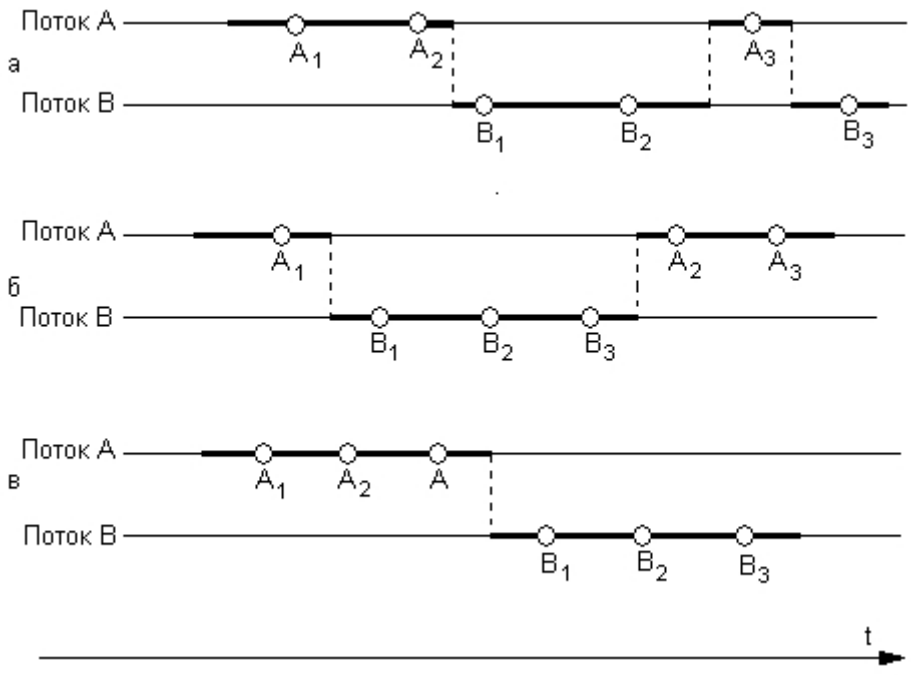


Рис. 37 Влияние относительных скоростей потоков на результат решения задачи

Сложность проблемы синхронизации кроется в нерегулярности возникающих ситуаций. Так, в предыдущем примере можно представить и другое развитие событий: могла быть потеряна

[1] Рассмотренный пример является типичным для приложений оперативной обработки данных (OLTP). В СУБД для решения рассмотренной проблемы используется так называемый механизм транзакций.

информация не о заказе, а об оплате или, напротив, все исправления были успешно внесены. Все определяется взаимными скоростями потоков и моментами их прерывания. Поэтому отладка взаимодействующих потоков является сложной задачей, поскольку ситуации, в которые "попадают" потоки могут различаться от запуска к запуску. Ситуации, подобные той, когда два или более потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков, называются **гонками**.

Критическая секция

Важным понятием синхронизации потоков является понятие «критической секции» программы. Критическая секция — это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда выполнение этой части еще не завершено. Критическая секция всегда определяется по отношению к определенным критическим данным, при несогласованном изменении которых могут возникнуть нежелательные эффекты. В предыдущем примере такими критическими данными являлись записи файла базы данных. Во всех потоках, работающих с критическими данными, должна быть определена критическая секция. Заметим, что в разных потоках критическая секция состоит в общем случае из разных последовательностей команд.

Чтобы исключить эффект гонок по отношению к критическим данным, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими данными, находился только один поток. При этом неважно, находится этот поток в активном или в приостановленном состоянии. Этот прием называют взаимным исключением.

Задача взаимного исключения

Задача взаимного исключения - фундаментальная по своему назначению задача. Любая ОС, управляющая параллельными процессами, должна обеспечить тот или иной вариант реализации ее решения. Суть задачи состоит в следующем: необходимо согласовать работу n>1 параллельных потоков при использовании некоторого критического ресурса таким образом, чтобы удовлетворить следующим требованиям:

- одновременно внутри критической секции должно находиться не более одного потока;

- критические секции не должны иметь приоритеты в отношении друг друга;

- остановка какого-либо процесса вне его критической секции не должна влиять на дальнейшую работу процессов по использованию критического ресурса;

- решение о вхождении потоков в их критические секции при одинаковом времени поступления запросов на такое вхождение и равноприоритетности потоков не откладывается на неопределенный срок, а является конечным во времени;

- относительные скорости развития потоков неизвестны и произвольны;

- любой поток может переходить в любое состояние, отличное от активного, вне пределов своей критической секции;

- освобождение критического ресурса и выход из критической секции должны быть произведены потоком, использующим критический ресурс, за конечное время.

Операционная система использует разные способы реализации взаимного исключения. Некоторые способы пригодны для взаимного исключения при вхождении в критическую секцию только потоков одного процесса, в то время как другие могут обеспечить взаимное исключение и для потоков разных процессов.

Самый простой и в то же время самый неэффективный способ обеспечения взаимного исключения состоит в том, что операционная система позволяет потоку запрещать любые прерывания на время его нахождения в критической секции. Однако этот способ практически не применяется, так как опасно доверять управление системой пользовательскому потоку — он может надолго занять процессор, а при крахе потока в критической секции крах потерпит вся система, потому что прерывания никогда не будут разрешены.

Для синхронизации потоков одного процесса прикладной программист может использовать глобальные блокирующие переменные. С этими переменными, к которым все потоки процесса имеют прямой доступ, программист работает, не обращаясь к системным вызовам ОС.

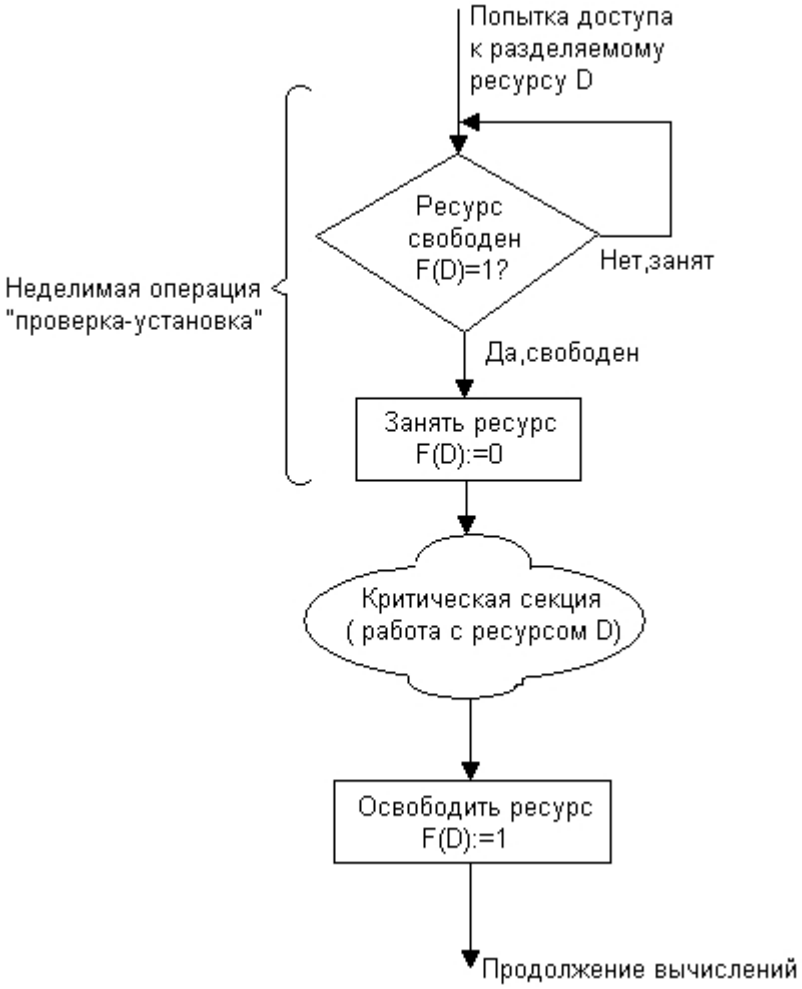


Рис. 38 Реализация критических секций с использованием блокирующих переменных

Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает. На рисунке показан фрагмент алгоритма потока, использующего для реализации взаимного исключения доступа к критическим данным D блокирующую переменную F(D). Перед входом в критическую секцию поток проверяет, не работает ли уже какой-нибудь поток с данными D. Если переменная F(D) установлена в 0, то данные заняты, и проверка циклически повторяется. Если же данные свободны (F(D) = 1), то значение переменной F(D) устанавливается в 0 и поток входит в критическую секцию. После того как поток выполнит все действия с данными O, значение переменной F(D) снова устанавливается равным 1.

Блокирующие переменные могут использоваться не только при доступе к разделяемым данным, но и при доступе к разделяемым ресурсам любого вида.

Если все потоки написаны с учетом вышеописанных соглашений, то взаимное исключение гарантируется. При этом потоки могут быть прерваны операционной системой в любой момент и в любом месте, в том числе в критической секции.

Однако следует заметить, что одно ограничение на прерывания все же имеется. Нельзя прерывать поток между выполнением операций проверки и установки блокирующей переменной. Поясним это. Пусть в результате проверки переменной поток определил, что ресурс свободен, но сразу после этого, не успев установить переменную в 0, был прерван. За время его приостановки другой поток занял ресурс, вошел в свою критическую секцию, но также был прерван, не завершив работы с разделяемым ресурсом. Когда управление было возвращено первому потоку, он, считая ресурс свободным, установил признак занятости и начал выполнять свою критическую секцию. Таким образом, был нарушен принцип взаимного исключения, что потенциально может привести к нежелательным последствиям. Во избежание таких ситуаций в системе команд многих компьютеров предусмотрена единая, неделимая команда анализа и присвоения значения логической переменной (например, команды BTC, BTR и BT5 процессора Pentium). При отсутствии такой команды в процессоре соответствующие действия должны реализовываться специальными системными примитивами, которые бы запрещали прерывания на протяжении всей операции проверки и установки.

Реализация взаимного исключения описанным выше способом имеет существенный недостаток: в течение времени, когда один поток находится в критической секции, другой поток, которому требуется тот же ресурс, получив доступ к процессору, будет непрерывно опрашивать блокирующую переменную, бесполезно тратя выделяемое ему процессорное время, которое могло бы быть использовано для выполнения какого-нибудь другого потока. Для устранения этого недостатка во многих ОС предусматриваются специальные системные вызовы для работы с критическими секциями. На рисунке показано, как с помощью этих функций реализовано взаимное исключение в операционной системе Windows NT. Перед тем как начать изменение критических данных, поток выполняет системный вызов EnterCriticalSection(). В рамках этого вызова сначала выполняется, как и в предыдущем случае, проверка блокирующей переменной, отражающей состояние критического ресурса. Если системный вызов определил, что ресурс занят (F(D) - 0), он в отличие от предыдущего случая не выполняет циклический опрос, а переводит поток в состояние ожидания D) и делает отметку о том, что данный поток должен быть активизирован, когда соответствующий ресурс освободится. Поток, который в это время использует данный ресурс, после выхода из критической секции должен выполнить системную функцию LeaveCriticalSection(), в результате чего блокирующая переменная принимает значение, соответствующее свободному состоянию ресурса (F(D) - 1), а операционная система просматривает очередь ожидающих этот ресурс потоков и переводит первый поток из очереди в состояние готовности.

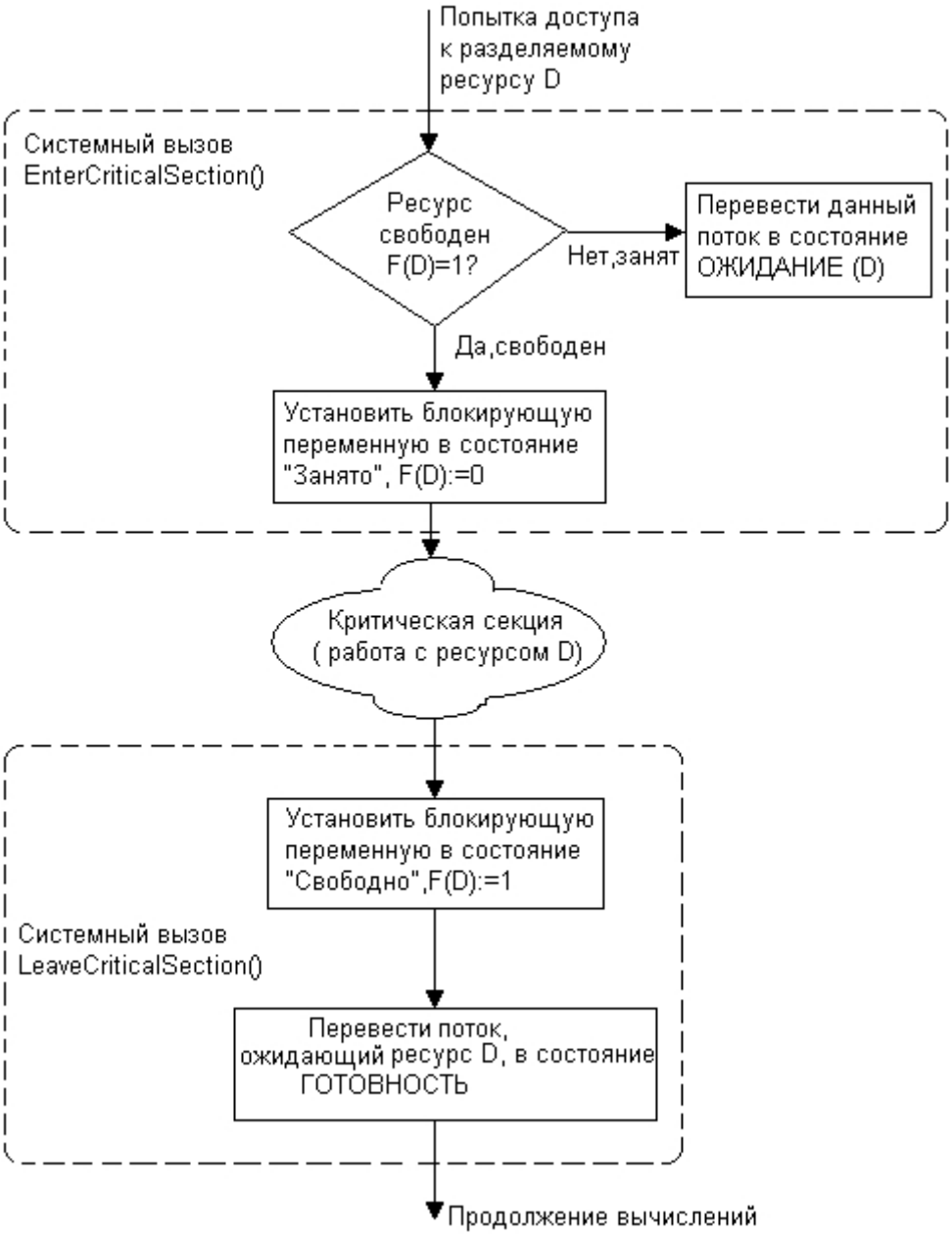


Рис. 39 Реализация взаимного исключения с использованием системных функций входа в критическую секцию и выхода из нее

Таким образом, исключается непроизводительная потеря процессорного времени на циклическую проверку освобождения занятого ресурса. Однако в тех случаях, когда объем работы в критической секции небольшой и существует высокая вероятность в очень скором доступе к разделяемому ресурсу, более предпочтительным может оказаться использование блокирующих переменных. Действительно, в такой ситуации накладные расходы ОС по реализации функции входа в критическую секцию и выхода из нее могут превысить полученную экономию.

Семафоры

Обобщением блокирующих переменных являются так называемые семафоры Дийкстры. Вместо двоичных переменных Дийкстра (Dijkstra) предложил использовать переменные, которые могут принимать целые неотрицательные значения. Такие переменные,

используемые для синхронизации вычислительных процессов, получили название семафоров.

Для работы с семафорами вводятся два примитива, традиционно обозначаемых P и V. Пусть переменная S представляет собой семафор. Тогда действия V(S) и P(S) определяются следующим образом.

- V(S): переменная S увеличивается на 1 единым действием. Выборка, наращивание и запоминание не могут быть прерваны. К переменной S нет доступа другим потокам во время выполнения этой операции.
- P(S): уменьшение S на 1, если это возможно. Если S=0 и невозможно уменьшить S, оставаясь в области целых неотрицательных значений, то в этом случае поток, вызывающий операцию P, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также являются неделимой операцией.

Никакие прерывания во время выполнения примитивов V и P недопустимы.

В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную, которую по этой причине часто называют двоичным семафором. Операция P заключает в себе потенциальную возможность перехода потока, который ее выполняет, в состояние ожидания, в то время как операция V может при некоторых обстоятельствах активизировать другой поток, приостановленный операцией P.

Рассмотрим использование семафоров на классическом примере взаимодействия двух выполняющихся в режиме мультипрограммирования потоков, один из которых пишет данные в буферный пул, а другой считывает их из буферного пула (задача писатель-читатель). Пусть буферный пул состоит из N буферов, каждый из которых может содержать одну запись. В общем случае поток-писатель и поток-читатель могут иметь различные скорости и обращаться к буферному пулу с переменой интенсивностью. В один период скорость записи может превышать скорость чтения, в другой – наоборот. Для правильной совместной работы поток-писатель должен приостанавливаться, когда все буферы оказываются занятыми, и активизироваться при освобождении хотя бы одного буфера. Напротив, поток-читатель должен приостанавливаться, когда все буферы пусты, и активизироваться при появлении хотя бы одной записи.

Введем два семафора: e – число пустых буферов, и f – число заполненных буферов, причем в исходном состоянии e=N, а f=0. Тогда работа потоков с общим буферным пулом может быть описана следующим образом: поток-писатель прежде всего выполняет операцию P(e), с помощью которой он проверяет, имеются ли в буферном пуле незаполненные буферы. В соответствии с семантикой операции P, если семафор e равен 0 (то есть свободных буферов в данный момент нет), то поток-писатель переходит в состояние ожидания. Если же значением e является положительное число, то он уменьшает число свободных буферов, записывает данные в очередной свободный буфер и после этого наращивает число занятых буферов операцией V(f). Поток-читатель действует аналогичным образом, с той разницей, что он начинает работу с проверки наличия заполненных буферов, а после чтения данных наращивает количество свободных буферов.

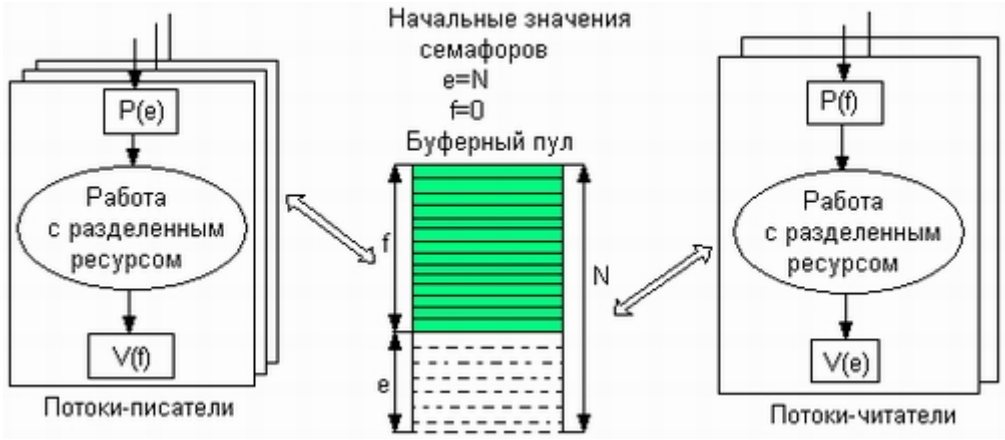


Рис. 40 Использование семафоров для синхронизации потоков

В данном случае предпочтительнее использовать семафоры вместо блокирующих переменных. Действительно, критическим ресурсом здесь является буферный пул, который может быть представлен как набор идентичных ресурсов — отдельных буферов, а значит, с буферным пулом могут работать сразу несколько потоков, и именно столько, сколько буферов в нем содержится. Использование двоичной переменной не позволяет организовать доступ к критическому ресурсу более чем одному потоку. Семафор же решает задачу синхронизации более гибко, допуская к разделяемому пулу ресурсов заданное количество потоков. Так, в нашем примере с буферным пулом могут работать максимум N потоков, часть из которых может быть «писателями», а часть — «читателями».

Таким образом, семафоры позволяют эффективно решать задачу синхронизации доступа к ресурсным пулам, таким, например, как набор идентичных в функциональном назначении внешних устройств (модемов, принтеров, портов), или набор областей памяти одинаковой величины, или информационных структур. Во всех этих и подобных им случаях с помощью семафоров можно организовать доступ к разделяемым ресурсам сразу нескольких потоков.

Семафор может использоваться и в качестве блокирующей переменной. В рассмотренном выше примере, для того чтобы исключить коллизии при работе с разделяемой областью памяти, будем считать, что запись в буфер и считывание из буфера являются критическими секциями. Взаимное исключение будем обеспечивать с помощью двоичного семафора b. Оба потока после проверки доступности буферов должны выполнить проверку доступности критической секции.

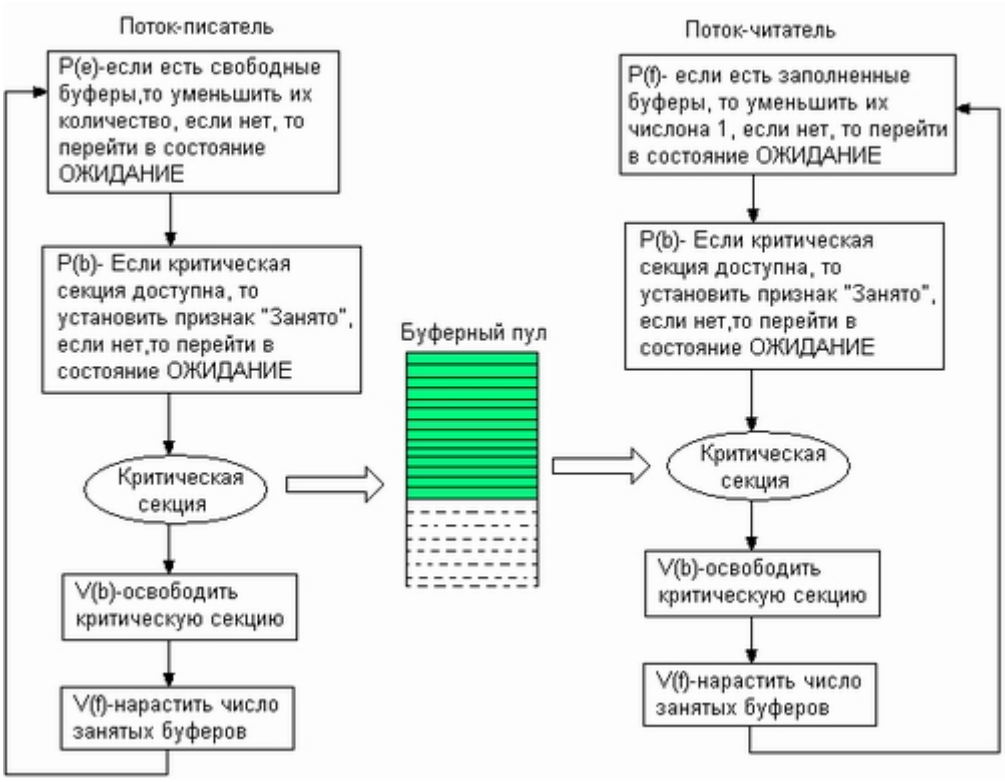


Рис. 41 Использование двоичного семафора

Тупики

Приведенный выше пример позволяет также проиллюстрировать еще одну проблему синхронизации – взаимные блокировки, называемые также дедлоками (deadlocks), клинчами (clinch), или тупиками. Покажем, что если переставить местами операции P(e) и P(b) в потоке-писателе, то при некотором стечении обстоятельств эти два потока могут взаимно блокировать друг друга, Итак, пусть поток-писатель начинает свою работу с проверки доступности критической секции — операции P(b), и пусть он первым войдет в критическую секцию. Выполняя операцию P(e), он может обнаружить отсутствие свободных буферов и перейти в состояние ожидания. Как уже было показано, из этого состояния его может вывести только поток-читатель, который возьмет очередную запись из буфера. Но поток-читатель не сможет этого сделать, так как для этого ему потребуется войти в критическую секцию, вход в которую заблокирован потоком-писателем. Таким образом, ни один из этих потоков не может завершить начатую работу и возникнет тупиковая ситуация, которая не может разрешиться без внешнего воздействия.

Рассмотрим еще один пример тупика. Пусть двум потокам, принадлежащим разным процессам и выполняющимся в режиме мультипрограммирования, для выполнения их работы нужно два ресурса, например принтер и последовательный порт. Такая ситуация может возникнуть, например, во время работы приложения, задачей которого является распечатка информации, поступающей по модемной связи.

На рисунке показаны фрагменты соответствующих программ. Поток А запрашивает сначала принтер; а затем порт, а поток В запрашивает устройства в обратном порядке. Предположим, что после того, как ОС назначила принтер потоку А и установила связанную с этим ресурсом блокирующую переменную, поток А был прерван. Управление получил поток В, который сначала выполнил запрос на получение COM-порта, затем при выполнении следующей команды был заблокирован, так как принтер оказался уже занятым потоком А. Управление

снова получил поток A, который в соответствии со своей программой сделал попытку занять порт и был заблокирован, поскольку порт уже выделен потоку B. В таком положении потоки A и B могут находиться сколь угодно долго.

В зависимости от соотношения скоростей потоков они могут либо взаимно блокировать друг друга, либо образовывать очереди к разделяемым ресурсам, либо совершенно независимо использовать разделяемые ресурсы.

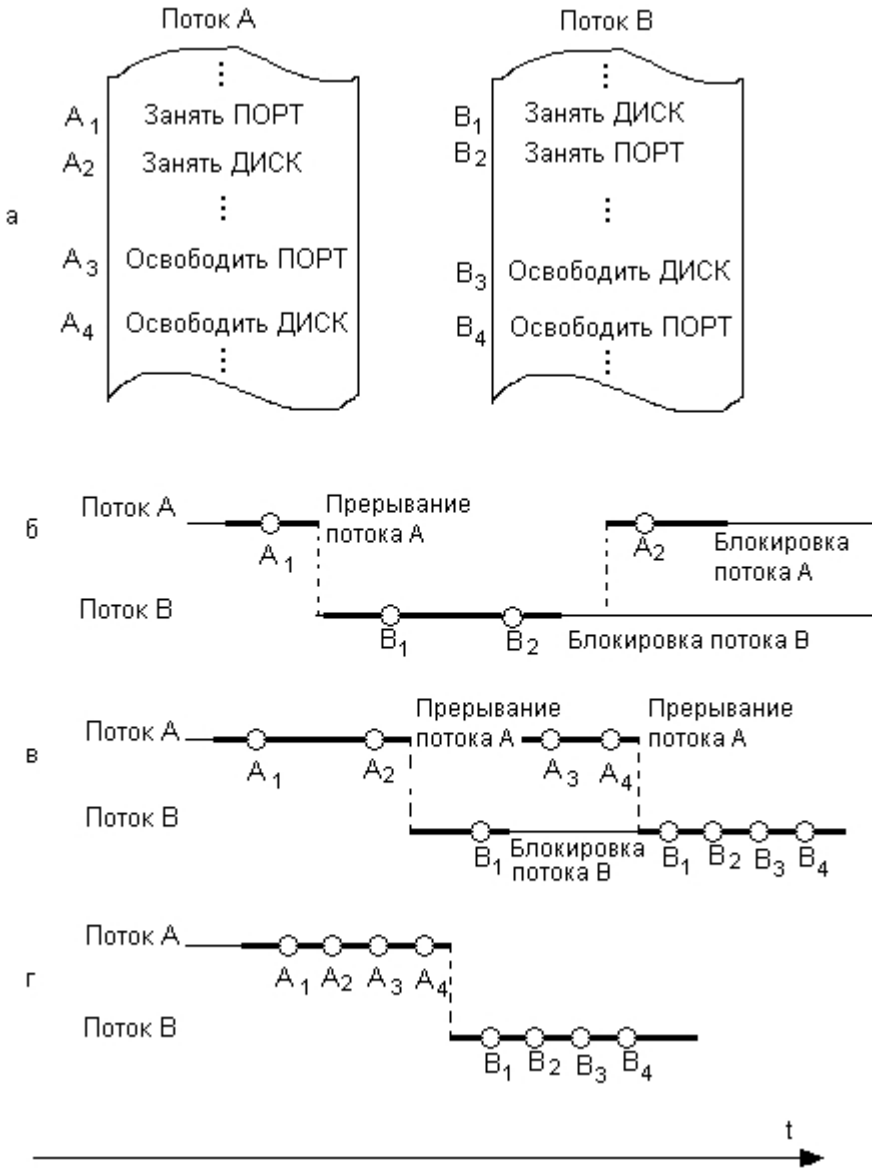


Рис. 42 Возникновение взаимных блокировок при выполнении программы

В рассмотренных примерах тупик был образован двумя потоками, но взаимно блокировать друг друга может и большее число потоков. Возможно такое распределение ресурсов R_i между несколькими потоками T_j, которое приводит к возникновению взаимных блокировок. Стрелки обозначают потребность потока в ресурсах. Сплошная стрелка означает, что соответствующий ресурс был выделен потоку, а пунктирная стрелка соединяет поток с тем ресурсом, который необходим, но не может быть пока выделен, поскольку занят другим потоком. Например, потоку T_1 для выполнения работы необходимы ресурсы R_1 и R_2, из которых выделен только один — R_1, а ресурс R_2 удерживается потоком T_2. Ни один из четырех показанных на рисунке потоков не может продолжить свою работу, так как не имеет всех необходимых для этого ресурсов.

Невозможность потоков завершить начатую работу из-за возникновения взаимных блокировок снижает производительность вычислительной системы. Поэтому проблеме предотвращения тупиков уделяется большое внимание. На тот случай, когда взаимная блокировка все же возникает, система должна предоставить администратору средства, с помощью которых он смог бы распознать тупик, отличить его от обычной блокировки из-за временной недоступности ресурсов. И, наконец, если тупик диагностирован, то нужны средства для снятия взаимных блокировок и восстановления нормального вычислительного процесса.

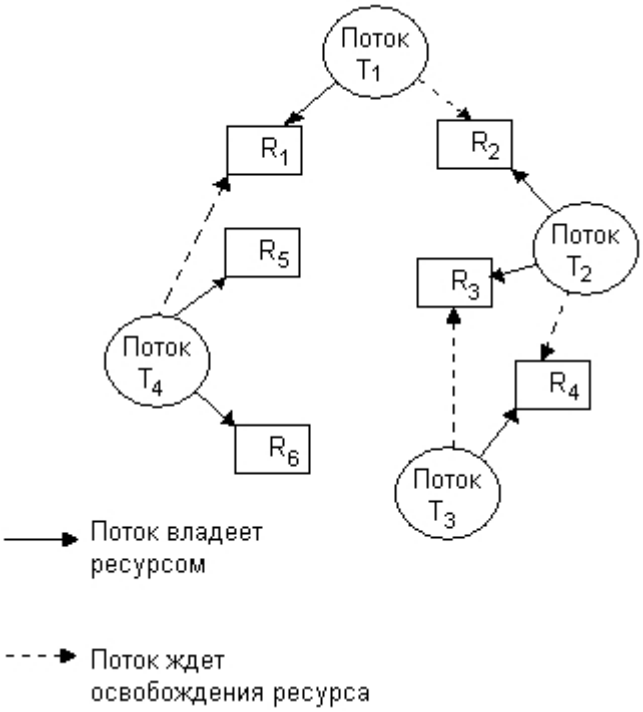


Рис. 43 Взаимная блокировка нескольких потоков

Тупики могут быть предотвращены на стадии написания программ, то есть программы должны быть написаны таким образом, чтобы тупик не мог возникнуть при любом соотношении взаимных скоростей потоков. Так, если бы в рассмотренном ранее примере поток А и поток В запрашивали ресурсы в одинаковой последовательности, то тупик был бы в принципе невозможен. Другой, более гибкий подход к предотвращению тупиков заключается в том, что ОС каждый раз при запуске задач анализирует их потребности в ресурсах и определяет, может ли в данной мультипрограммной смеси возникнуть тупик. Если да, то запуск новой задачи временно откладывается. ОС может также использовать определенные правила при назначении ресурсов потокам, например, ресурсы могут выделяться операционной системой в определенной последовательности, общей для всех потоков.

В тех же случаях, когда тупиковую ситуацию не удалось предотвратить, важно быстро и точно ее распознать, поскольку блокированные потоки не выполняют никакой полезной работы. Если тупиковая ситуация образована множеством потоков, занимающих массу ресурсов, распознавание тупика является нетривиальной задачей. Существуют формальные, программно реализованные методы распознавания тупиков, основанные на ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам. Анализ этих таблиц позволяет обнаружить взаимные блокировки.

Если же тупиковая ситуация возникла, то не обязательно снимать с выполнения все заблокированные потоки. Можно снять только часть из них, освободив ресурсы, ожидаемые

остальными потоками, можно вернуть некоторые потоки в область подкачки, можно совершить «откат» некоторых потоков до так называемой контрольной точки, в которой запоминается вся информация, необходимая для восстановления выполнения программы с данного места. Контрольные точки расставляются в программе в тех местах, после которых возможно возникновение тупика.

Синхронизирующие объекты ОС

Рассмотренные выше механизмы синхронизации, основанные на использовании глобальных переменных процесса, обладают существенным недостатком — они не подходят для синхронизации потоков разных процессов. В таких случаях операционная система должна предоставлять потокам системные объекты синхронизации, которые были бы видны для всех потоков, даже если они принадлежат разным процессам и работают в разных адресных пространствах.

Примерами таких синхронизирующих объектов ОС являются системные семафоры, мьютексы, события, таймеры и другие – их набор зависит от конкретной ОС, которая создает эти объекты по запросам процессов. Чтобы процессы могли разделять синхронизирующие объекты, в разных ОС используются разные методы. Некоторые ОС возвращают указатель на объект. Этот указатель может быть доступен всем родственным процессам, наследующим характеристики общего родительского процесса. В других ОС процессы в запросах на создание объектов синхронизации указывают имена, которые должны быть им присвоены. Далее эти имена используются разными процессами для манипуляций объектами синхронизации. В таком, случае работа с синхронизирующими объектами подобна работе с файлами. Их можно создавать, открывать, закрывать, уничтожать.

Кроме того, для синхронизации могут быть использованы такие «обычные» объекты ОС, как файлы, процессы и потоки. Все эти объекты могут находиться в двух состояниях: сигнальном и несигнальном (свободном). Для каждого объекта смысл, вкладываемый в понятие «сигнальное состояние», зависит от типа объекта. Так, например, поток переходит в сигнальное состояние тогда, когда он завершается. Процесс переходит в сигнальное состояние тогда, когда завершаются все его потоки. Файл переходит в сигнальное состояние в том случае, когда завершается операция ввода-вывода для этого файла. Для остальных объектов сигнальное состояние устанавливается в результате выполнения специальных системных вызовов. Приостановка и активизация потоков осуществляются в зависимости от состояния синхронизирующих объектов ОС.

Потоки с помощью специального системного вызова сообщают операционной системе о том, что они хотят синхронизировать свое выполнение с состоянием некоторого объекта. Будем далее называть этот системный вызов Wait(X), где X — указатель на объект синхронизации. Системный вызов, с помощью которого поток может перевести объект синхронизации в сигнальное состояние, назовем Set(X).

Поток, выполнивший системный вызов Wait(X), переводится операционной системой в состояние ожидания до тех пор, пока объект X не перейдет в сигнальное состояние. Примерами системных вызовов типа Wait() и Set() являются вызовы WaitForSingleObject() и SetEvent() в Windows NT, DosSenWait() и OosSemSet() в OS/2, sleep() и wakeup() в UNIX.

Поток может ожидать установки сигнального состояния не одного объекта, а нескольких. При этом поток может попросить ОС активизировать его при установке либо одного из указанных объектов, либо всех объектов. Поток может в качестве аргумента системного вызова Wait() указать также максимальное время, которое он будет ожидать перехода объекта в сигнальное состояние, после чего ОС должна его активизировать в любом случае. Может случиться, что установки некоторого объекта в сигнальное состояние ожидают сразу

несколько потоков. В зависимости от объекта синхронизации в состояние готовности могут переводиться либо все ожидающие это событие потоки, либо один из них.

Синхронизация тесно связана с планированием потоков, Во-первых, любое обращение потока с системным вызовом Wait(X) влечет за собой действия в подсистеме планирования – этот поток снимается с выполнения и помещается в очередь ожидающих потоков, а из очереди готовых потоков выбирается и активизируется новый поток. Во-вторых, при переходе объекта в сигнальное состояние (в результате выполнения некоторого потока — либо системного, либо прикладного) ожидающий этот объект поток (или потоки) переводится в очередь готовых к выполнению потоков. В обоих случаях осуществляется перепланирование потоков, при этом если в ОС предусмотрены изменяемые приоритеты и/или кванты времени, то они пересчитываются по правилам, принятым в этой операционной системе.

Рассмотрим несколько примеров, когда в качестве синхронизирующих объектов используются файлы, потоки и процессы.

Пусть программа приложения построена так, что для выполнения запросов, поступающих из сети, основной поток создает вспомогательные серверные потоки.

При поступлении от пользователя команды завершения приложения основной поток должен дождаться завершения всех серверных потоков и только после этого завершиться сам. Следовательно, процедура завершения должна включать вызов Wait(Xl, X2, ...), где XI, X2 — указатели на серверные потоки. В результате выполнения данного системного вызова основной поток будет переведен в состояние ожидания и останется в нем до тех пор, пока все серверные потоки не перейдут в сигнальное состояние, то есть завершатся. После этого OG переведет основной поток в состояние готовности. При получении доступа к процессору основной поток завершится.

Другой пример. Пусть выполнение некоторого приложения требует последовательных работ-этапов. Для каждого этапа имеется свой отдельный процесс. Сигналом для начала работы каждого следующего процесса является завершение предыдущего. Для реализации такой логики работы необходимо в каждом процессе, кроме первого, предусмотреть выполнение системного вызова Wait(X), в котором синхронизирующим объектом является предшествующий поток.

Объект-файл, переход которого в сигнальное состояние соответствует завершению операции ввода-вывода с этим файлом, используется в тех случаях, когда поток, инициировавший эту операцию, решает дождаться ее завершения, прежде чем продолжить свои вычисления.

Однако круг событий, е которыми потоку может потребоваться синхронизировать свое выполнение, отнюдь не исчерпывается завершением потока, процесса иди операции ввода-вывода. Поэтому в ОС, как правило, имеются и другие, более универсальные объекты синхронизации, такие как событие (event), мъютекс (nmtex), системный семафор и другие.

Мьютекс, как и семафор, обычно используется для управления доступом к данным.

В отличие от объектов-потоков, объектов-процессов и объектов-файлов, которые при переходе в сигнальное состояние переводят в состояние готовности все потоки, ожидающие этого события, объект - мьютекс «освобождает» из очереди ожидающих только один поток.

Работа мьютекса хорошо поясняется в терминах «владения». Пусть поток, который, пытаясь получить доступ к критическим данным, выполнил системный вызов Wait(X), где X — указатель на мьютекс. Предположим, что мьютекс находится в сигнальном состоянии, в этом случае поток тут же становится его владельцем, устанавливая его в несигнальное состояние, и входит в критическую секцию. После того как поток выполнил работу с критическими

данными, он «отдает» мьютекс, устанавливая его в сигнальное состояние. В этот момент мьютекс свободен и не принадлежит ни одному потоку. Если какой-либо поток ожидает его освобождения, то он становится следующим владельцем этого мьютекса, одновременно мьютекс переходит в несигнальное состояние.

Объект-событие (в данном случае слово «событие» используется в узком смысле, как обозначение конкретного вида объектов синхронизации) обычно используется не для доступа к данным, а для того, чтобы оповестить другие потоки о том, что некоторые действия завершены. Пусть, например, в некотором приложении работа организована таким образом, что один поток читает данные из файла в буфер памяти, а другие потоки обрабатывают эти данные, затем первый поток считывает новую порцию данных, а другие потоки снова ее обрабатывают и так далее. В начале работы первый поток устанавливает объект-событие в несигнальное состояние. Все остальные потоки выполнили вызов Wait(X), где X — указатель события, и находятся в приостановленном состоянии, ожидая наступления этого события. Как только буфер заполняется, первый поток сообщает об этом операционной системе, выполняя вызов Set(X). Операционная система просматривает очередь ожидающих потоков и активизирует все потоки, которые ждут этого события.

Сигналы

Сигнал дает возможность задаче реагировать на событие, источником которого может быть операционная система или другая задача. Сигналы вызывают прерывание задачи и выполнение заранее предусмотренных действий. Сигналы могут вырабатываться синхронно, то есть как результат работы самого процесса, а могут быть направлены процессу другим процессом; то есть вырабатываться асинхронно. Синхронные сигналы чаще всего приходят от системы прерываний процессора и свидетельствуют о действиях потока процесса, блокируемых аппаратурой, например деление на нуль, ошибка адресации, нарушение защиты памяти и т. д.

Примером асинхронного сигнала является сигнал с терминала. Во многих ОС предусматривается оперативное снятие процесса с выполнения. Для этого пользователь может нажать некоторую комбинацию клавиш (Ctrl+C, Ctrl+Break), в результате чего ОС вырабатывает сигнал и направляет его активному процессу. Сигнал может поступить в любой момент выполнения процесса (то есть он является асинхронным), требуя от процесса немедленного завершения работы. В данном случае реакцией на сигнал является безусловное завершение процесса.

В системе может быть определен набор сигналов. Программный код процесса, которому поступил сигнал, может либо проигнорировать его, либо прореагировать на него стандартным действием (например, завершиться), либо выполнить специфические действия, определенные прикладным программистом. В последнем случае в программном коде необходимо предусмотреть специальные системные вызовы, с помощью которых операционная система информируется, какую процедуру надо выполнить в ответ на поступление того или иного сигнала.

Сигналы обеспечивают логическую связь между процессами, а также между процессами и пользователями (терминалами). Поскольку посылка сигнала предусматривает знание идентификатора процесса, то взаимодействие посредством сигналов возможно только между родственными процессами, которые могут получить данные об идентификаторах друг друга.

В распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, блокирующие переменные, семафоры, сигналы и другие аналогичные средства, основанные на разделяемой памяти, оказываются

непригодными. В таких системах синхронизация может быть реализована только посредством обмена сообщениями.

Обмен сообщениями (message passing) (Хоар, 1978 год)

Цели предложенного подхода - избавиться от проблем разделения памяти и предложить модель взаимодействия процессов для распределенных систем. Предлагается две операции:

send(destination, &message, msize);

receive([source], &message, msize);

соответственно, передача и прием сообщения.

В качестве адресата выступает процесс. Отправитель может не специфицироваться (например быть любым - широковещательное сообщение). Передача может осуществляться как с буферизацией (почтовые ящики) так и без нее (рандеву - Ада, Оккам). Пайпы ОС UNIX - почтовые ящики, заменяют файлы и не хранят границы сообщений (все сообщения объединяются в одно большое, которое можно читать произвольными порциями.

Механизмы семафоров и обмена сообщениями взаимозаменяемы семантически и на мультипроцессорах могут быть реализованы один через другой.

Реализация взаимоисключений

Алгоритм Петерсона

Первое решение проблемы взаимного исключения, удовлетворяющее всем требованиям, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение.

Пусть два потока имеют доступ к массиву флагов готовности и к переменной очередности. Предлагается использовать следующий код.

```
int ready[2] = {0, 0};  /* разделяемая переменная */
int turn;               /* разделяемая переменная */
while(some_condition) {
     ready[i] = 1;
     turn = 1 - i;
     while( ready[1-i] && turn == 1-i )
          ;
     critical section
     ready[i] = 0;
     remainder section
}
```

При исполнении пролога критической секции поток P_1 заявляет о своей готовности выполнить критический участок и одновременно предлагает другому потоку приступить к его выполнению. Если оба потока подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из

предложений всегда последует после другого. Тем самым работу в критическом участке продолжит поток, которому было сделано последнее предложение.

Так как в процессе ожидания разрешения на вход поток P_0 не изменяет значения переменных, то он сможет начать исполнение своего критического участка после не более чем одного прохода по критической секции потока P_1.

Алгоритм булочной (Bakery algorithm)

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух потоков. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих потоков, который получил название алгоритм булочной. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же поток) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех потоков будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива

```
enum {false, true} choosing[n];
int number[n];
```

Изначально 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(*a_0*, *a_1*, ..., *a_n*) — это число ***k*** такое, что ***k*** >= a_i для всех ***i = 0, ...,n***

Структура кода потока P_i для алгоритма булочной приведена ниже

```
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
}
```

Аппаратная поддержка взаимоисключений

Наличие аппаратной поддержки взаимоисключений позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как атомарные операции. Рассмотрим, как концепции таких команд могут быть использованы для реализации взаимоисключений.

Команда Test-and-Set (Проверить и присвоить 1)

О выполнении команды Test-and-Set, осуществляющей проверку значения логической переменной с одновременной установкой ее значения в 1, можно думать, как об атомарном выполнении следующей функции.

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

С использованием этой атомарной команды мы можем записать следующий алгоритм, обеспечивающий взаимоисключения.

```
int lock = 0; /* Общая переменная */
while(some_condition) {
     while(Test_and_Set(&lock))
          ;
     critical section
     lock = 0;
     remainder section
}
```

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет условию ограниченного ожидания.

Команда Swap (Обменять значения)

Выполнение команды Swap, обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией.

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

Применяя атомарную команду Swap, мы можем реализовать предыдущий алгоритм, введя дополнительную логическую переменную key локальную для каждого процесса (потока):

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

Приведенный вариант также не удовлетворяет условию ограниченного ожидания, однако и в этом, и в предыдущем случае условие выполняется после небольшой модификации алгоритма.

Классические задачи взаимодействия потоков

Задача "Производитель-потребитель"

Имеется большое число вариантов постановки и решения такой задачи в рамках конкретных ОС. Наиболее простой случай, когда взаимодействуют два потока с жестко распределенными между ними функциями. Один поток вырабатывает сообщения, предназначенные для восприятия и обработки другим потоком поток, вырабатывающий сообщения, называют производителем, а воспринимающий сообщения — потребителем. Потоки взаимодействуют через некоторую обобщенную область памяти, которая по смыслу является критическим ресурсом. В эту область поток-производитель должен помещать очередное сообщение (в простейшем случае предполагается, что область способна хранить только одно сообщение и сообщения фиксированной длины), а поток-потребитель должен считывать очередное сообщение. Необходимо согласовать работы двух потоков при одностороннем (в простейшем случае) обмене сообщениями по мере развития потоков таким образом, чтобы удовлетворить следующим требованиям:

- выполнять требования задачи взаимного исключения по отношению к критическому ресурсу – обобщенной памяти для хранения сообщения;

- учитывать состояние обобщенной области памяти, характеризующей возможность или невозможность посылки (принятия) очередного сообщения;

Попытка потока-производителя поместить очередное сообщение в область, из которой не было считано предыдущее сообщение потоком-потребителем, должна быть блокирована. Поток-производитель должен быть либо оповещен о невозможности помещения сообщения, либо переведен в состояние ожидания возможности поместить очередное сообщение через некоторое время в область памяти, по мере ее освобождения. Аналогично должна быть блокирована попытка потока-потребителя считать сообщение из области в ситуации, когда поток-производитель не поместил туда очередного сообщения. Возможны также несколько

вариантов реакции на данную ситуацию – сообщение о невозможности считывания либо перевод потока-потребителя в состояние ожидания поступления очередного сообщения. Если используется вариант с ожиданиями изменения состояния обобщенной области для хранения сообщения, необходимо обеспечить перевод ожидающих потоков в состояние готовности всякий раз, когда изменится состояние области. Либо поток-производитель поместит очередное сообщение в область. Оно теперь может быть считано ожидающим потоком-потребителем. Либо поток-потребитель считал очередное сообщение из области и обеспечил возможность ожидающему потоку-производителю послать очередное сообщение.

Множественность постановки задачи “производитель-потребитель” определяется тем, что число как потоков-потребителей, так и потоков-производителей может быть большим одного. Каждый из таких процессов может устанавливать не только одностороннюю, но и двустороннюю связь через одну и ту же обобщенную область или другие области. Области могут хранить не одно, а большее количество сообщений.

Решение данной задачи было представлено выше.

Задача "Читатели-писатели"

Эта задача также имеет много вариантов. Наиболее характерная область использования этой задачи — при построении файловых систем ОС. В отношении некоторой области памяти, являющейся по смыслу критическим ресурсом для параллельных потоков, работающих с ней, выделяется два типа потоков.

Первый тип — потоки-читатели. Они считывают одновременно информацию из области, если это допускается при работе с конкретным устройством памяти.

Второй тип — потоки-писатели. Они записывают информацию в область и могут делать это, только исключая как друг друга, так и потоки-читатели, т. е. запись должна удовлетворяться на основе решения задачи взаимного исключения. Имеются различные варианты взаимодействия между потоками-писателями и потоками-читателями. Наиболее широко распространены следующие.

1. Устанавливается приоритетность в использовании критического ресурса потокам-читателям. Если хотя бы один поток-читатель пользуется ресурсом, то он закрыт для использования всем потокам-писателям и доступен для использования всем потокам-читателям.

2. Больший приоритет имеют потоки-писатели. При появлении запроса от потока-писателя необходимо закрыть ресурс для использования всем потокам-читателям, которые выдадут запрос позже него.

Рассмотрим следующий пример решения задачи программирования взаимодействия множества потоков с использованием семафоров. Пусть выполняются n потоков, которые разделяют некоторый объект программы, например переменную. Часть потоков (писатели) только модифицируют разделяемый объект, другие (читатели) - только считывают значение. Необходимо организовать корректное выполнение этой системы взаимодействующих потоков.

Прежде всего, отметим, что потоки-читатели могут иметь одновременный доступ к разделяемому объекту, так как чтение не меняет значение объекта и, следовательно, потоки-читатели не могут влиять друг на друга. Для потоков-писателей следует организовать взаимное исключение при доступе к разделяемому объекту.

Далее, потоки-писатели должны иметь преимущество при доступе к разделяемому объекту, поскольку они готовы записать новые данные. Следовательно, потоки-читатели должны

получать доступ к разделяемому объекту лишь в том случае, если нет потоков-писателей, желающих получить доступ к этому разделяемому объекту. А потому, если поток-писатель изъявил желание получить доступ к разделяемому объекту, все потоки-читатели должны быть задержаны. С другой стороны, потоки-читатели тоже нехорошо совершенно дискриминировать и потому потоки-читатели, изъявившие желание получить доступ к разделяемому объекту до того, как появился первый поток-писатель, должны иметь возможность завершить исполнение своего критического интервала.

Одно из решений задачи выглядит следующим образом.

```
Семафор Mutex = 1;   // Контроль доступа к переменной ReadersCount
Семафор Access = 1;  // Контроль доступа к данным
int ReadersCount = 0;// Число читателей
void Reader( void ){
     while( true ){
          P(Mutex);          // Получение монопольного доступа к RC
          ReadersCount ++;   // Увеличение числа читателей
          if( ReadersCount == 1 ) // Если это первый читатель
               P(Access);    // Захватываем доступ для читателей
          V(Mutex);          // Отказ от монопольного доступа к RC
          … Доступ к данным …
          P(Mutex);          // Получение монопольного доступа к RC
          ReadersCount --;   // Уменьшение числа читателей
          if( ReadersCount == 0 ) // Если это последний читатель
               V(Access);    // Открываем доступ для всех
          V(Mutex);          // Отказ от монопольного доступа к RC
          … Работа вне критической области …
     }
}
void Writer( void ){
     while( true ){
          … Формирование данных …
          P(Access);    // Захватываем доступ для себя
          … Запись данных …
          V(Access);    // Открываем доступ для всех
     }
}
```

Данное решение справедливо для случая привилегированных потоков-читателей.

Задача "Обедающие философы"

Хотя и название, и формулировка задачи носят некоторый абстрактный характер, такая задача синхронизации также имеет место при построении систем распределения ресурсов в составе ОС. В рамках этой задачи формулируются требования на синхронизацию работы потоков, которые совместно используют пересекающиеся группы ресурсов.

Обратимся к формулировке задачи, изложенной в терминологии, предложенной впервые Э.Дейкстрой.

За круглым столом расставлены стулья, каждый из которых занимает определенный философ (в первой формулировке Э. Дейкстры число философов равнялось пяти). В центре стола – большое блюдо спагетти, а на столе лежат пять вилок — каждая между двумя соседними тарелками. Каждый философ находится только в двух состояниях — либо он размышляет, либо ест спагетти. Начать думать после еды философу ничто не мешает. Но чтобы начать есть, необходимо выполнить ряд условий. Предполагается, что любой философ, прежде чем начать есть, должен положить из общего блюда спагетти себе в тарелку. Для этого он одновременно должен держать в левой и правой руках по вилке, набрать спагетти в тарелку с их помощью и, не выпуская вилок из рук, начать есть. Закончив еду, философ кладет вилки слева и справа от своей тарелки и опять начинает размышлять до тех пор, пока снова проголодается.

Нетрудно заметить, что вилки в данной “столовой ситуации” выступают в качестве пересекающихся ресурсов. Неприятная ситуация может возникнуть в случае, когда философы одновременно проголодаются и одновременно попытаются взять, например, свою левую вилку. В данном случае возникает тупиковая ситуация, так как никто из них не может по условию начать есть, не имея второй вилки. Однако, вторая вилка может появиться для любого философа только от соседа справа, который в свою очередь ждет появления для себя вилки от своего соседа справа и одновременно не хочет положить свою левую вилку на стол и т. д. Вторая неприятная ситуация (голодание) может возникнуть уже в отношении не всех, а только одного процесса. В обществе философов такая ситуация возникает в случае заговора двух соседей слева и справа против философа, в отношении которого строятся козни. Каждый раз, когда последний желает утолить голод, заговорщики, опережая его, поочередно забирают вилки то слева, то справа от него. Такие согласованные действия злоумышленников приводят жертву к вынужденному голоданию, так как он никогда не может воспользоваться обеими вилками.

Предлагаемое решение исключает взаимоблокировку и позволяет реализовать максимально возможный параллелизм. Состояние философа: ест, размышляет или голодает (пытаясь получить вилки) – отражены в массиве state.

```
#define LEFT        (i+4)%5
#define RIGHT       (i+1)%5
#define THINKING    0
#define HUNGRY      1
#define EATING      2
int state[5];  // Массив состояний философов
Семафор Mutex; // Взаимное исключение для критических областей
Семафор S[5];  // Каждому философу по семафору
void Philosopher( int i ){ // i – номер философа
```

```
     while( true ){
          think();        // Философ размышляет
          take_forks(i); // Получает 2 вилки или блокируется
          eat();          // Философ обедает
          put_forks(i);  // Кладет на стол обе вилки
     }
}
void take_forks( int i){
     P(Mutex);            // Вход в критическую область
     state[i] = HUNGRY;  // Наличие голодного философа
     test(i);             // Попытка получить две вилки
     V(Mutex);            // Выход из критической области
     P(s[i]);             // Блокируемся, если вилок не досталось
}
void put_forks( int i){
     P(Mutex);            // Вход в критическую область
     state[i] = THINKING;// Наличие голодного философа
     test(LEFT);          // Попытка накормить соседа слева
     test(RIGHT);         // Попытка накормить соседа справа
     V(Mutex);            // Выход из критической области
}
void test( int i ){
 if(state[i]==HUNGRY&&state[LEFT]!=EATING&& state[RIGHT]!=EATING){
     state[i] = EATING;
     V(s[i]);
 }
}
```

Проблема спящего брадобрея

Данная задача формулируется следующим образом.

В парикмахерской есть один брадобрей, его кресло и N стульев для посетителей. Если желающих воспользоваться его услугами нет, брадобрей сидит и спит. Если в парикмахерскую приходит клиент, он должен разбудить брадобрея. Если клиент приходит и видит, что брадобрей занят, он либо садится на стул (если есть место), либо уходит (если места нет).

В предлагаемом решении используется три семафора: Customers, для подсчета ожидающих посетителей (клиент в кресле брадобрея не учитывается, поскольку он уже не ждет), Barbers

– количество брадобреев, простаивающих в ожидании клиента (0 или 1), и Mutex для реализации взаимного исключения.

```
#define CHAIRS 5
Семафор Customers = 0; // Количество ожидающих посетителей
Семафор Barbers = 0;   // Количество ожидающих брадобреев
Семафор Mutex = 1;
int waiting = 0;
void barber( void ){
     while( true ){
          P(Customers); // Ждать посетителей
          P(Mutex);     // Запрос доступа к переменной waiting
          waiting --;   // Уменьшение числа ожидающих клиентов
          V(Barbers);   // Один брадобрей готов к работе
          V(Mutex);     // Отказ от доступа к переменной waiting
          … Постричь клиента …
     }
}
void customer( void ){
     P(Mutex);          // Запрос доступа к переменной waiting
     if( waiting < CHAIRS ){ // Есть ли свободные стулья?
          waiting ++;   // Увеличение числа ожидающих клиентов
          V(Customers); // При необходимости, разбудить брадобрея
          V(Mutex);     // Отказ от доступа к переменной waiting
          P(Barbers);   // Ждать освобождения брадобрея
          … Постричься …
     }
     else{
          V(Mutex);      // Много посетителей, придется уйти
     }
}
```

Помимо представленных существует много задач с экзотическими названиями и содержанием, но имеющих, тем не менее, совершенно определенную практическую значимость при управлении процессами и распределении ресурсов.

Передача данных между взаимодействующими потоками

При взаимодействии потокам необходимо передавать друг другу данные. По объему передаваемой информации и степени возможного воздействия на поведение другого потока все средства такого обмена можно условно разделить на три категории.

- Сигнальные. Передается минимальное количество информации - один бит, "да" или "нет". Используются, как правило, для извещения потока о наступлении какого-либо события. Сигналы, очевидно, могут одновременно использоваться и для синхронизации выполнения потоков.

- Канальные. Один поток может последовательно передавать другому данные. Передача может осуществляться единым потоком или в виде последовательности сообщений. Объем передаваемой информации в единицу времени ограничен пропускной способностью. Передача данных может быть синхронной и асинхронной. Примеры реализаций: именованные и неименованные каналы, очереди сообщений, сокеты. Очевидно, канальные средства передачи данных также могут использоваться для синхронизации выполнения.

- Разделяемая память. Два или более процессов могут совместно использовать некоторую область адресного пространства (в случае нескольких потоков одного процесса они разделяют память этого процесса изначально). Созданием и поддержкой разделяемой памяти занимается операционная система. Возможность обмена информацией подобным способом максимальна. Разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

При взаимодействии потоков основной проблемой, с которой приходится сталкиваться, является синхронизация их исполнения. Передача данных между потоками – технический вопрос и требует в большинстве случаев изучения системных вызовов, реализованных в конкретной операционной системе. Исключение составляют случаи, когда механизм обмена данными обладает синхронизирующими свойствами. В этом случае пригодны все рассуждения, приведенные при описании использования синхронизирующих объектов.

Архитектура файловой системы

Файлы с точки зрения пользователя

Системы управления данными на внешних носителях появились одновременно с самими носителями, однако первоначально каждая прикладная программа сама решала проблему организации структурного хранения информации. Такая ситуация затрудняла организацию одновременного хранения на носителе данных нескольких программ и явилась предпосылкой появления централизованных систем управления внешней памятью. С технической стороны этому способствовало появление устройств, поддерживающих быстрый доступ по произвольному адресу (магнитные диски сменили магнитные ленты).

Использование централизованной системы управления внешней памятью подразумевает наличие общепринятой модели представления данных. Основным элементом этой модели является понятие файла.

Файл – именованный блок данных. Современные системы, ориентированные на работу с пользователем, оперируют понятиями файл и **файловая система** – логическая структура, организованная на базе последовательно адресуемого адресного пространства на внешнем устройстве и обеспечивающая уникальное именование файлов и доступ к данным файлов по их именам. Для файловой системы определено множество операций, и естественно, операции записи или чтения файла концептуально проще, чем низкоуровневые операции работы с устройствами.

Система управления файлами – это часть операционной системы, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во внешней памяти и обеспечить пользователю удобный интерфейс при работе с этими данными. Система управления файлами берет на себя распределение внешней памяти, отображение имен файлов и других объектов файловой системы в адреса внешней памяти и обеспечение доступа к данным.

Например, операционная система может разделить внешнюю память на блоки фиксированного размера (4096 байт). С точки зрения пользователя, каждый файл состоит из набора индивидуальных элементов, называемых записями (например, характеристика какого-нибудь объекта). Каждый файл хранится в виде определенной последовательности блоков (не обязательно смежных); каждый блок хранит целое число записей. Адреса блоков, содержащих данные файла, могут быть организованы в связный список и вынесены в отдельную таблицу в памяти. Или адреса блоков данных файла могут храниться в виде массива в отдельном блоке внешней памяти (так называемом индексе или индексном узле). Или информация о расположении блоков файла на внешнем носителе может быть организована любым другим способом. К сказанному необходимо добавить, что в современных условиях файлы обычно представляют собой неструктурированную явно последовательность байтов (то есть длина записи равна 1).

При работе с файлами операционная система хранит в оперативной памяти некоторую информацию о файлах, например, расположение файла на внешнем носителе, права доступа к файлу и т.д. В этом случае говорят о системной таблице индексов или индексных дескрипторов файлов.

Фактическим стандартом логической организации хранилища файлов является использование иерархической структуры, называемой **деревом каталогов (директорий)**. **Каталог (директория)** – специальный тип объекта файловой системы, который не хранит данные, но может содержать другие объекты файловой системы, например, файлы и другие

каталоги. Файловая система имеет единственную корневую директорию, и при установлении правила, что два объекта в одной директории не могут иметь одинаковые имена, каждому файлу можно поставить в соответствие так называемое полное имя, содержащее названия всех каталогов на пути из корневой директории к данному файлу. Данное имя, во-первых, уникально, во-вторых, позволяет быстро найти файл в файловой системе.

Итак, понятие "файловая система" включает:

- совокупность всех файлов на диске;

- наборы структур данных, используемых для управления файлами, такие, например, как каталоги, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске и т.д.;

- комплекс системных программных средств, реализующих управление файлами, в частности: создание, уничтожение, чтение, запись, именование, поиск и другие операции над файлами.

Файлы управляются операционной системой. То, как они структурированы, поименованы, используются, защищены, реализованы - одна из главных тем проектирования ОС. Перечислим основные функции файловой системы.

- Идентификация файлов. Связывание имени файла с выделенным ему пространством внешней памяти.

- Распределение внешней памяти между файлами.

- Поддержка операций управления объектами файловой системы.

- Обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера.

- Обеспечение защиты от несанкционированного доступа.

- Обеспечение совместного доступа к файлам, не требуя от пользователя специальных усилий по обеспечению синхронизации доступа.

- Обеспечение высокой производительности.

Типы объектов файловой системы

Первые файловые системы оперировали только с одним типом объектов – файлами. Логическая структура представляла собой список файлов с различными именами. Появление понятий каталог и дерево каталогов обеспечило реализацию удобного механизма уникального именования файлов и использования при хранении иерархической структуры. Данные свойства позволяют размещать в файловой системе имена (и, возможно, содержимое) объектов, отличных от файлов и директорий. Примерами таких объектов являются:

- мягкие ссылки (ярлыки) – объекты, при выполнении над которыми ряда операций производится перенаправление операции на другой объект файловой системы;

- файлы устройств – при поступлении запроса на выполнение операции с таким объектом, вызывается функция драйвера некоторого устройства, связанного с данным объектом (связь устанавливается при создании объекта);

- файлы межпроцессного взаимодействие – при организации межпроцессного взаимодействия операционная система в качестве идентификатора системного объекта (например, мьютекса или канала) может использовать имя объекта файловой системы;

выполнение операций над данным объектом обслуживает непосредственно подсистема организации межпроцессного взаимодействия в соответствии с логикой используемого объекта.

Существуют и другие типы объектов файловой системы, однако мы будем рассматривать только два – файлы и директории.

Имена объектов файловой системы

Файлы - абстрактные объекты. Они предоставляют пользователям возможность сохранять информацию, скрывая от него детали того, как и где она хранится и то, как диски в действительности работают. Вероятно, одна из наиболее важных характеристик любого абстрактного механизма - способ именования объектов, которыми он управляет. Когда процесс создает файл, он дает файлу имя. После завершения процесса файл продолжает существовать и через свое имя может быть доступен другим процессам.

Обычно ОС накладывают некоторые ограничения:

- на используемые в имени символы;

- на длину имени;

- на длину полного имени.

Например, в ОС Unix учитывается регистр символов в имени файла (case sensitive), а в MS-DOS - нет. В популярной файловой системе FAT длина имен ограничивается известной схемой 8.3 (8 символов - собственно имя, 3 символа - расширение имени). Современные файловые системы, как правило, поддерживают более удобные для пользователя длинные символьные имена файлов. Так, в соответствии со стандартом POSIX допускаются имена длиной до 255 символов, та же самая длина устанавливается для имен файлов и в ОС Windows NT/2000 для файловой системы NTFS.

Операции над файлами

Операционная система должна предоставить в распоряжение пользователя набор операций для работы с файлами, реализованных через системные вызовы. Чаще всего при выполнении пользователем одной операции операционная система выполняет несколько действий. Например, находит данные файла и его атрибуты по его символьному имени; затем считывает необходимые атрибуты файла в отведенную область оперативной памяти и анализирует права пользователя на выполнение требуемой операции; и, наконец, выполняет операцию, после чего освобождает занимаемую данными файла область памяти.

Перечислим основные операции над файлами.

- Create. Создание файла, не содержащего данных. Смысл данного вызова - объявить, что файл существует и присвоить ему ряд атрибутов.

- Delete. Удаление файла и освобождение занятого им дискового пространства.

- Open. Перед использованием файла процесс должен его открыть. Цель данного системного вызова – разрешить системе проанализировать атрибуты файла и проверить права доступа к файлу, считать в оперативную память список адресов блоков файла для быстрого доступа к его данным и организовать структуры для поддержки выполнения процессом операций над файлом.

- Close. При завершении работы с файлом, его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл нужно закрыть, чтобы освободить место во внутренних таблицах системы управления файлами.

- Seek. Дает возможность специфицировать место внутри файла, откуда будет производиться следующее считывание (или запись) данных, то есть задать текущую позицию.

- Read. Чтение данных из файла. Обычно это происходит с текущей позиции. Процесс должен задать объем считываемых данных и предоставить буфер для них.

- Write. Запись данных в файл с текущей позиции. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

- Get attributes. Предоставляет процессам нужные им сведения об атрибутах файла. В качестве примера можно привести, утилиту make, которая использует информацию о времени последней модификации файлов.

- Set attributes. Дает возможность процессу установить некоторые атрибуты. Наиболее очевидный пример - установка режима доступа к файлу.

- Rename. Возможность переименования файла создает дополнительные удобства для пользователя. Данная операция может быть смоделирована копированием данного файла в файл с новым именем и последующим удалением оригинала.

Директории – логическая структура файлового архива

Количество файлов на компьютере может быть большим. Многие системы хранят сотни тысяч файлов, занимающие многие гигабайты дискового пространства. Эффективное управление этими данными подразумевает наличие в них четкой логической структуры. Все современные файловые системы поддерживают многоуровневое именование файлов за счет поддержания во внешней памяти объектов со специальной структурой - каталогов (или директорий).

Каждый каталог содержит список каталогов и/или файлов и/или объектов файловой системы других типов, содержащихся в данном каталоге. Каталоги имеют один и тот же внутренний формат, где каждому объекту файловой системы соответствует одна запись в файле директории. Таким образом, файлы на диске образуют иерархическую древовидную структуру.

Внутри одного каталога имена листовых файлов уникальны. Имена файлов, находящихся в разных каталогах могут совпадать. Для того чтобы однозначно определить файл по его имени (избежать коллизии имен) принято именовать файл полным именем (pathname), которое состоит из списка имен вложенных каталогов, по которому можно найти путь от корня к файлу, плюс имя файла в каталоге, непосредственно содержащем данный файл. Таким образом, имя включает цепочку имен - путь к файлу, например /usr/ast/mailbox . Это так называемое абсолютное имя. Такие имена уникальны. Компоненты пути разделяют символами '/' (слеш) в Unix или обратными слешами '\' в Windows.

Другой способ задания имени - относительный путь к файлу. Он использует концепцию рабочей или текущей директории процесса, работающего с данным файлом (например, в Linux рабочая директория является частью структуры данных процесса). Тогда к файлам в такой директории можно ссылаться только по имени, при этом поиск файла будет осуществляться в рабочей директории. Это удобнее, но по существу то же самое, что и абсолютная форма.

Операции над директориями

Так же, как и в случае файлов, система обязана обеспечить пользователя набором операций, необходимых для работы с директориями, реализованных через системные вызовы.

Несмотря на то, что директории – это тоже файлы (то есть именованные области данных на внешнем носителе), логика работы с ними отличается от логики работы с обычными файлами и определяется природой этих объектов, предназначенных поддерживать структуру файлового архива. Совокупность системных вызовов для управления директориями зависит от особенностей конкретной ОС. Рассмотрим в качестве примера некоторые системные вызовы ОС Unix.

- Create. Создание директории. Вновь созданная директория включает записи с именами '.' и '..', однако считается пустой.

- Delete. Удаление директории. Удалена может быть только пустая директория.

- Opendir. Открытие директории для последующего чтения. Например, чтобы перечислить файлы, входящие в директорию, процесс должен открыть директорию и считать имена всех файлов, которые она включает.

- Closedir. Закрытие директории после ее чтения для освобождения места во внутренних системных таблицах.

- Readdir. Данный системный вызов возвращает содержимое текущей записи в открытой директории. Вообще говоря, для этих целей может быть использован системный вызов Read, но в этом случае от программиста потребуется знание внутренней структуры директории. Readdir возвращает содержимое записи в стандартном формате, независимо от используемой структуры хранения содержимого директорий.

- Rename. Имена директорий можно менять, так же как и имена файлов.

- Link. Связывание - это техника, которая позволяет информации о файле появляться более чем в одной директории. Данный системный вызов позволяет задать для одного блока данных несколько имен (имен файлов), которые могут располагаться в различных каталогах.

- Unlink. Удаление записи о файле из директории. Если удаляемый файл присутствует только в одной директории, то он вообще удаляется из файловой системы, в противном случае система ограничивается только удалением специфицируемой записи.

Имеется также ряд других системных вызовов, например, связанных с защитой информации.

Пользовательский интерфейс системы управления файлами

Для организации хранения информации на диске пользователь вначале обычно выполняет его форматирование, выделяя на нем место для структур данных, которые описывают состояние файловой системы в целом. Затем пользователь создает нужную ему структуру каталогов (или директорий), которые по существу являются списками вложенных каталогов и собственно файлов. И, наконец, заполняет дисковое пространство файлами, приписывая их тому или иному каталогу. Таким образом, ОС должна предоставить в распоряжение пользователя совокупность сервисов, традиционно реализованных через системные вызовы, которые обеспечивают:

- создание файловой системы на диске (форматирование);

- необходимые операции для работы с каталогами;

- необходимые операции для работы с файлами.

Кроме того, файловые службы могут решать проблемы проверки и сохранения целостности файловой системы, проблемы повышения производительности и ряд других.

Далее мы будем рассматривать архитектуру файловой системы применительно к UNIX-системам, поскольку в основе архитектур файловых систем различных ОС лежат схожие идеи.

Виртуальная Файловая Система

Виртуальная файловая система (Virtual File System) - это дополнительный уровень абстрагирования, позволяющий осуществлять единообразную работу с различными файловыми системами – ext3, Minix, FAT и т.д. Это прослойка между процессом и настоящей файловой системой.

Принцип работы

Выше мы перечислили набор операций для работы с файлами и директориями. Семантика операций из данного набора может отличаться в зависимости от типа файловой системы (например, для операций по управлению доступом к файлу). Однако большинство запросов обслуживаются одинаковым способом при работе с различными файловыми системами.

При поступлении запроса на работу с объектом файловой системы, он обслуживается кодом виртуальной файловой системы, содержащимся в ядре ОС. Этот код анализирует запросы, определяет местоположение используемого объекта и тип файловой системы, которой он принадлежит, и вызывает необходимые функции модуля работы с соответствующей файловой системой для выполнения операции ввода/вывода. Такой механизм работы с файлами используется для упрощения объединения и использования нескольких типов файловых систем.

Когда какой-либо процесс выдает системный вызов, связанный с работой файловой системы, ядро вызывает функцию, расположенную в коде виртуальной файловой системы (VFS). Эта функция производит действия, независимые от структуры файловой системы, определяет местоположение используемого объекта и тип файловой системы, которой он принадлежит, и перенаправляет вызов к функции драйвера этой файловой системы (код, реализующий операции с файловой системой), который выполняет операции, связанные с ее структурой и выполнением операций ввода/вывода.

Структура VFS

VFS не ориентируется на какую-либо конкретную файловую систему, механизмы реализации файловой системы полностью скрыты как от пользователя, так и от приложений. В ОС нет системных вызовов, предназначенных для работы со специфическими типами файловой системы, а имеются абстрактные вызовы типа open, read, write и другие, которые имеют содержательное описание, обобщающее некоторым образом содержание этих операций в наиболее популярных типах файловых систем (например, ext3, ufs, nfs и т.п.). VFS также предоставляет ядру возможность оперирования файловой системой, как с единым целым: операции монтирования и демонтирования, а также операции получения общих характеристик конкретной файловой системы (размера блока, количества свободных и занятых блоков и т.п.) в единой форме (**монтирование** – включение логической структуры файловой системы некоторого отдельного внешнего устройства в общую логическую структуру, в UNIX – «подвешивание» дерева некоторой файловой системы к некоторой «ветке»-каталогу единого дерева). Если конкретный тип файловой системы не поддерживает какую-то абстрактную операцию VFS, то файловая система должна вернуть ядру код возврата, извещающий об этом факте.

Интерфейс VFS состоит из ряда операций, которые оперируют тремя типами объектов: файловые системы, индексные дескрипторы и открытые файлы.

VFS содержит информацию о всех типах поддерживаемых файловых систем. Здесь используется таблица, которая создается во время компиляции ядра ОС. Каждая запись в такой таблице содержит тип файловой системы: она включает в себя наименование типа и указатель на функцию, вызываемую во время монтирования этой файловой системы. При монтировании файловой системы вызывается соответствующая функция монтирования. Эта функция используется для считывания суперблока (блок данных, описывающих параметры файловой системы), установки внутренних переменных и возврата дескриптора смонтированной системы в VFS. После того, как система смонтирована, функции VFS используют этот дескриптор для доступа к процедурам используемой файловой системы.

Дескриптор смонтированной файловой системы содержит в себе некоторую информацию, которая одинакова для каждого типа файловой системы – указатели на функции, используемые для выполнения операций на данной файловой системе и некоторые данные, используемые этой системой. Указатели на функции, расположенные в дескрипторе файловой системы, позволяют VFS получить доступ к внутренним функциям файловой системы.

В VFS используются еще два типа дескрипторов: это inode (индексный дескриптор) и дескриптор открытого файла (блок управления файлом). Каждый из них содержит информацию, связанную с используемыми файлами. Индексный дескриптор (inode) содержит указатели к функциям, используемым по отношению к любому файлу (например, create или unlink). Дескриптор файлов содержит указатели к функциям, оперирующим только с открытыми файлами (например, read или write).

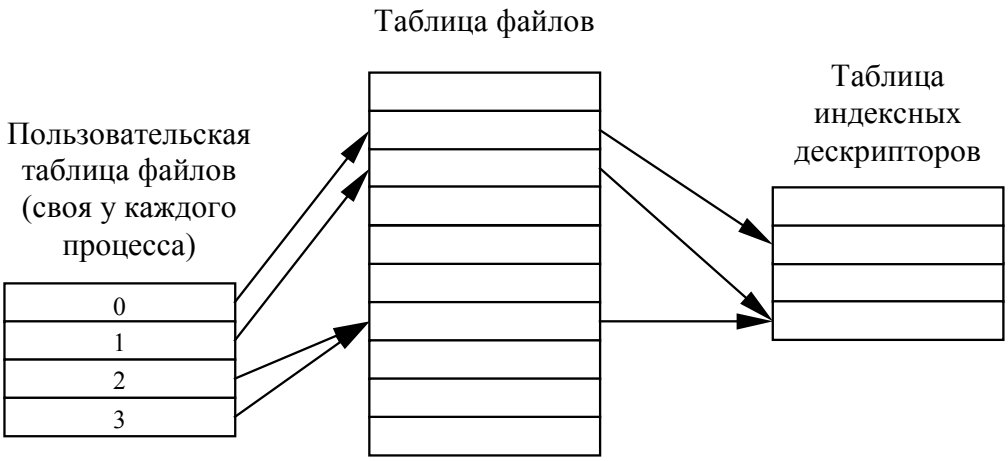


Рис. 44 Классическая связь между таблицами, используемыми при работе с файлами

Процесс при формировании запроса на выполнение операции над файлом использует handle (индекс), полученный после открытия файла. Из таблицы файлов извлекаются параметры выполнения операции (например, смещение в файле, начиная с которого нужно выполнить операцию чтения). Таблица индексных дескрипторов предоставляет информацию о расположении файла на диске, его атрибутах и другую информацию, зависящую от типа файловой системы.

В VFS вся информация индексных дескрипторов разделена на две части - не зависящую от типа файловой системы, которая хранится в специальной структуре ядра - структуре vnode, и зависящую от типа файловой системы - структура inode, формат которой на уровне VFS не определен, а используется только ссылка на нее в структуре vnode. Имя inode не означает, что эта структура совпадает со структурой индексного дескриптора inode файловой системы UNIX (например, ext3). Это имя используется для обозначения зависящей от типа файловой системы информации о файле, как дань традиции.

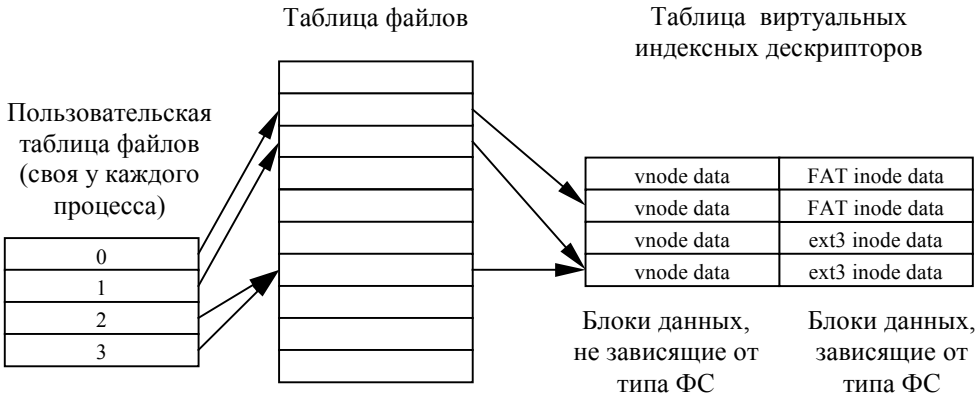


Рис. 45 Связь между таблицами, используемыми при работе с файлами в VFS

Типы объектов в VFS

Виртуальная файловая система VFS поддерживает следующие типы файлов:

- обычные файлы;

- каталоги;

- специальные файлы;

- именованные конвейеры (именованные каналы);

- символьные связи (мягкие ссылки).

Символьные связи (мягкие ссылки)

Мягкая связь, называемая символьной связью, реализуется с помощью системного вызова symlink. Символьная связь – это файл данных, содержащий имя файла, с которым предполагается установить связь. Слово "предполагается" использовано потому, что символьная связь может быть создана даже с несуществующим файлом. При создании символьной связи образуется как новый вход в каталоге, так и новый индексный дескриптор inode. Кроме этого, резервируется отдельный блок данных для хранения полного имени файла, на который он ссылается.

Многие системные вызовы пользуются файлом символьных связей для поиска реального файла. Связанные файлы не обязательно располагаются в той же файловой системе.

Имеются три системных вызова, которые имеют отношение к символьным связям:

- readlink - чтение полного имени файла или каталога, на который ссылается символьная связь. Эта информация хранится в блоке, связанном с символьной связью.

- lstat - аналогичен системному вызову stat, но используется для получения информации о самой связи.

- lchown - аналогичен системному вызову chown, но используется для изменения владельца самой символьной связи.

Именованные конвейеры (именованные каналы)

Конвейер – это средство обмена данными между процессами. Конвейер буферизует данные, поступающие на его вход, таким образом, что процесс, читающий данные на его выходе, получает их в порядке "первый пришел - первый вышел" (FIFO). Именованные конвейеры позволяют обмениваться данными произвольной паре процессов, т.к. каждому такому

конвейеру соответствует файл на диске. Никакие данные не связываются с файлом-конвейером, но в каталоге содержится запись о нем, и он имеет индексный дескриптор.

Реализация VFS

Для множества типов конкретных файловых систем заводится набор структур, содержащих:

- символьное имя типа файловой системы;

- указатель на функцию инициализации файловой системы;

- указатель на структуру, описывающую функции, реализующие абстрактные операции VFS в данной конкретной файловой системе.

Функции инициализации файловых систем вызываются во время инициализации операционной системы. Эти функции ответственны за создание внутренней среды файловой системы каждого типа.

Например, в UNIX System V Release 4 предусмотрено 7 абстрактных операций над файловой системой:

VFS_MOUNT - монтирование файловой системы,

VFS_UNMOUNT - размонтирование файловой системы,

VFS_ROOT - получение vnode для корня файловой системы,

VFS_STATVFS - получение статистики файловой системы,

VFS_SYNC - выталкивание буферов файловой системы на диск,

VFS_VGET - получение vnode по номеру дескриптора файла,

VFS_MOUNTROOT - монтирование корневой файловой системы;

и следующее множество абстрактных операций над файлами:

VOP_OPEN - открыть файл,

VOP_CLOSE - закрыть файл,

VOP_READ - читать из файла,

VOP_WRITE - записать в файл,

VOP_IOCTL - управление вводом-выводом,

VOP_SETFL - установить флаги статуса,

VOP_GETATTR - получить атрибуты файла,

VOP_SETATTR - установить атрибуты файла,

VOP_LOOKUP - найти vnode по имени файла,

VOP_CREATE - создать файл,

VOP_REMOVE - удалить файл,

VOP_LINK - связать файл,

VOP_MAP - отобразить файл в память.

Повторим, что при работе с VFS приходится поддерживать два вида индексных дескрипторов файлов – на уровне VFS и на уровне конкретного типа файловой системы.

Описание структур, используемых в Linux, дано в третьем разделе.

Лабораторная работа 1. Краткосрочное планирование задач

Цель работы – реализация одного из алгоритмов диспетчеризации.

Данная работа предполагает два варианта исполнения:

а) создание программного блока, реализующего диспетчеризацию, для симулятора;

б) модификация исходных кодов ядра UNIX, относящихся к подсистеме планировщика с последующей их компиляцией и установкой полученного ядра. (Здесь и далее предполагается выполнение работ с использованием Linux RedHat 7.3, версия ядра - 2.4.20)

Постановка задачи для конкретного студента включает выбор варианта исполнения и реализуемого алгоритма диспетчеризации.

В ходе выполнения лабораторной работы студент должен решить следующие задачи.

1. Изучить архитектуру планировщика и его место в общей архитектуре.

2. Освоить понятия описателей (дескрипторов) процессов и потоков.

3. Определить обрабатываемые планировщиком события.

4. Модифицировать структуры описателей (дескрипторов) процессов и потоков в соответствие с заданным алгоритмом диспетчеризации.

5. Модифицировать обработчики целевых событий.

6. Выполнить компиляцию и сборку симулятора или ядра ОС UNIX.

7. Провести сравнительные эксперименты с использованием модулей, в которых реализованы различные алгоритмы диспетчеризации.

Симулятор многозадачной системы

Лабораторные работы 1 и 2 предполагают вариант исполнения с использованием программной лаборатории - симулятора многозадачной системы (симулятора диспетчеризации и замещения страниц), работа которого основывается на понятии **эксперимента**.

Модель эксперимента

Симулятор диспетчеризации и замещения страниц представляет собой программную систему имитации выполнения нескольких задач на однопроцессорной машине. Модель имитации включает следующие положения и допущения.

1. В системе в произвольные моменты времени поступают запросы на создание новых процессов. Интервал поступления новых запросов является случайной величиной пуассоновского типа, имеющей один управляющий параметр - **усредненное время между поступлениями запросов на создание процесса**. Формула распределения вероятностей: $p_k = l^k / k! * \exp(-l)$, $k=0,1,2,...$, где l - усредненное время между поступлениями запросов.

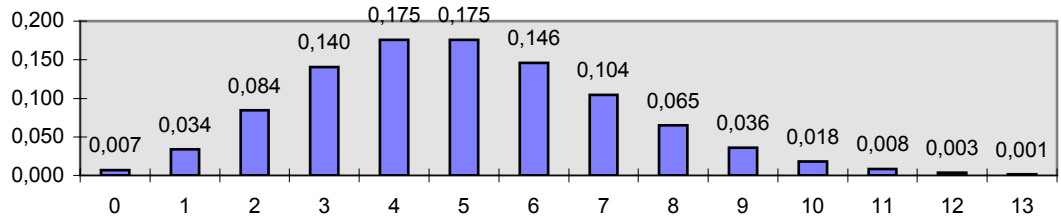


Рис. 46 Пример распределения вероятностей при значении параметра "усредненный интервал между запусками процессов" равном 5

2. Одновременно система может обслуживать не более некоторого **максимального числа процессов**. Процесс, для которого поступила заявка на создание, обслуживается только в том случае, если число работающих процессов меньше максимального. В противном случае, его обслуживание начинается только после завершения одного из активных процессов. Если ожидающих обработки процессов больше одного, их обслуживание производится в порядке поступления запросов на выполнение.

3. Каждый процесс может иметь несколько потоков. Число потоков является случайной величиной биномиального типа, имеющей два управляющих параметра - **максимальное число потоков одного процесса** и **среднее число потоков одного процесса**. Формула распределения вероятностей: $p_k = C^k_n * (l/n)^k * (1-(l/n))^{n-k}$, k=0,1,2,..,n , где n - максимальное число потоков одного процесса, l - среднее число потоков одного процесса.

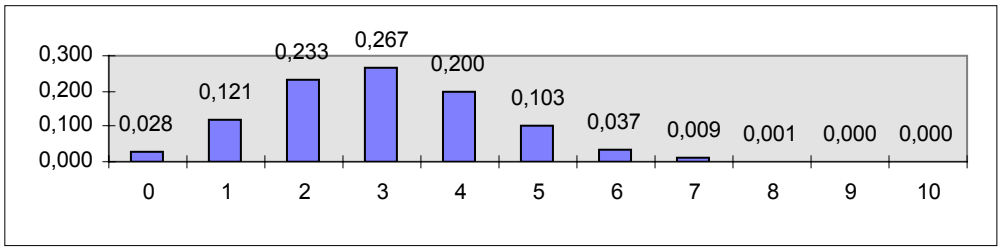


Рис. 47 Пример распределения вероятностей при значении параметров n=10, k=3

Если при вычислении вероятности получилось значение 0, процесс имеет один поток.

4. Каждый поток для завершения работы требует некоторого времени центрального процессора. Время работы потока является равномерной случайной величиной и определяется максимальным временем выполнения потока. Формула распределения вероятностей: $p_k = 1 / n$, k=1,...,n.

5. Предполагается, что при работе потока вычисления чередуются с выполнением операций ввода-вывода. При этом для выполнения очередной порции вычислений требуется, чтобы поток потребил определенное количество времени центрального процессора; для выполнения операций ввода-вывода требуется прохождение некоторого количества времени, и не имеет значения, какой поток или потоки выполнялись в течение этого времени на центральном процессоре.

Таким образом, для каждого потока формируется диаграмма исполнения, состоящая из последовательности участков вычислений, разделяемых участками ожидания завершения операций ввода-вывода.

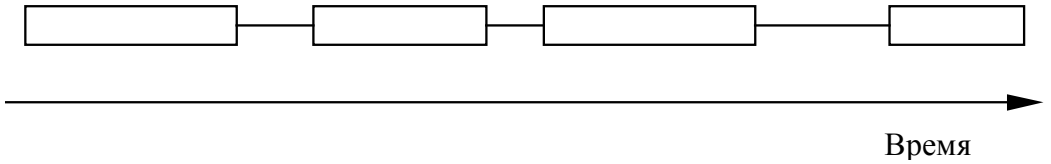


Рис. 48 Диаграмма выполнения потока

Диаграмма формируется на основании предположения, что время исполнения и ожидания - пуассоновские случайные величины, характеризуемые параметрами среднее **время выполнения вычислений (исполнения)** и **среднее время ожидания завершения ожидания**.

6. Основной поток процесса создает все остальные потоки. Время создания дочерних потоков - равномерная случайная величина на интервале времени потребления основным потоком центрального процессора.

7. Эксперимент продолжается некоторое заданное число секунд, каждая секунда делится на такты одинакового размера, расчеты всех времен выполняются в тактах. Такт - время выполнения одной команды процессора. Такты условно объединяются в кванты - с целью упрощения реализации некоторых алгоритмов диспетчеризации. Эксперимент продолжается в течение **времени эксперимента**.

8. Каждый процесс имеет виртуальное адресное пространство, состоящее из страниц размером 4 Кб. Размер ВАП определяется параметром **размер ВАП процесса** (в страницах). Размер физической памяти определяется параметром **размер физической памяти** (в страницах).

9. Каждый процесс содержит в своем адресном пространстве сегмент кода, сегмент данных и по одному сегменту стека на каждый поток. Распределение сегментов выполняется при создании процесса, каждый сегмент занимает минимум одну страницу. Общее число страниц, используемое процессом - равномерная случайная величина в интервале между значениями от “число сегментов” до “размер ВАП процесса”. Сегменты считаются одинаковыми по размеру (с точностью до 1 страницы).

10. Для каждого потока хранится адрес последней выполненной команды, адрес последних использованных данных и адрес вершины стека. При создании потока адрес последней выполненной команды и адрес последних использованных данных вычисляются по формуле равномерной случайной величины в сегменте кода и данных. Адрес вершины стека устанавливается в конец сегмента стека.

При переходе к следующей команде новые адреса вычисляются следующим способом:

- определяется, использует ли текущая команда стек или данные (или не использует) - вероятность каждого события 1/3;

- к адресу команды прибавляется 2, затем он дополнительно уменьшается или увеличивается (равновероятно); величина изменения является пуассоновской случайной величиной со значением параметра, равном 2;

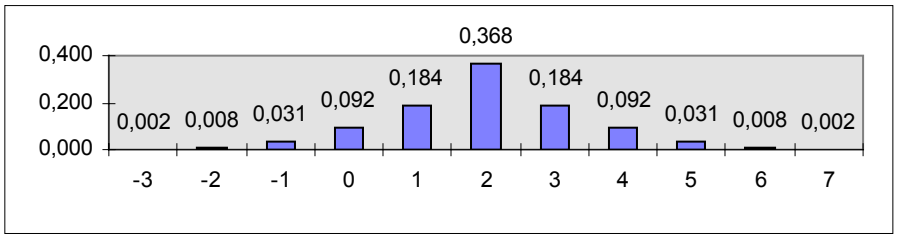


Рис. 49 Распределение вероятности изменения адреса команды

- если используется сегмент данных, определяется, уменьшается или увеличивается адрес данных (равновероятно); величина изменения является пуассоновской случайной величиной со значением параметра, равном 2;

- если используется сегмент стека, определяется, уменьшается или увеличивается адрес вершины стека (равновероятно); величина изменения является пуассоновской случайной величиной со значением параметра, равном 2;

- если какой-либо адрес вышел за пределы сегмента, для него вычисляется случайное значение в пределах сегмента по формуле равномерной случайной величины.

Данные предположения основываются на алгоритме функционирования стека и принципе локальности для кода и данных.

После задания параметров эксперимента производится формирование диаграммы запуска процессов, диаграмм работы всех потоков и используемых ими страниц ВАП процесса и т.д. Указанное множество данных, полученное после вычисления вероятностных характеристик, мы будем называть **экспериментом**.

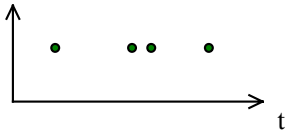
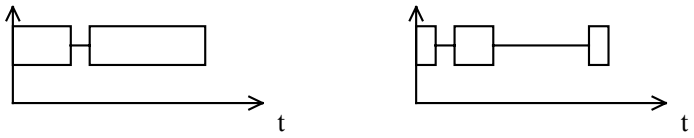
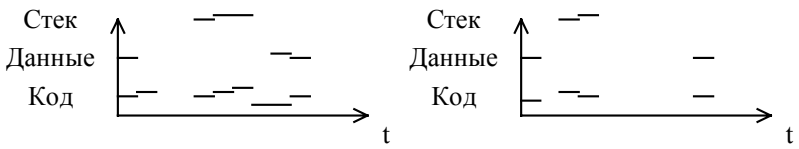


Диаграмма запуска процессов



Диаграммы исполнения потоков



Диаграммы использования потоками страниц ВАП процессов

Рис. 50 Сформированные данные эксперимента

Считается, что в начале эксперимента физическая память свободна. После запуска эксперимента начинается отсчет тактов времени. Планировщик может обрабатывать следующие события:

- создание процесса;

- создание потока;

- уничтожение процесса;
- уничтожение потока;
- переход активного потока в состояние ожидания;
- завершение ожидания какого-либо потока;
- начало такта;
- начало кванта.

Блок определения замещаемой страницы вступает в работу при загрузке страницы в оперативную память, при определении, какую страницу необходимо вытеснить из памяти, и при выгрузке страницы из памяти.

Состояние системы в каждый конкретный момент описывается совокупностью следующих значений:

- вектор дескрипторов работающих процессов;
- вектор дескрипторов работающих потоков;
- вектор описателей страниц физической памяти.

В результате выполнения эксперимента вычисляется ряд статистических значений, соответствующих критериям оценки алгоритмов планирования, и среднее число страничных сбоев в секунду. Изменяя какой-либо один из параметров эксперимента, можно выполнить сравнительный анализ работы алгоритмов при различных значениях максимального числа процессов, частоты их появления, соотношения размера ВАП и оперативной памяти и т.д.

Архитектура программной лаборатории

Программная лаборатория представляет собой стандартное оконное приложение и обеспечивает удобный пользовательский интерфейс, позволяющий в диалоговом режиме задавать исходные данные решаемой задачи, наблюдать полученные результаты и сохранять их.

Логическая структура разрабатываемой программной лаборатории представляет собой совокупность следующих функциональных блоков:

1. Блок визуализации – отвечает за интерфейс программного комплекса.

2. Блок симуляции – отвечает за работу процесса симуляции.

3. Блок планирования – отвечает за выбор текущего процесса (потока) выбранным методом диспетчеризации.

4. Блок замещения страниц – отвечает за замещение страницы из памяти выбранным методом, если это необходимо.

5. Блок статистики – отвечает за накопление и анализ результатов эксперимента.

Общая структура программной лаборатории иллюстрируется на следующей схеме.

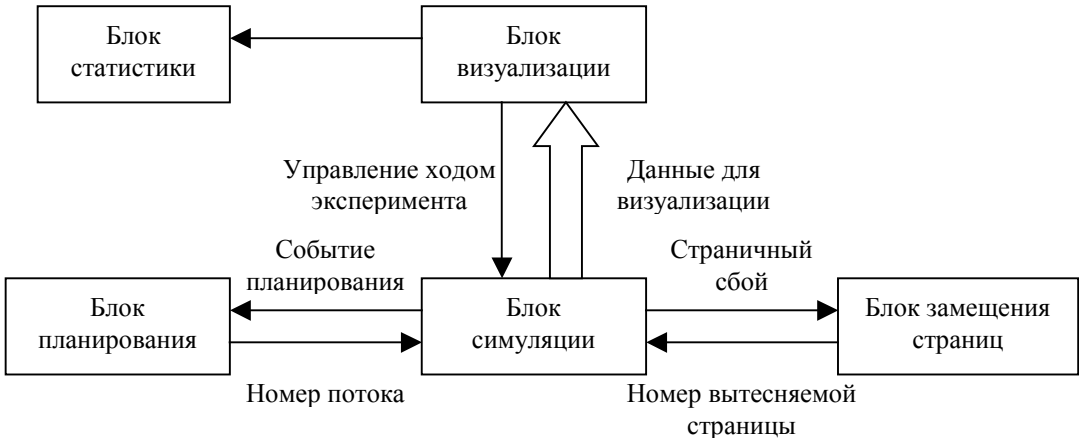


Рис. 51 Архитектура симулятора многозадачной системы

Блоки 1-4 выполняют следующие действия.

1. Блок визуализации.

- задание параметров;
- сохранение/загрузка эксперимента;
- управление процессом симуляции;
- визуализация процесса симуляции;
- визуализация результатов (статистика);
- сохранение результатов.

2. Блок симуляции.

- инициализация эксперимента;
- выполнение основного цикла по времени эксперимента;
- проверка наступления событий планирования и замещения страниц;
- обновление изображения;
- сбор статистики;
- завершение эксперимента.

3. Блок планирования.

- инициализация эксперимента (инициализация планировщика в системе);
- выполнение одиночной операции планирования по наступлении различных моментов планирования;
- деинициализация эксперимента.

4. Блок замещения страниц.

- инициализация эксперимента;
- выполнение одиночной операции замещения страниц;
- деинициализация эксперимента.

Проведение эксперимента

Для проведения эксперимента необходимо указать параметры эксперимента и сформировать данные эксперимента, нажав на кнопку «Сформировать эксперимент». Эксперимент также может быть сохранен и впоследствии загружен (для проведения нескольких испытаний над одним набором данных).

После формирования эксперимента нужно выбрать используемые алгоритмы планирования и замещения страниц. В программной лаборатории реализованы алгоритмы планирования «случайный выбор потока» и «карусельное планирование» (Round Robin), алгоритмы замещения страниц «случайный» и «FIFO».

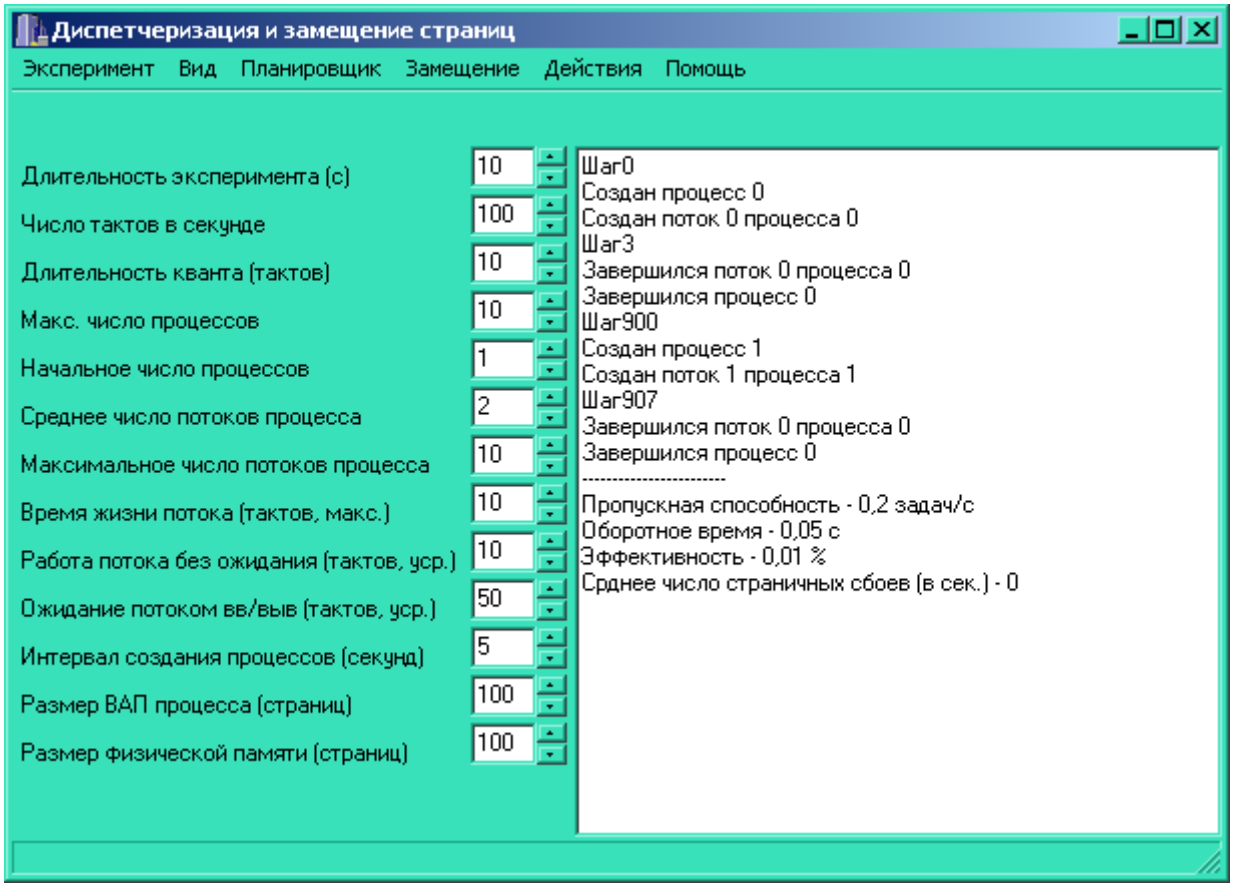


Рис. 52 Окно программы – симулятора многозадачной системы

В ходе эксперимента в журнал будет записываться информация о моментах планирования и принятых решениях и о произошедших страничных сбоях. По окончании эксперимента выводятся статистические данные.

Выполнение лабораторной работы

Выполнение работы подразумевает реализацию одного из алгоритмов планирования.

Программная лаборатория выполнена в среде разработки Borland C++ Builder 5.0. Для выполнения лабораторной работы предоставляются исходные тексты лаборатории, за исключением реализованных алгоритмов (они представлены в виде библиотек).

Описание используемых структур данных и назначение функций и процедур, используемых в коде, содержится в имеющемся описании программы.

Архитектура планировщика в Linux (Ядро 2.4.18)

Планировщик Linux реализует планирование на основе динамических приоритетов. Данный подход основывается на использовании некоторого базового приоритета, на основании которого формируется эффективный приоритет, зависящий от состояния планировщика, процесса и всей системы. Например, если процесс является ограниченным вводом-выводом, имеет эффективный приоритет выше, чем базовый. Напротив, процесс, полностью использующий предоставляемые ему кванты времени центрального процессора, имеет пониженный эффективный приоритет.

Ядро Linux поддерживает два независимых диапазона приоритетов. Первый используется для процессов разделения времени – nice value, число от -20 до 19 со значением по умолчанию равным 0. Большие значения nice соответствуют меньшему приоритету. Процессы с меньшим значением nice выполняются перед процессами с низким значением. Значение nice также используется при определении размера кванта времени, предоставляемого процессу – процессы с меньшим значением nice получают большие кванты.

Второй диапазон - приоритеты реального времени. Они имеют значения от 0 до 99. Все процессы реального времени являются более приоритетными, чем процессы разделения времени.

Linux поддерживает две дисциплины обслуживания процессов реального времени – FIFO и Round Robin (SCHED_FF и SCHED_RR соответственно). Для остальных работает специальная дисциплина SCHED_OTHER, которая рассмотрена ниже. Процессы SCHED_FF исполняются до тех пор, пока они не выполнят системный вызов освобождения процессора, и для них не определяется размер кванта. Для процессов SCHED_RR квант определяется, и в рамках одного значения приоритета происходит их поочередное выполнение.

Для обеих дисциплин реального времени используются статические приоритеты, то есть неизменяемые планировщиком по своему усмотрению. Диапазон приоритетов реального времени – от 0 до MAX_RT_PRIO – 1 (по умолчанию, 100 – 1 = 99). Приоритеты процессов разделения времени – от MAX_RT_PRIO до MAX_PRIO = MAX_RT_PRIO+40, что по умолчанию составляет диапазон от 100 до 139. Значение приоритета процесса разделения времени вычисляется как MAX_RT_PRIO + nice + 20.

Квант – числовое значение, определяющее, как долго процесс может выполняться до того, как он будет вытеснен. Планировщик должен динамически определять размеры квантов. Использование слишком больших квантов ухудшает время отклика процессов, слишком малых – приводит к потерям времени на переключение контекстов. Планировщик Linux вычисляет размер кванта на основании приоритета процесса.

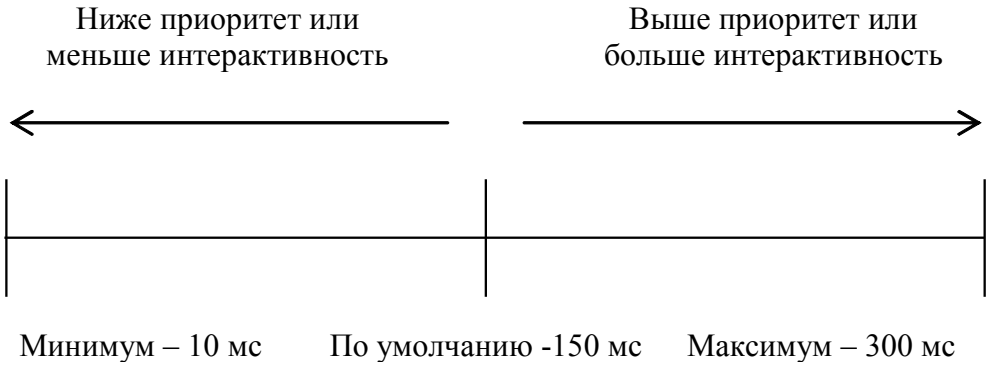


Рис. 53 Размер кванта, выделяемого процессу

Отметим, что процесс нет должен использовать весь квант сразу. Он может выполнить одно действие на 100 мс или 5 действий по 20 мс, отказываясь от центрального процессора в конце каждого действия.

Когда процесс использовал весь предоставленный ему квант, он считается истекшим (expired). Он не получит центральный процессор до тех пор, пока остальные готовые к исполнению процессы не исчерпают свои кванты. После этого будет произведен перерасчет размеров квантов для всех процессов.

Планировщик оперирует понятием jiffie – это время между двумя последовательными прерываниями таймера. Квант содержит несколько jiffie.

Реализация планировщика содержится в файле kernel/sched.c

Очередь процессов

Основная структура данных планировщика – **очередь процессов** (**runqueue**) – список процессов, готовых к исполнению на конкретном процессоре. Каждый готовый к исполнению процесс находится ровно в одной очереди. Очередь процессов также содержит параметры планирования для данного процессора. Она имеет следующую структуру.

```
struct runqueue {
    spinlock_t     lock;          /* признак блокировки очереди */
    unsigned long  nr_running;     /* число процессов в очереди */
    unsigned long  nr_switches;
                                /* число переключений контекста */
    unsigned long  expired_timestamp;
                             /* время последней смены массивов */
    struct task_struct *curr;          /* исполняющийся процесс */
    struct task_struct *idle;
                            /* idle-процесс данного процессора */
    struct prio_array  *active;
         /* указатель на массив приоритетов активных процессов */
    struct prio_array  *expired;
         /* указатель на массив приоритетов истекших процессов */
    struct prio_array  arrays[2];       /* массивы приоритетов */
    int         prev_cpu_load[NR_CPUS];
                                /* загрузка каждого процессора */
}
```

Для работы с очередями процессов определена группа макросов, например:

`cpu_rq(cpu)` – возвращает указатель на очередь указанного процессора;

`this_rq()` – возвращает указатель на очередь текущего процессора;

`task_rq(p)` – возвращает указатель на очередь, которой принадлежит указанный процесс.

Прежде, чем производить операции над очередью, ее необходимо заблокировать, временно закрепив за собой исключительное право выполнения операций над ней. Для этого могут использоваться, например, функции

```
static inline runqueue_t *lock_task_rq( task_t *p, ulong flags );
static inline void lock_task_rq(runqueue_t *rq, ulong flags );
```

Для избежания взаимоблокировок, в случае использование нескольких очередей, они должны блокироваться в порядке возрастания адресов.

Каждая очередь процессов содержит два массива приоритетов – активный и истекший. Каждый из этих массивов содержит один список процессов, готовых к исполнению, на каждый уровень приоритета. Массив приоритетов также содержит битовую маску, в которой содержатся признаки наличия процессов с каждым уровнем приоритета.

```
struct prio_array {
    int        nr_active;          /* число активных процессов */
    spinlock_t *lock;/* указатель на признак блокировки очереди */
    runqueue_t *rq;               /* очередь – владелец массива */
    unsigned long   bitmap[BITMAP_SIZE];      /* битовый массив */
    list_t     queue[MAX_PRIO];        /* списки по приоритетам */
};
```

MAX_PRIO – максимальное число приоритетов в системе (по умолчанию, 140). Массив приоритетов содержит битовый массив, в котором отведен бит для каждого уровня приоритета. Изначально, все биты равны 0. Когда процесс с некоторым уровнем приоритета переходит в состояние «готов к исполнению», соответствующий бит устанавливается в 1. Таким образом, поиск процесса с наивысшим приоритетом сводится к поиску первого бита в данном массиве (sched_find_first_bit()) и взятию первого элемента из соответствующего списка.

Кванты времени центрального процессора

Множество операционных систем, включая ранние версии Linux, производят вычисление размеров квантов следующим образом:

```
for (each task on the system) {
    recalculate priority
    recalculate timeslice
}
```

Такой подход потенциально может занимать много времени, и может привести к блокированию доступа к дескрипторам процессов на значительное время (блокировка доступа к дескрипторам необходима).

Описываемая версия планировщика не требует подобного цикла пересчета. Вместо этого, поддерживаются два массива приоритетов для каждого процессора: активный и истекший. Активный массив содержит процессы, еще не использовавшие полностью предоставленный им квант времени, истекший – исчерпавшие свой квант. Когда размер кванта процесса уменьшается до 0, производится перерасчет кванта, и процесс перемещается в истекший

массив. Когда в активном массиве не останется элементов, он просто меняется местами с пассивным (см. `void schedule(void)`):

```
struct prio_array array = rq->active;
if (!array->nr_active) {
    rq->active = rq->expired;
    rq->expired = array;
}
```

Выбор процесса на исполнение

Выбор процесса, которому будет предоставлено время центрального процессора, выполняется в функции schedule(). Данная функция вызывается непосредственно ядром в случае, если произошло одно из событий, требующих определения процесса, которому будет предоставлено время центрального процессора. Рассмотрим фрагменты кода.

```
asmlinkage void schedule(void)
{
     ...
/* Для процесса запоминается момент завершения его кванта */
     prev->sleep_timestamp = jiffies;
     /* При работе планировщика прерывания разрешены */
     spin_lock_irq(&rq->lock);
/* если задача находится в состоянии TASK_RUNNING, то она остается в этом состоянии; если задача находится в состоянии TASK_INTERRUPTIBLE и для нее поступили сигналы, то она переводится в состояние TASK_RUNNING. В любом другом случае задача удаляется из очереди runqueue */
     switch (prev->state) {
     case TASK_INTERRUPTIBLE:
          if (unlikely(signal_pending(prev))) {
               prev->state = TASK_RUNNING;
               break;
          }
     default:
          deactivate_task(prev, rq);
     case TASK_RUNNING:
          ;
     }
/* Если в очереди нет процессов, запускается idle-процесс*/
     if (unlikely(!rq->nr_running)) {
          next = rq->idle;
```

```
            rq->expired_timestamp = 0;
            goto switch_tasks;
      }
/* Если в активном массиве не осталось процессов, он меняется
местами с пассивным */
      array = rq->active;
      if (unlikely(!array->nr_active)) {
            rq->active = rq->expired;
            rq->expired = array;
            array = rq->active;
            rq->expired_timestamp = 0;
      }
/* Определяется процесс с наивысшим приоритетом */
      idx = sched_find_first_bit(array->bitmap);
      queue = array->queue + idx;
      next = list_entry(queue->next, task_t, run_list);
/* Производится переключение задач */
switch_tasks:
      prefetch(next);
      prev->need_resched = 0;

      if (likely(prev != next)) {
            rq->nr_switches++;
            rq->curr = next;
            context_switch(prev, next);
            barrier();
            rq = this_rq();
      }
      spin_unlock_irq(&rq->lock);
      ...
}
```

Вычисление эффективного приоритета и размера кванта

Значение nice number процесса хранится в поле static_prio структуры task_struct (дескриптора процесса). Планировщик формирует динамический приоритет и сохраняет его в поле prio. Эффективный приоритет вычисляется на основании значения nice процесса и эвристического анализа интерактивности процесса. Вычисление выполняет функция

```
static inline int effective_prio(task_t *p);
```

Является ли процесс интерактивным, определяется на основании анализа деятельности процесса – анализируется отношение времени, которое процесс проводит в состоянии ожидания (сна), ко времени, которое процесс проводит в состоянии исполнения. Для этого используется поле дескриптора sleep_avg, которое может принимать значения от 0 до MAX_SLEEP_AVG (по умолчанию, 20 мс). Когда процесс пробуждается, значение поля увеличивается на время сна. При потреблении каждого jiffie значение поля уменьшается на единицу.

Размер кванта определяется следующим макросом (HZ-число jiffie в секунде).

```
#define MIN_TIMESLICE         ( 10 * HZ / 1000)
#define MAX_TIMESLICE         (300 * HZ / 1000)
#define TASK_TIMESLICE(p) (MIN_TIMESLICE + \
((MAX_TIMESLICE-MIN_TIMESLICE)*(MAX_PRIO-1-(p)->static_prio)/39))
```

Таким образом, размер кванта пропорционален значению nice процесса.

Вытеснение процесса

С периодичностью, равной jiffie, происходят прерывания таймера. При этом выполняется вызов функции

```
void scheduler_tick(int user_tick, int system);
```

Данная функция определяет, исчерпал ли выполняющийся процесс выделенный ему квант времени (значение кванта активного процесса уменьшается на единицу с каждым вызовом функции). В случае, если квант исчерпан, он переводится из массива активных процессов в массив истекших.

Функция содержит дополнительную поддержку интерактивных процессов. Дело в том, что если завершился квант интерактивного процесса, планировщик поставит данный процесс в конец списка активного массива (практически, процесс получит еще один квант времени). Это реализуется посредством следующего участка кода.

```
if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
     if (!rq->expired_timestamp)
          rq->expired_timestamp = jiffies;
     enqueue_task(p, rq->expired); // !!!
} else
     enqueue_task(p, rq->active); // !!!
```

Данная функция также обновляет статистику использования процессора.

Изменение алгоритма планирования

Для изменения алгоритма планирования вы должны определить используемые структуры данных и модифицировать содержимое по крайней мере следующих функций.

```
static inline void activate_task(task_t *p, runqueue_t *rq);
/* обработка создания новой задачи */
```

```
static inline void deactivate_task(struct task_struct *p,
runqueue_t *rq);
/* обработка завершения задачи */
void scheduler_tick(int user_tick, int system);
/* обработка прерывания таймера */
asmlinkage void schedule(void);
/* выбор процесса на исполнение */
asmlinkage long sys_sched_yield(void);
/* добровольное освобождение процессом процессора */
void __init sched_init(void);
/* инициализация планировщика */
```

При этом для упрощения задачи рекомендуется использовать некоторое подмножество существующих достаточно сложных структур данных.

Например, для реализации алгоритма Round Robin достаточно использовать один из списков в массиве приоритетов.

Компиляция и установка ядра Linux

Вы можете воспользоваться любым способом компиляции и установки ядра. В данном параграфе приведена типичная последовательность действий, которая позволит Вам успешно выполнить рассматриваемую задачу. (Действия приведенной инструкции должны выполняться строго в указанном порядке.)

1. Зарегистрируйтесь под учетной записью root.

2. Проверьте наличие установленных исходных кодов ядра

rpm -q kernel-headers kernel-source make dev86

3. Если указанные пакеты установлены, перейдите к Шагу 4. Иначе, смонтируйте Red Hat Linux 7.x CD-ROM и выполните команды rpm -Uvh для инсталляции указанных RPM.

4. Перейдите в каталог исходных текстов

cd /usr/src/linux

5. Выполните настройку параметров компиляции (если это Вам нужно).

"make config" или "make menuconfig"

По окончании настройки не забудьте сохранить результат.

6. Скомпилируйте ядро

make bzImage modules

После выполнения данного шага, в каталоге /usr/src/linux/arch/i386/boot должен появиться файл "bzImage". Если все шаги компиляции были выполнены верно, и все настройки, выполненные в окне команды "make config", корректны, появится данный файл. Если указанного файла нет, вы где-то допустили ошибку, и вам придется остановиться и выполнить все шаги сначала.

7. Скопируйте ядро в то место, где обычно оно лежит, и настройте модули.

cp /usr/src/linux/arch/i386/boot/bzImage /boot/vmlinuz-my

cp /usr/src/linux/System.map /boot/System.map-my

make modules_install

8. Обеспечьте ядру возможность загрузиться (предполагается, что в качестве boot-менеджера Вы используете lilo)

mkinitrd /boot/initrd-my.img

Добавьте в файл /etc/lilo.conf следующие строки:

```
image=/boot/vmlinuz-my
        label=MyKernel
        initrd=/boot/initrd-my.img
        root=/dev/hda1
        read-only
```

Строчка root=/dev/hda1 зависит от конфигурации (hda1 – первый основной раздел IDE Primary Master жесткого диска)

/sbin/lilo –v

Если ошибок нет – все сделано правильно.

9. Перезагрузитесь

/sbin/reboot или /sbin/shutdown -r now.

10. В диалоге начальной загрузки (в ответ на приглашение "Boot:"), нажмите Tab (или Control-X, зависит от вашего загрузчика) и среди прочих ядер в списке вы увидите "MyKernel". Введите: MyKernel и нажмите Enter. Ваше новое ядро начнет загружаться.

Литература по лабораторной работе 1

1. Э. Таненбаум. Современные операционные системы. 2-е издание. СПб: Питер, 2002.

2. А.Я. Архангельский. Интегрированная среда разработки C++Builder 5. М: Бином, 2000.

3. Robert Love. Linux Kernel Development. SAMS, 2003.

4. David Rusling. The Linux Kernel (Электронный источник - www.linuxdoc.org/LDP/tlk/)

5. Cross-Referencing Linux (Электронный источник - lxr.linux.no)

Лабораторная работа 2. Замещение областей памяти

Цель работы – реализация одного из алгоритмов замещения страниц памяти.

Данная работа предполагает два варианта исполнения:

а) создание программного блока, реализующего замещение страниц, для симулятора;

б) модификация исходных кодов ядра UNIX, относящихся к подсистеме управления памятью с последующей их компиляцией и установкой полученного ядра.

Постановка задачи для конкретного студента включает выбор варианта исполнения и реализуемого алгоритма замещения страниц.

В ходе выполнения лабораторной работы студент должен решить следующие задачи.

1. Изучить архитектуру подсистемы замещения страниц и ее место в общей архитектуре.

2. Изучить способы описания физической памяти и виртуального адресного пространства процесса и используемые при этом структуры.

3. Определить обрабатываемые подсистемой замещения страниц события.

4. Модифицировать структуры описателей (дескрипторов) физических страниц и страниц виртуальных адресных пространств процессов в соответствие с заданным алгоритмом замещения.

5. Модифицировать обработчики целевых событий.

6. Выполнить компиляцию и сборку симулятора или ядра ОС UNIX.

7. Провести сравнительные эксперименты с использованием модулей, в которых реализованы различные алгоритмы замещения страниц.

Симулятор многозадачной системы

Выполнение лабораторных работы 1 и 2 предполагает использование одного и того же симулятора многозадачной системы, соответственно для данного варианта исполнения справедливо все написанное в разделе, посвященном лабораторной работе 1.

Управление памятью в Linux (ядро 2.4.18)

Описание физической памяти в Linux

Linux доступен для широкого диапазона архитектур, соответственно, в нем используется независимый от архитектуры способ описания памяти. Вначале мы рассмотрим структуры, использующиеся для учета банков памяти, страниц и флагов, которые относятся к управлению виртуальной памятью (ВП).

Первое принципиальное понятие, применяемое в ВП ОС Linux - Неоднородный Доступ к Памяти, Non-Uniform Memory Access (NUMA). В больших машинах память может быть организована в виде банков, доступ к которым занимает различное время в зависимости от "расстояния" до процессора. Например, могут быть банки памяти, назначенные конкретным процессорам, или банк памяти вблизи интерфейсной карты, очень подходящий для организации прямого доступа к нему.

Каждый банк памяти называют узлом (node), и это понятие представлено в Linux`е в виде структуры `struct pg_data_t` (даже в случае UMA архитектуры). Все узлы системы сохранены в списке с именем `pgdat_list`, заканчивающемся нулевым указателем. Каждый узел связан со следующим полем `pg_data_t->node_next`. Для ЭВМ с архитектурой UMA, таких как персональные компьютеры, используется только одна статическая структура `pg_data_t`, с именем `contig_page_data`.

Каждый узел делится на множество блоков, называемых зонами, которые представляют собой диапазоны адресов памяти. Зоны описываются структурами типа `struct zone_t`, и каждая из них имеет тип - ZONE_DMA, ZONE_NORMAL или ZONE_HIGHMEM. ZONE_DMA - память в районе низких физических адресов, которая может потребоваться некоторым устройствам, способным обращаться только к младшим адресам (например, некоторые устройства с интерфейсом ISA могут работать только с младшими 16 Мб). Память в пределах ZONE_NORMAL может быть непосредственно отображена ядром в верхнюю область линейного виртуального адресного пространства. Необходимо отметить, что множество операций ядра могут работать только используя зону ZONE_NORMAL, соответственно она является критической с точки зрения производительности. ZONE_HIGHMEM – остальная часть памяти. На архитектуре x86 зоны распределены следующим образом:

ZONE_DMA - первые 16MiB памяти

ZONE_NORMAL - 16MiB - 896MiB

ZONE_HIGHMEM - 896 MiB - Конец

Каждая физическая страница представлена структурой `struct page`, и все эти структуры сохранены в глобальном массиве `mem_map`, которое обычно хранится в начале зоны ZONE_NORMAL или сразу после области, зарезервированной для загружаемого образа ядра, в ЭВМ с малым количеством памяти.

Основные отношения между всеми этими структурами показаны в иллюстрации.

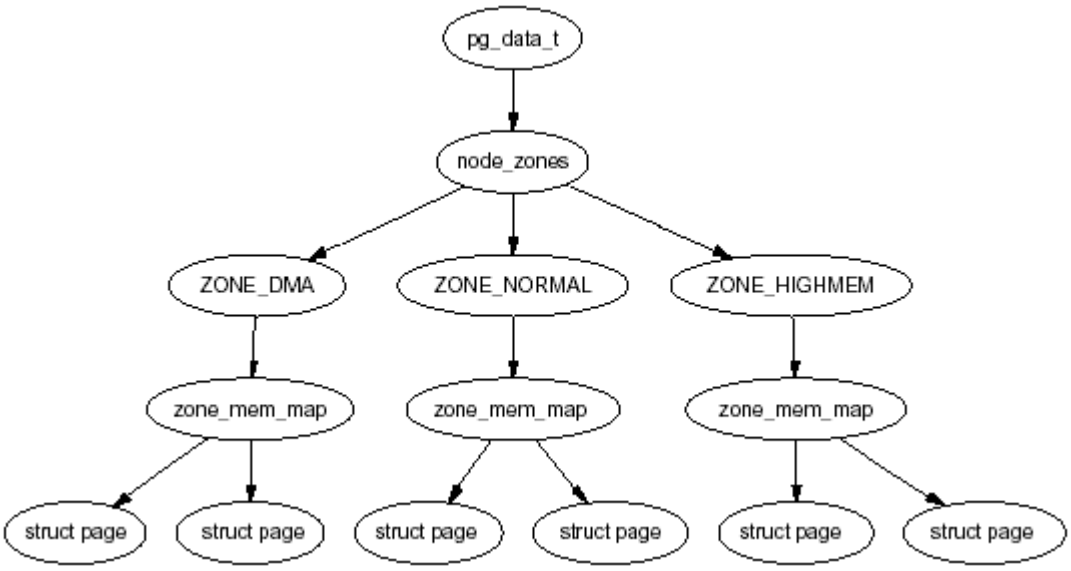


Рис. 54 Отношения между узлами, зонами и страницами

Поскольку количество памяти, непосредственно доступной ядру (ZONE_NORMAL) ограничено в размере, Linux поддерживает понятие Верхней Области Памяти (High Memory).

Узлы

Как мы упомянули ранее, каждый узел в памяти описан структурой `pg_data_t`. Назначая страницы, Linux использует локально узловую политику распределения, чтобы выделять память от узла, самого близкого к текущему процессору. Поскольку процессы как правило работают на одном и том же центральном процессоре или могут быть вообще к нему привязаны, в большинстве случаев будет использоваться память непосредственно этого процессора.

`struct pglist_data` объявлена следующим образом в <linux/mmzone.h>:

```
129 typedef struct pglist_data {
130 zone_t node_zones[MAX_NR_ZONES];
131 zonelist_t node_zonelists[GFP_ZONEMASK+1];
132 int nr_zones;
133 struct page *node_mem_map;
134 unsigned long *valid_addr_bitmap;
135 struct bootmem_data *bdata;
136 unsigned long node_start_paddr;
137 unsigned long node_start_mapnr;
138 unsigned long node_size;
139 int node_id;
140 struct pglist_data *node_next;
141 } pg_data_t;
```

Кратко опишем каждое из полей:

node_zones зоны этого узла: ZONE_HIGHMEM, ZONE_NORMAL, ZONE_DMA;

node_zonelists – предпочтительный порядок использования зон. Обычно неудавшееся выделение в ZONE_HIGHMEM приводит к попытке использования ZONE_NORMAL, в случае неудачи и здесь, используется ZONE_DMA;

nr_zones - число зон в этом узле, от 1 до 3. Не все узлы будут иметь три. Банк центрального процессора, например, может не иметь ZONE_DMA;

node_mem_map – указатель на первую страницу узла в массиве элементов типа struct page, представляющего каждую физическую страницу. Располагается внутри глобального массива mem_map;

valid_addr_bitmap - битовый массив, который описывает "дыры" в узле памяти, для которых не существует физической памяти;

bdata - для менеджера памяти, работающего во время загрузки;

node_start_paddr - стартовый физический адрес узла. Значение, хранящееся здесь – Page Frame Number (PFN) - определяется как page_phys_addr >> PAGE_SHIFT;

(PAGE_SHIFT = $\log_2$PAGE_SIZE)

node_start_mapnr - дает номер первой страницы узла в пределах глобального массива mem_map;

node_size - общее количество страниц в этой зоне;

node_id – ID (идентификатор) узла, начинаются с 0;

node_next - указатель на следующий узел в списке, завершающимся нулевым указателем.

Все узлы в системе содержаться в списке, который называется pgdat_list. Вплоть до последних 2.4 ядер (> 2.4.18), блоки программы, обходящие список, выглядели так:

```
pg_data_t * pgdat;
pgdat = pgdat_list;
do {
/* do something with pgdata_t */
...
} while ((pgdat = pgdat->node_next));
```

В более свежих ядрах введен макрос for_each_pgdat(), который тривиально определен как цикл for, чтобы улучшить удобочитаемость кода.

Зоны

Зоны описываются структурой struct zone_t. Она определена следующим образом в <linux/mmzone.h>:

```
37 typedef struct zone_struct {
41 spinlock_t lock;
42 unsigned long free_pages;
43 unsigned long pages_min, pages_low, pages_high;
```

```
44 int need_balance;
45
49 free_area_t free_area[MAX_ORDER];
50
76 wait_queue_head_t * wait_table;
77 unsigned long wait_table_size;
78 unsigned long wait_table_shift;
79
83 struct pglist_data *zone_pgdat;
84 struct page *zone_mem_map;
85 unsigned long zone_start_paddr;
86 unsigned long zone_start_mapnr;
87
91 char *name;
92 unsigned long size;
93 } zone_t;
```

Кратко опишем каждое из полей:

`lock` - Признак заблокированной зоны.

`free_pages` - общее количество свободных страниц в зоне;

`pages_min, pages_low, pages_high` - зональные отметки уровня воды (watermarks), которые описаны ниже;

`need_balance` - флаг, говорящий демону выгрузки страниц kswapd о необходимости уравновесить зону;

`free_area` – битовый массив незанятых страниц;

`wait_table` – хэш-таблица стоящих в очереди процессов, ожидающих появления свободной страницы. Это важно для функций `wait_on_page()` и `unlock_page();`

`wait_table_size` - размер хэш-таблицы, который является степенью 2;

`wait_table_shift` - определено как число битов в long минус логарифм по основанию 2 размера хэш-таблицы (предыдущий пункт);

`zone_pgdat` – указатель на родительский `pg_data_t`;

`zone_mem_map` – номер первой страницы зоны в глобальном массиве `mem_map`;

`zone_start_paddr` - аналогично `node_start_paddr`;

`zone_start_mapnr` - аналогично `node_start_mapnr`;

`name` - название зоны: DMA ("прямой доступ к памяти"), "Normal" ("Нормальный") или "HighMem" (верхняя область памяти);

`size` - размер зоны в страницах.

Отметки уровня воды (watermarks)

Когда количество доступной памяти в системе становится мало, пробуждается демон выгрузки страниц kswapd и начинает освобождать страницы. Если нагрузка велика, демон будет освобождать память синхронно (непрерывно).

Каждая зона имеет три отметки уровня воды, называемые `pages_low`, `pages_min` и `pages_high`, которые помогают отслеживать, насколько велика нагрузка на зону. Число страниц для `pages_min` рассчитывается в функции `free_area_init_core()` в ходе инициализации памяти и базируется на размере зоны в страницах. Первоначально значение рассчитывается как `ZoneSizeInPages/128`. Самое низкое значение - 20 страниц (80 КБ на x86), максимально возможное значение - 255 страниц (1MB на x86).

`pages_min` – когда достигается `pages_min` демон kswapd начинает работать в синхронном режиме;

`pages_low` – когда достигнуто число свободных страниц, равное `pages_low`, `kswapd` пробуждается, чтобы начать освобождать страницы. Значение `pages_low` по умолчанию – 2*`pages_min`;

`pages_high` – однажды пробудившись, `kswapd` будет считать зону "неуравновешенной" до тех пор, пока число свободных страниц не достигнет значения `pages_high`. Значение по умолчанию для `pages_high` равно `3*pages_min`.

Страницы

Каждая физическая страница в системе имеет связанную с ней структуру `struct page`, которая используется для хранения ее статуса (состояния). Она объявлена следующим образом в <linux/mm.h>:

```
152 typedef struct page {
153 struct list_head list;
154 struct address_space *mapping;
155 unsigned long index;
156 struct page *next_hash;
158 atomic_t count;
159 unsigned long flags;
161 struct list_head lru;
163 struct page **pprev_hash;
164 struct buffer_head * buffers;
175
176 #if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
177 void *virtual;
179 #endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */
180 } mem_map_t;
```

Кратко опишем каждое из полей:

`list` - страница может принадлежать многим спискам, и данное поле используется в качестве головы списка. Например, выделенная страница памяти будет принадлежать одному из трех циклически связанных списков, поддерживаемых структурой `address_space`. Это `clean_pages` (страницы, в которые не производилась запись), `dirty_pages` (страницы, в которые производилась запись) и `locked_pages` (заблокированные страницы). Это поле также используется при создании списка свободных страниц и в некоторых других случаях.

`mapping` - когда файлы или устройства отображаются в память, их `inode` (описатель объекта файловой системы) ассоциируется со структурой `address_space`. Данное поле содержит указатель на эту структуру или структуру типа `swapper_space`, если страница анонимна, и для нее установлено отображение на диск.

`index` - это поле допускает двойное использование в зависимости от состояния страницы. В первом случае, если страница является частью отображения файла в память, в поле хранится смещение в файле; если страница является частью отображения раздела подкачки в память (кэша раздела подкачки) - это будет смещением в пределах `address_space` для адресного пространства раздела подкачки (`swapper_space`). Во втором случае, если для какого-либо конкретного процесса освобождается блок страниц, в поле будет хранится степень освобождаемого блока (т.е. освобождается 2^{index} страниц).

`next_hash` – для страниц, являющихся частью отображения файла в память, вычисляется хэш-функция от номера inode и смещения в файле. Данное поле связывает вместе страницы, которые разделяют то же самое значение хэш-функции.

`count` – число ссылок на страницу. Если число опускается до 0, страница может быть освобождена. Любое большее значение означает, что страница используется одним или более процессами или ядром, например, при ожидании завершения операции ввода-вывода.

`flags` - флаги, которые описывают статус страницы. Все они объявлены в <linux/mm.h>

и перечислены ниже в таблице. Существует набор макросов, выполняющих операции проверки, сброса и установки битов.

`lru` – с точки зрения стратегии замещения, страницы, которые могут быть вытеснены, присутствуют либо в списке `active_list`, либо в списке `inactive_list` (объявлены в page_alloc.c). Данное поле используется для организации связей в этих списках.

`pprev_hash` - дополнение к `next_hash`;

`buffers` - если страница содержит буферы устройства блочного типа, связанные с ней, данное поле указывает на структуру типа `buffer_head`. Страница, используемая процессом, также может иметь связанную структуру `buffer_head`, если она отображена в пространство подкачки.

`virtual` - обычно только страницы из зоны ZONE_NORMAL могут быть непосредственно отображены и использованы ядром. Для получения доступа к страницам зоны ZONE_HIGHMEM используется демон kmap(), устанавливающий отображение страницы верхней памяти в доступное ядру пространство адресов. Одновременно может быть отображено только ограниченное постоянное число страниц верхней памяти. В этом случае данное поле содержит виртуальный адрес отображенной страницы (доступный процессору непосредственно и, соответственно, ядру).

Определение типа `mem_map_t` совпадает с типом `struct page`.

Таблица. Некоторые флаги состояния страницы

Имя бита	Описание
PG_active	Отмечает «горячие» страницы. Бит устанавливается, если страница в списке `active_list` (LRU), и сбрасывается, если страница удаляется из этого списка.
PG_dirty	Признак того, что страницу нужно записать на диск. Когда страница записывается на диск, бит не очищается немедленно, чтобы страница не была освобождена до завершения операции записи.
PG_launder	Бит важен только для подсистемы замещения страниц. Когда подсистема виртуальной памяти выгружает страницу на диск, она устанавливает данный бит и вызывает функцию `writepage()`. При просмотре, если обнаруживается страница с установленными битам PG_launder и PG_locked, подсистема виртуальной памяти подождет завершения операции ввода/вывода.
PG_locked	Устанавливается, когда страница должна быть заблокирована в памяти для окончания дискового ввода/вывода. Сбрасывается по окончании операции.
PG_lru	Установлен, если страница находится в `active_list` или `inactive_list`.
PG_referenced	Бит, установленный в том случае, если страница использовалась ее процессом-владельцем. Используется при работе алгоритма замещения страниц.
PG_reserved	При загрузке устанавливается для страниц, которые не выгружаются на диск. Позднее используется для указания пустых страниц или страниц, которые не существуют.

Адресное пространство процесса

Адресное пространство процесса делится на две части – пользовательскую и часть, принадлежащую ядру. Работа с этими частями существенно отличается. Например, часть, принадлежащая ядру является всегда видимой вне зависимости от того, какой процесс выполняется, и не меняется при переключениях контекста. Запросы на выделение памяти для ядра обслуживаются немедленно. При выделении памяти в пользовательской части, для дескрипторы выделенных страниц устанавливаются таким образом, чтобы ссылаться на специальную страницу, заполненную нулями. В случае если процесс попытается выполнить запись в выделенные ему страницы, происходит страничный сбой, и только в этот момент для процесса будет выделена новая страница.

Пользовательская часть адресного пространства процесса состоит из множества регионов, выровненных по границе страницы. Регионы не перекрываются и представляют множество страниц, объединенных единым назначением и правами доступа. Страницы региона могут быть еще не выделены, быть активными и располагаться в оперативной памяти или быть выгружены на диск.

Пользовательская часть адресного пространства процесса описывается структурой `mm_struct`, регионы описываются структурами `struct vm_area_struct`. Если регион имеет соотнесенный с ним блок на диске, у него будет установлено поле `vm_file`. Используя `vm_file->f_dentry->d_inode->i_mapping`, можно получить доступ к

структуре address_space, содержащую всю необходимую информацию для выполнения дисковых операций (загрузка/выгрузка страниц).

Структура mm_struct определена следующим образом (<linux/sched.h>)

```
210 struct mm_struct {
211 struct vm_area_struct * mmap;
212 rb_root_t mm_rb;
213 struct vm_area_struct * mmap_cache;
214 pgd_t * pgd;
215 atomic_t mm_users;
216 atomic_t mm_count;
217 int map_count;
218 struct rw_semaphore mmap_sem;
219 spinlock_t page_table_lock;
220
221 struct list_head mmlist;
222
226 unsigned long start_code, end_code, start_data, end_data;
227 unsigned long start_brk, brk, start_stack;
228 unsigned long arg_start, arg_end, env_start, env_end;
229 unsigned long rss, total_vm, locked_vm;
230 unsigned long def_flags;
231 unsigned long cpu_vm_mask;
232 unsigned long swap_address;
233
234 unsigned dumpable:1;
235
236 /* Architecture-specific MM context */
237 mm_context_t context;
238 };
```

mmap – начало списка регионов;

mm_rb – регионы упорядочены в связном списке и в красно-черном дереве (бинарном, для осуществления быстрого поиска); данное поле – корень бинарного дерева;

mmap_cache – в данном поле сохраняется результат последнего вызова find_vma();

pgd – Page Global Directory (таблица таблиц страниц) процесса;

mm_users – число пользователей адресного пространства процесса;

mm_count – число пользователей данной структуры mm_struct;

`map_count` – количество регионов;

`mmap_sem` – поле, позволяющее блокировать доступ к списку регионов при выполнении над ним операций чтения и записи;

`page_table_lock` – поле, используемое для блокировки доступа к большинству полей структуры;

`mmlist` – все структуры `mm_struct` связаны в список через данное поле;

`start_code,end_code` – начало и конец секции кода;

`start_data,end_data` – начало и конец секции данных;

`start_brk,brk` – начало и конец кучи;

`start_stack` – начало стека;

`arg_start,arg_end` – начало и конец секции аргументов командной строки;

`env_start,env_end` – начало и конец секции переменных окружения;

`rss` – Resident Set Size – число резидентных страниц для данного процесса;

`total_vm` – суммарный размер всех регионов процесса;

`locked_vm` – число резидентных страниц, заблокированных в памяти;

`def_flags` – может иметь значение VM_LOCKED (вся выделяемая в будущем память будет заблокирована);

`cpu_vm_mask` – битовая маска, определяющая все возможные процессоры в многопроцессорной системе (SMP);

`swap_address` – последний адрес, отправленный в область подкачки при последней выгрузке процесса из оперативной памяти целиком;

`dumpable` – Устанавливается функцией `prctl()`. Используется только при отладке.

`context` – контекст адресного пространства (специфичен для конкретной архитектуры).

Структура `vm_area_struct` определена следующим образом (<linux/mm.h>)

```
44 struct vm_area_struct {
45 struct mm_struct * vm_mm;
46 unsigned long vm_start;
47 unsigned long vm_end;
49
50 /* linked list of VM areas per task, sorted by address */
51 struct vm_area_struct *vm_next;
52
53 pgprot_t vm_page_prot;
54 unsigned long vm_flags;
55
56 rb_node_t vm_rb;
```

```
57
63 struct vm_area_struct *vm_next_share;
64 struct vm_area_struct **vm_pprev_share;
65
66 /* Function pointers to deal with this struct. */
67 struct vm_operations_struct * vm_ops;
68
69 /* Information about our backing store: */
70 unsigned long vm_pgoff;
72 struct file * vm_file;
73 unsigned long vm_raend;
74 void * vm_private_data;
75 };
```

vm_mm – структура mm_struct – владелец региона;

vm_start – адрес начала региона;

vm_end – адрес конца региона;

vm_next – все регионы объединены в список через это поле;

vm_page_prot – флаги доступа к страницам региона, установленные в таблице страниц;

vm_flags – флаги доступа к региону;

vm_rb – поле, используемое для соединения регионов в бинарное дерево;

vm_next_share – разделяемые регионы, базирующиеся на отображении файлов в память, связаны в список через данное поле;

vm_pprev_share – дополнение к vm_next_share;

vm_ops – содержит указатели на функции open(), close() и nopage();

vm_pgoff – выровненное по границе страницы смещение в файле, который отображен в память;

vm_file – файл, отображаемый в память;

vm_raend – при обработке страничного сбоя в оперативную память считывается несколько страниц (чтение с предвыборкой), данное поле определяет число дополнительно считываемых страниц;

vm_private_data – используется некоторыми драйверами устройств.

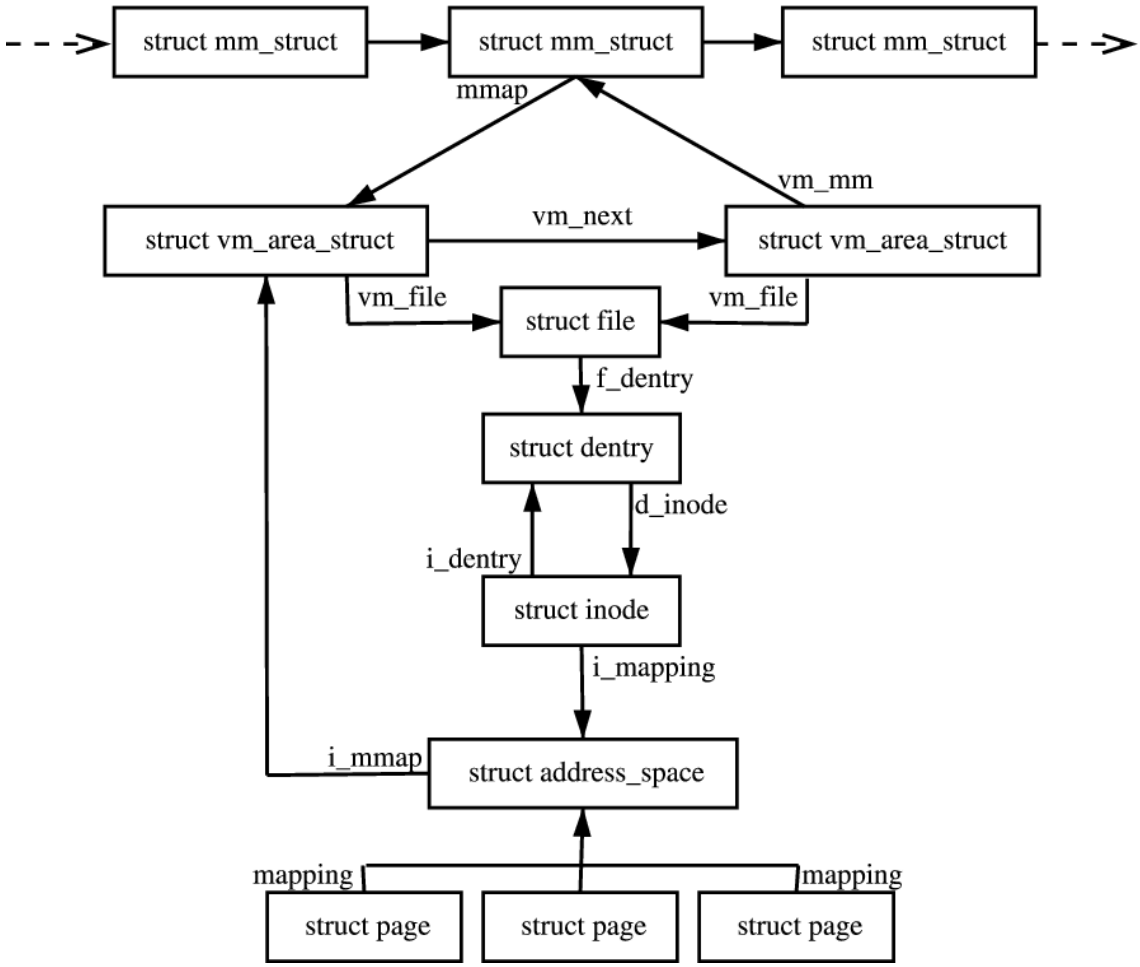


Рис. 55 Структуры данных, связанные с адресным пространством процесса

Существует ряд вызовов для работы с регионами. Упомянем два из них:

`find_vma()` – выполняет поиск региона, содержащего указанный адрес или ближайший к нему;

`find_vma_prev()` – то же, но возвращает еще и предыдущий регион.

Повторим, что в случае, если регион соотнесен с областью внутри файла на диске, используя `vm_file->f_dentry->d_inode->i_mapping`, можно получить доступ к структуре `address_space`. Данная структура определенна следующим образом (<linux/fs.h>).

```
401 struct address_space {
402 struct list_head clean_pages;
403 struct list_head dirty_pages;
404 struct list_head locked_pages;
405 unsigned long nrpages;
406 struct address_space_operations *a_ops;
407 struct inode *host;
408 struct vm_area_struct *i_mmap;
409 struct vm_area_struct *i_mmap_shared;
```

```
410 spinlock_t i_shared_lock;
411 int gfp_mask;
412 };
```

clean_pages – количество страниц, не требующих записи на диск;

dirty_pages – количество страниц, измененных процессом и требующих записи на диск;

locked_pages – число страниц, заблокированных в памяти;

nrpages – число резидентных страниц, используемое данной областью;

a_ops – структура указателей на функции файловой системы;

host – inode файла;

i_mmap – регион, частью которого является данная область;

i_mmap_shared – указатель на следующий регион, разделяющий данную область;

i_shared_lock – поле, используемое для блокировки доступа к структуре;

gfp_mask – маска, используемая при вызове __alloc_pages().

Периодически менеджер памяти должен сбрасывать страницы на диск. Структура a_ops предоставляет возможность использовать набор функций для работы со страницами области, не заботясь о том, каким образом выполняются операции (<linux/fs.h>).

```
383 struct address_space_operations {
384 int (*writepage)(struct page *);
385 int (*readpage)(struct file *, struct page *);
386 int (*sync_page)(struct page *);
387 /*
388 * ext3 requires that a successful prepare_write()
    * call be followed
389 * by a commit_write() call - they must be balanced
390 */
391 int (*prepare_write)(struct file *, struct page *,
         unsigned, unsigned);
392 int (*commit_write)(struct file *, struct page *,
         unsigned, unsigned);
393 /* Unfortunately this kludge is needed for FIBMAP.
     * Don't use it */
394 int (*bmap)(struct address_space *, long);
395 int (*flushpage) (struct page *, unsigned long);
396 int (*releasepage) (struct page *, int);
397 #define KERNEL_HAS_O_DIRECT
398 int (*direct_IO)(int, struct inode *, struct kiobuf *,
```

```
                        unsigned long, int);
399 };
```

`writepage` – записать страницу на диск;

`readpage` – прочитать страницу с диска;

`sync_page` – синхронизировать измененную страницу с диском;

`prepate_write` – используется перед тем, как данные копируются из пользовательской части процесса в страницы, которые будут записаны на диск;

`commit_write` – записывает данные, скопированные после вызова prepate_write;

`flushpage` – блокирует страницу до завершения операций ввода/вывода с ней;

`releasepage` – пытается записать все буферы, связанные со страницей, перед ее освобождением.

Страничные сбои

Linux, как и многие другие операционные системы, для страниц, не расположенных в оперативной памяти, использует стратегию выборки «по запросу». Это означает, что страницы считываются в оперативную память в том случае, если возник страничный сбой. В Linux, при считывании страницы из области подкачки, вместе с ней считываются $2^{page\_cluster}$ страниц и помещаются в кэш подкачки.

Различают два вида страничных сбоев – major, когда данные необходимо считывать с жесткого диска, и minor – когда считывать с диска не надо. Статистика о числе страничных сбоев ведется для каждого процесса в полях `task_struct->maj_flt` и `task_struct->min_flt`.

Обработку страничного сбоя производит функция `do_page_fault()`. В данную функцию передается адрес страничного сбоя, просто не найдена страница или это ошибка защиты страницы, сбой ли это чтения или записи, сбой ли это в пользовательской части адресного пространства или в части ядра. Граф вызовов функции изображен на рисунке.

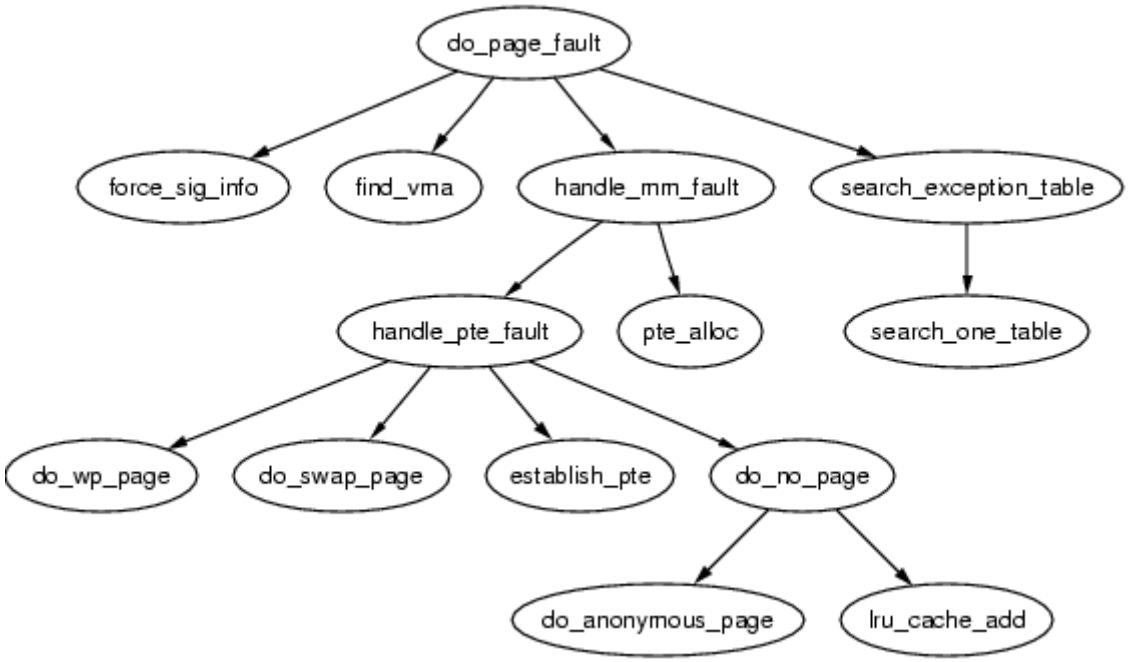


Рис. 56 Граф вызовов функции do_page_fault

`handle_mm_fault()` – платформенно-независимая функция считывания страницы с жесткого диска, обработки страниц COW(Copy-On-Write) и т.д. Она возвращает 1 в случае minor-сбоя, 2 в случае major-сбоя, 0 инициирует посылку сигнала SIGBUS, любое другое значение означает, что память выделить невозможно (вызывается обработчик “out-of-memory”).

Функция `handle_mm_fault()` определяет регион, в котором произошел сбой, и соответствующую сбою таблицу страниц (если она не существовала, то создается). После этого вызывается функция `handle_pte_fault()`.

На основании содержимого дескриптора страницы, производится выбор способа обработки сбоя.

Во-первых, определяется, отсутствует ли страница в оперативной памяти (`pte_none() == false && pte_present() == true`) или для данной страницы ВАП еще не была назначена физическая страница (`pte_none() == true`). В первом случае выполняется «подкачка по запросу» посредством вызова функции `do_swap_page()`, во втором – «выделение памяти по запросу» посредством вызова `do_no_page()`.

Во-вторых, проверяются права доступа к странице. Если в таблице страниц указано, что страница защищена от записи, это означает, что данная страница используется несколькими процессами в режиме Copy-On-Write, и необходимо сделать «личную» копию данной страницы для текущего процесса. Это делает функция `do_wp_page()`.

Выделение памяти по запросу

Когда процесс обращается к странице в первый раз, для данной страницы виртуального адресного пространстве еще не назначена физическая страница. Выделение выполняет функция `do_no_page()`. Исключение составляет случай, когда регион – владелец страницы, предоставляет функцию для выполнения данного действия (функция `nopage()` в поле `vm_area_struct->vm_ops`).

Если поле vm_area_struct->vm_ops не предоставляет функцию nopage(), вызывается функция do_anonymous_page(). Если обслуживается запрос на чтение, в дескриптор страницы помещается ссылка на специальную страницу, заполненную нулями, и для данной страницы запрещается запись (для всех процессов используется одна и та же такая страница). Когда процесс попытается выполнить запись на эту страницу, возникнет еще одн страничный сбой, который можно будет обработать.

Если произошла попытка первой записи в страницу, вызываются функции alloc_page() и clear_user_highpage(), выделяющие страницу физической памяти и обнуляющие ее. Если выделение прошло успешно, Resident State Size процесса (mm_struct->rss) увеличивается на 1, и вызывается функция flash_page_to_ram() для гарантированной синхронизации процессорного кэша. После этого страница вставляется в список lru, чтобы впоследствии ее можно было выгрузить на диск, и модифицируется таблица страниц.

Если страница была отображена на пространство на жестком диске (во внешней памяти), поле vm_area_struct->vm_ops предоставляет функцию nopage(), которая реализована в драйвере устройства внешней памяти. Данная функция выполняет выделение памяти для страницы и чтение ее из внешней памяти. После возврата из функции выполняется ряд проверок, модифицируется таблица страниц и синхронизируется процессорный кэш.

Подкачка по запросу

Если страница была выгружена на жесткий диск, ее чтение выполняет функция do_swap_page(). Информация о местонахождении страницы на диске, достаточная для ее чтения, содержится в дескрипторе страницы. Однако страница может все еще находиться в оперативной памяти.

Дело в том, что страницы, разделяемые несколькими процессами, не могут быть выгружены на диск немедленно. По содержимому структуры struct page невозможно определить все таблицы страниц, в которых существует ссылка на нее, а поиск посредством перебора таблиц страниц всех процессов занимает слишком много времени. Поэтому, когда страница выгружается в область подкачки, сначала она попадает в кэш подкачки.

Соответственно, если произошел страничный сбой, есть шанс, что требуемая страница находится в кэше подкачки. В этом случае счетчик ссылок на страницу увеличивается на 1 и исправляется таблица страниц процесса.

Если страница присутствует только на диске, вызывается функция swapin_readahead(), считывающая требуемую страницу и несколько следующих за ней ($2^{page\_cluster}$).

Демон выгрузки страниц (kswapd)

Работающая система может использовать все имеющиеся физические страницы, распределив их работающим процессам, выделив для хранения буферов обмена с устройствами ввода/вывода и т.д. Соответственно, потребуется выбрать ряд страниц, содержимое которых будет вытеснено на жесткий диск, а сами страницы освобождены для использования в других целях.

Методы, используемые в Linux для определения вытесняемых страниц, базируются на нескольких различных идеях и их параметры подобраны на основании анализа замеров производительности и отзывов пользователей.

За исключением страниц, управляемых менеджером памяти ядра (kernel allocator или slab allocator), все используемые страницы содержатся в кэше страниц и связаны в список через

поля `page->lru`. Таким образом, они могут быть всегда просмотрены для поиска замещаемых страниц.

Страницы, выделенные процессам, не так просто выгрузить, поскольку для поиска по структуре `struct page` соответствующей ей таблицы страниц требуется просмотреть все таблицы страниц, что слишком трудоемко. Вместо этого, когда кэш страниц содержит достаточно большое количество страниц, просматривается таблица процессов и их таблиц страниц, определяются предназначенные к вытеснению страницы и выгружаются функцией `swap_out()` до тех пор, пока не будет освобождено достаточное количество страниц.

Если страница используется несколькими процессами, для нее уменьшается счетчик использования, но она не выгружается до тех пор, пока он не станет равен 0. Считается, что до этого она находится в «кэше подкачки».

При загрузке системы начинает исполняться демон ядра `kswapd`, выполняющий функцию `kswapd()` (mm/vmscan.c). Обычно он находится в состоянии ожидания (сна). Данный демон отвечает за выгрузку страниц при уменьшении размера свободной памяти. В настоящий момент он пробуждается подсистемой выделения физической памяти (physical page allocator), когда число свободных страниц в зоне становится меньше `pages_low`.

Демон swapd поддерживает состояние кэша страниц, усекает кэш менеджера памяти ядра и при необходимости выгружает процессы. Он освобождает страницы до тех пор, пока количество свободных страниц в зоне не достигнет `pages_high`.

Когда kswapd выходит из состояния ожидания, он выполняет следующие действия:

- вызывает функцию `kswapd_can_sleep()`, просматривающую все зоны и проверяющую флаг `need_balance`; если хоть у одной зоны данный флаг установлен, необходима дальнейшая обработка, если нет – kswapd опять засыпает;

- удаляет себя из очереди ожидания kswapd_wait;

- вызывает функцию `kswapd_balance()`, которая просматривает все зоны, и в зонах, требующих балансировки, производится освобождение страниц функцией `try_to_free_pages_zone()` до тех пор, пока не будет в наличии `pages_high` свободных страниц;

- формируется задание для tq_disk, чтобы страницы были записаны на диск;

- kswapd добавляется обратно в очередь kswapd_wait.

Кэш страниц

Кэш страниц – это список страниц, каждой из которых соответствует страница в файле, устройстве блочного типа или области подкачки. В нем содержатся четыре типа страниц:

- страницы, относящиеся к файлу, отображаемому в память, из которых было выполнено чтение;

- буферные страницы – блоки, считанные с устройства блочного типа или файловой системы;

- анонимные страницы (страницы, выделяемые по запросу пользовательским процессам) при поступлении в кэш не имеют соотнесенных с ними страниц во внешней памяти, однако если возникает необходимость выгрузить их, ядро выделяет для них пространство в области подкачки;

- страницы, относящиеся к разделяемой памяти, обрабатываются почти так же как и анонимные страницы; разница состоит только в том, что для разделяемых страниц выделяется пространство на внешнем носителе (и они добавляются в кэш подкачки) сразу после первой операции записи.

«Кэш подкачки» - это абстрактное понятие, включающее в себя все страницы физической памяти, для хранения которых зарезервировано место на внешнем носителе. Не существует специальных структур данных, связывающих страницы, входящие в «кэш подкачки».

Страницы присутствуют в кэше по двум причинам. Во-первых, для уменьшения числа операций чтения с внешних носителей. Для страниц, прочитанных с диска, организована хеш-таблица `page_hash_table`. При формировании значения ключа используется структура `struct address_space` и смещение в файле. Эта таблица всегда просматривается перед обращением к диску. Во-вторых, кэш страниц организован в виде очереди, которая используется в алгоритме выбора замещаемой страницы.

Кэш страниц состоит из двух списков (определены в mm/pagealloc.c), называемых `active_list` и `inactive_list`. Первый представляет «горячие» (используемые) страницы, второй – холодные (редко используемые или не используемые). Доступ к спискам блокируется полем `pagemap_lru_lock`. Для работы с кэшем страниц предоставляется несколько функций.

Добавление страниц в кэш страниц

Страницы, считываемые из файла или устройства блочного типа, добавляются в кэш страниц посредством вызова из функции `generic_file_read()` функции `__add_to_page_cache()`.

Анонимные страницы (страницы, выделяемые по запросу пользовательских процессов) в момент, когда для них выделяется место во внешней памяти (при первой попытке выгрузить страницу). Единственное отличие между анонимными страницами и страницами, отображенными в файл состоит в том, что анонимные страницы используют структуру `swapper_space`, а отображенные в файл – `address_space`.

Разделяемые страницы добавляются в одном из двух случаев. Первый – когда страница считывается из области подкачки или к ней происходит первое обращение. Второй – когда код вытеснения страниц во внешнюю память вызывает `shmem_unuse()`. Это происходит в ситуации, когда деактивируется область подкачки и обнаруживается, что страница, отображенная в область подкачки, не используется ни одним процессом. В обоих случаях используется функция `add_to_page_cache()`.

Заполнения списка «холодных» страниц

При усечении (очистке) кэшей страницы перемещаются из списка `active_list` в список `inactive_list` функцией `refill_inactive()`. В качестве входного параметра она получает количество страниц, которые нужно выгрузить на диск `nr_pages`, и вычисляет число перемещаемых страниц по формуле:

```
pages = nr_pages * nr_active_pages / (2*nr_inactive_pages+1)
```

что позволяет поддерживать размер списка `active_list` примерно в 2/3 размера `inactive_list`.

Страницы берутся из конца списка `active_list`, если флаг PG_referenced установлен, он сбрасывается, и страница помещается в начало списка `active_list`. Если флаг

PG_referenced сброшен, то страница перемещается в начало списка `inactive_list`, и ее флаг PG_referenced устанавливается.

Удаление страниц из кэша страниц

Решение о вытеснении страниц и определение способа вытеснения выполняется в функции `shrink_cache()`. Объем работы функции определяется двумя параметрами – `nr_pages` и `priority` (начальные значения SWAP_CLUSTER_MAX и DEF_PIORITY соответственно).

Внутренние параметры `max_scan` и `max_mapped` вычисляются с использованием priority. Каждый раз, когда функция `shrink_caches()` вызывается из-за недостаточного количества освобожденных страниц, приоритет уменьшается до тех пор, пока не будет достигнут наивысший приоритет, равный 1.

`max_scan` – это максимальное число страниц, просматриваемое функцией `shrink_cache()`. Оно вычисляется как

```
max_scan = nr_inactive_pages / priority
```

При минимальном приоритете (6) просматривается 1/6 часть списка, при максимальном – весь список.

Второй параметр – `max_mapped` – определяет, максимальное число страниц, которое процессы могут иметь в кэше страниц. Если это число превышено, процессы начинают вытесняться во внешнюю память целиком. Это вычисляется как минимум из двух величин:

```
max_mapped = min( max_scan / 10, nr_pages*2(10-priority) )
```

При минимальном приоритете (6) max_mapped = min(nr_inactive_pages / 60, nr_pages*2^4), при максимальном – весь список max_mapped = min(nr_inactive_pages / 10, nr_pages*2^9)

Дальнейший код функции представляет собой цикл, в котором просматриваются максимум `max_scan` страниц из списка `inactive_list` (или пока список не закончился). Для разных типов страниц принимаются различные решения.

- Страница принадлежит адресному пространству процесса. Значение `max_mapped` уменьшается на 1. Если оно достигло 0, таблицы страниц процессов начинают последовательно просматриваться и страницы процессов выгружаются функцией `swap_out()`.

- Страница заблокирована и у нее установлен флаг PG_launder. Создается ссылка на страницу (чтобы она не пропала) и вызывается функция `wait_on_page()`, ожидающая завершения ввода/вывода. Затем число ссылок на страницу уменьшается на 1, и если оно стало равным 0, страница освобождается.

- Страница изменена (установлен флаг PG_dirty), не используется ни одним процессом, не имеет буферов и относится к устройству или файлу, отображаемому в память. Бит PG_dirty сбрасывается, бит PG_launder устанавливается, создается ссылка на страницу и вызывается функция `write_page()`, предоставленная отображением (`address_space->a_ops`), которая освобождает страницу.

- Страница содержит буфер, ассоциированный с данными на диске. Создается ссылка на страницу и предпринимается попытка освободить ее функцией `try_to_release_page()`. В случае успеха, и данная страница анонимна, ее можно освободить. В противном случае, уменьшается число ссылок.

- Анонимная страница, принадлежащая процессу и не содержащая буферов. Со страницы снимается блокировка, значение `max_mapped` уменьшается на 1. Если оно достигло 0, таблицы страниц процессов начинают последовательно просматриваться и страницы процессов выгружаются функцией `swap_out()`.

- Страница имеет счетчик ссылок, равный 0. Если страница отображена на файл, она удаляется из списка страниц, связанных с `inode`. Затем страница удаляется из кэша страниц и освобождается.

Выгрузка страниц процесса

Если в кэше страниц обнаруживается число страниц процессов, превышающее max_mapped, вызывается функция swap_out(), выполняющая выгрузки страниц процессов. Начиная со структуры mm_struct, на которую указывает переменная swap_mm и mm->swap_address, просматриваются таблицы страниц до тех пор, пока не будет освобождено nr_pages страниц.

Все страницы процесса проверяются, вне зависимости от того, находятся ли они в списке и когда к ним осуществлялся доступ; однако страницы, принадлежащие списку active_list, и страницы, к которым недавно осуществлялся доступ, пропускаются.

Поскольку решение о выгрузке страниц процессов уже принято, делается попытка выгрузить на диск сразу по крайней мере SWAP_CLUSTER страниц, однако при этом просмотр всех структур mm_struct выполняется только один раз (чтобы избежать зацикливания если нет блоков такого размера, которые можно выгрузить). Вытеснение страниц блоками увеличивает скорость вытеснения, а также вероятность того, что близкие страницы ВАП процесса будут располагаться близко на внешнем носителе.

Когда процесс выбирается для вытеснения на внешний носитель, число ссылок на его структуру mm_struct увеличивается на 1, и вызывается swap_out_mm(), которая перебирает все регионы процесса и для каждого вызывает swap_out_vma(). Далее, swap_out_pgd() и swap_out_pmd просматривают таблицы страниц для региона и, наконец, вызывается try_to_swap_out() для вытеснения конкретной страницы.

try_to_swap_out() проверяет, не является ли страница элементом списка active_list, не использовалась ли она недавно и находится ли она в зоне, для которой выполняется балансировка. Если все условия выполнены, страница вытесняется и ссылка на нее удаляется из таблицы страниц процесса. Если данная страница изменялась, инициируется сохранене ее содержимого на диск.

Отметим, что страницы, содержащие дисковые буферы, не обрабатываются и не выгружаются на диск.

Стратегия замещения страниц

В ходе объяснений мы упоминали, что используется стратегия замещения, основанная на алгоритме LRU. Это не совсем так, поскольку страницы в списках не хранятся в порядке LRU. Скорее, active_list предназначается для того, чтобы хранить рабочее множество страниц для всех процессов. Отметим также, что поскольку все страницы содержатся в двух массивах и при определении вытесняемой страницы выбирается страница любого процесса, а не обязательно того, что вызвал страничный сбой, мы имеем дело с глобальной стратегией замещения.

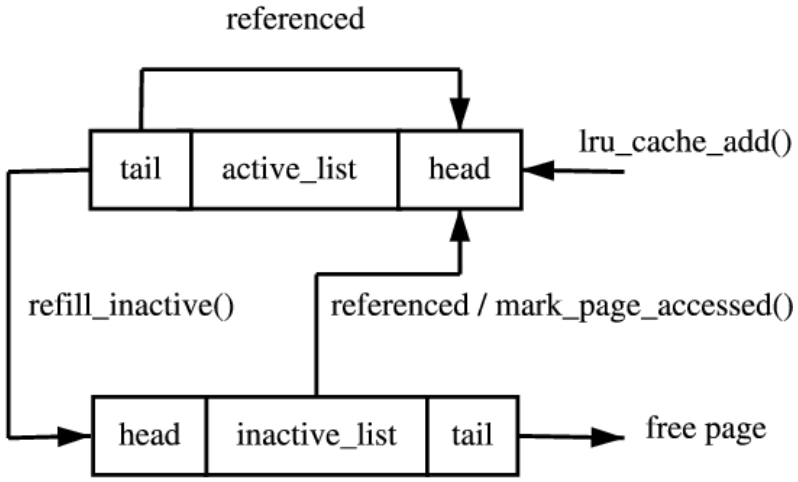


Рис. 57 Списки кэша страниц

Мы имеем две очереди. Каждая выделенная страница помещается в конец первой очереди (inactive_list). Если к ней произошло обращение, она перемещается во вторую очередь (active_list). При этом используются функции lru_cache_add() и mark_page_accessed(). Перемещение из второй очереди назад в первую осуществляет функция refill_inactive().

Описанный алгоритм имеет большое сходство со стратегией замещения «часы».

Изменение стратегии замещения

При изменении стратегии замещения нужно четко локализовать места изменений. Например, изменять структуры, описывающие физическую память, имеет смысл лишь в том случае, если Вы собираете статистику использования страниц (например, при реализации алгоритма lru).

При реализации стратегии замещения обратите внимание на следующие моменты:

- текущее использование функций, изменяющих значение флагов PG_references и PG_dirty. (SetPageReferenced(),mark_page_accessed(), set_page_dirty() и др.);

- изучите структуры данных, используемые для реализации стратегии замещения;

- рассмотрите функции, оперирующие перемещением страниц в списках lru (lru_cache_add(), refill_inactive(), shrink_cache());

- изучите функции, ответственные непосредственно за вытеснение страниц (do_swap_page(), kswapd(), shrink_caches(), try_to_free_pages(), swap_out*());

- разберите процесс инициализации подсистемы (page_cache_init(), kswapd_init()).

Литература по лабораторной работе 2

1. Э. Таненбаум. Современные операционные системы. 2-е издание. СПб: Питер, 2002.

2. А.Я. Архангельский. Интегрированная среда разработки C++Builder 5. М: Бином, 2000.

3. Mel Gorman. Understanding The Linux Virtual Memory Manager. 2003. (Электронный источник - http://www.csn.ul.ie/~mel/projects/vm/guide/html/understand/)

4. Robert Love. Linux Kernel Development. SAMS, 2003.

5. David Rusling. The Linux Kernel (Электронный источник - www.linuxdoc.org/LDP/tlk/)

6. Cross-Referencing Linux (Электронный источник - lxr.linux.no)

7. Скотт Максвелл. Ядро Linux в комментариях. К. Издательство «ДиаСофт», 2000.

Лабораторные работы 3-4. Синхронизация процессов/потоков. Передача данных между процессами/потоками

Цель работы - практическое освоение механизмов синхронизации процессов и их посредством механизмов, предоставляемых ОС.

Лабораторная работа 3 предполагает решение одной из классических задач синхронизации в конкретной операционной среде.

1. Реализация задачи «поставщик-потребитель». Требуется реализовать приложение-поставщик и приложение-потребитель или многопоточное приложение с потоком «поставщик» и потоком «потребитель».

2. Реализация задачи «читатели-писатели» (аналогично).

3. Реализация задачи «обедающие философы». Требуется реализовать многопоточное приложение с параметром «число философов».

4. Реализация задачи «спящий парикмахер» (аналогично).

Лабораторная работа 4 подразумевает создание приложений (или одного многопоточного приложения), которые не только синхронизуются, но и обмениваются данными.

Работа имеет следующие варианты заданий:

1. Имеется поток (сервер), обрабатывающий запросы другого потока (клиента). Сервер в один момент времени может обрабатывать только один запрос. Клиент не должен посылать следующий запрос, не дождавшись ответа сервера. В качестве запроса посылается номер стоки файла, содержимое которого сервер возвращает в качестве ответа. Предполагаются следующие варианты задачи: синхронизация с помощью семафоров, передача запроса - ответа через общую память или через программные каналы.

2. Процесс писатель записывает содержимое некоторого файла. Процессы - читатели считывают данные, записанные процессом - писателем. Необходимо обеспечить взаимное исключение доступа к данным писателя и любого из читателей. Предполагаются следующие варианты задачи: синхронизация с помощью семафоров, обмен данными через общую память, программные каналы или очереди.

3. Обедающие философы (вариация). Создается пять процессов (по одному на философа). Процессы разделяют пять переменных (вилок). Каждый процесс находится только в двух состояниях - либо он "размышляет", либо "ест спагетти". Чтобы начать "есть", процесс должен взять "две вилки" (захватить две переменные). Закончив "еду", процесс освобождает захваченные переменные и начинает "размышлять" до тех пор, пока снова "проголодается". Предполагается синхронизация с помощью семафоров.

4. Исходный процесс порождает два процесса P1 и P2, каждый из которых готовит данные для обработки их основным процессом. Подготавливаемые данные процесс P2 помещает в канал K1, затем они оттуда читаются процессом P1, переписываются в канал K2, дополняются своими данными. Обработка данных основным процессом заключается в чтении информации из программного канала K2 и печати ее.

5. Исходный процесс создает два программных канала K1 и K2 и порождает новый процесс P1, а тот, в свою очередь, еще один процесс P2, каждый из которых готовит данные для

обработки их основным процессом. Подготавливаемые данные процесс P1 помещает в канал К1, а процесс Р2 в канал К2, откуда они процессом Р1 копируются в канал К1 и дополняются новой порцией данных. Обработка данных основным процессом заключается в чтении информации из программного канала К1 и печати ее.

6. Исходный процесс создает программный канал К1 и порождает новый процесса Р1, а тот, в свою очередь, порождает еще один процесс Р2. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Файл, читаемый процессом Р2, должен быть достаточно велик с тем, чтобы его чтение не завершилось ранее, чем закончится установленная задержка в n секунд. После срабатывания будильника процесс Р1 посылает сигнал процессу Р2, прерывая чтение файла. Обработка данных основным процессом заключается в чтении информации из программного канала и печати ее.

В ходе выполнения лабораторной работы слушатель должен решить следующие задачи.

1. Ознакомиться с заданием к лабораторной работе.

2. Выбрать набор примитивов синхронизации и реализующих их системных вызовов, обеспечивающих решение задачи.

3. Для указанного варианта составить программу на языке Си, реализующую требуемые действия.

4. Отладить и протестировать составленную программу, используя инструментарий ОС UNIX(Windows).

5. Защитить лабораторную работу, ответив на контрольные вопросы и написав отчет.

Механизмы межпроцессного взаимодействия ОС UNIX

Предполагается, что слушателям знакомы программные средства, связанные с созданием и управлением процессами в рамках ОС UNIX. Данная лабораторная работа предполагает комплексное их использование при решении задачи синхронизации процессов и их взаимодействия посредством различных механизмов, предоставляемым ОС.

Кратко перечислим состав системных вызовов, требуемых для выполнения лабораторных работ:

1. Создание, завершение процесса, получение информации о процессе, - `fork(), exit(), getpid(), getppid();`

2. Синхронизация процессов - `signal(), kill(), sleep(), alarm(), wait(), pause();`

3. Создание информационного канала и работа с ним - `pipe(), read(), write().`

Механизм IPC (Inter-Process Communication Facilities) включает:

- средства, обеспечивающие возможность синхронизации процессов при доступе к совместно используемым ресурсам (семафоры - semaphores);

- средства, обеспечивающие возможность посылки процессом сообщений другому произвольному процессу (очереди сообщений - message queries);

- средства, обеспечивающие возможность наличия общей для процессов памяти (сегменты разделяемой памяти - shared memory segments).

Наиболее общим понятием IPC является ключ, хранимый в общесистемной таблице и обозначающий объект межпроцессного взаимодействия, доступный нескольким процессам. Обозначаемый ключом объект может быть очередью сообщений, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип key_t, состав которого зависит от реализации и определяется в файле <sys/types.h>. Ключ используется для создания объекта межпроцессного взаимодействия или получения доступа к существующему объекту. Обе операции выполняются посредством операции get. Результатом операции get является его целочисленный идентификатор, который может использоваться в других функциях межпроцессного взаимодействия.

Семафоры.

Для работы с семафорами поддерживаются три системных вызова:

- `semget()` для создания и получения доступа к набору семафоров;

- `semop()` для манипулирования значениями семафоров (это тот системный вызов, который позволяет процессам синхронизоваться на основе использования семафоров

- `semctl()` для выполнения разнообразных управляющих операций над набором семафоров

Прототипы перечисленных системных вызовов описаны в файлах

```
#include <sys/ipc.h>
#include <sys/sem.h>
```

Системный вызов semget() имеет следующий синтаксис:

```
semid = int semget(key_t key, int count, int flag);
```

параметрами которого является ключ (key) набора семафоров и дополнительные флаги (flags), определенные в <sys/ipc.h>, число семафоров в наборе семафоров (count), обладающих одним и тем же ключом. Системный вызов возвращает идентификатор набора семафоров semid. После вызова semget() индивидуальный семафор идентифицируется идентификатором набора семафоров и номером семафора в этом наборе. Флаги системного вызова semget() приведены ниже в таблице.

Таблица. Флаги системного вызова `semget()`

IPC_CREAT	Semget создает новый семафор для данного ключа. Если флаг IPC_CREAT не задан, а набор семафоров с указанным ключом уже существует, то обращающийся процесс получит идентификатор существующего набора семафоров.
IPC_EXLC	Флаг IPC_EXLC вместе с флагом IPC_CREAT предназначен для создания (и только для создания) набора семафоров. Если набор семафоров уже существует, Semget возвратит -1, а системная переменная errno будет содержать значение EEXIST.

Младшие 9 бит флага задают права доступа к набору семафоров.

Системный вызов `semctl()` имеет формат

```
int semctl (int semid, int sem_num, int command, union semun arg),
```

где semid - это идентификатор набора семафоров, sem_numb - номер семафора в группе, command - код операции, а arg - указатель на структуру, содержимое которой интерпретируется по-разному, в зависимости от операции.

Структура msg имеет вид:

```
union semun { int val;
struct semid_ds *buf;
unsigned short *array; };
```

С помощью semctl() можно

- уничтожить набор семафоров или индивидуальный семафор в указанной группе (IPC_RMID);

- вернуть значение отдельного семафора (GETVAL) или всех семафоров (GETALL);

- установить значение отдельного семафора (SETVAL) или всех семафоров (SETALL);

- вернуть число семафоров в наборе семафоров (GETPID).

Основным системным вызовом для манипулирования семафором является

```
int semop (int semid, struct sembuf *op_array, count),
```

где semid - это ранее полученный дескриптор группы семафоров, op_array - массив структур sembuf, определенных в файле <sys/sem.h.h> и содержащих описания операций над семафорами группы, а count - размер этого массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора. Каждый элемент массива op_array имеет следующую структуру (структура sembuf):

- номер семафора в указанном наборе семафоров

- операция над семафором;

- флаги.

Если указанные в массиве op_array номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов последовательно меняет значение семафора (если это возможно) в соответствии со значением поля "операция". Возможны три случая:

1. Отрицательное значение sem_op.

Если значение поля операции sem_op отрицательно, и его абсолютное значение меньше или равно значению семафора semval, то ядро прибавляет это отрицательное значение к значению семафора.

Если в результате значение семафора стало нулевым, то ядро активизирует все процессы, ожидающие нулевого значения этого семафора.

Если же значение поля операции sem_op по абсолютной величине больше семафора semval, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора и усыпляет текущий процесс до наступления этого события.

2. Положительное значение sem_op.

Если значение поля операции sem_op положительно, то оно прибавляется к значению семафора semval, а все процессы, ожидающие увеличения значения семафора, активизируются (пробуждаются в терминологии UNIX).

3. Нулевое значение sem_op.

Если значение поля операции sem_op равно нулю, то если значение семафора semval также равно нулю, выбирается следующий элемент массива op_array.

Если же значение семафора semval отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания

При использовании флага IPC_NOWAIT ядро ОС UNIX не блокирует текущий процесс, а лишь сообщает в ответных параметрах о возникновении ситуации, приведшей бы к блокированию процесса при отсутствии флага IPC_NOWAIT.

Очереди сообщений.

Для обеспечения возможности обмена сообщениями между процессами механизм очередей поддерживается следующими системными вызовами:

`msgget()` для образования новой очереди сообщений или получения дескриптора существующей очереди;

`msgsnd()` для постановки сообщения в указанную очередь сообщений;

`msgrcv()` для выборки сообщения из очереди сообщений;

`msgctl()` для выполнения ряда управляющих действий

Прототипы перечисленных системных вызовов описаны в файлах

```
#include <sys/ipc.h>
#include <sys/msg.h>
```

По системному вызову msgget() в ответ на ключ (key) и набор флагов (полностью аналогичны флагам в системном вызове semget()) ядро либо создает новую очередь сообщений и возвращает пользователю идентификатор созданной очереди, либо находит элемент таблицы очередей сообщений, содержащий указанный ключ, и возвращает соответствующий идентификатор очереди:

```
int msgqid = msgget(key_t key, int flag).
```

Для помещения сообщения в очередь служит системный вызов msgsnd():

```
int msgsnd (int msgqid, void *msg, size_t size, int flag),
```

где msg - это указатель на структуру длиной size, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив-сообщение.

Структура msg имеет вид:

```
struct msg {
long mtype; /* тип сообщения */
char mtext[SOMEVALUE]; /* текст сообщения (SOMEVALUE - любое */};
```

Параметр flag определяет действия ядра при выходе за пределы допустимых размеров внутренней буферной памяти (флаг IPC_NOWAIT со значением, рассмотренным выше).

Условиями успешной постановки сообщения в очередь являются:

- наличие прав процесса по записи в данную очередь сообщений;

- не превышение длиной сообщения заданного системой верхнего предела;

- положительное значение указанного в сообщении типа сообщения.

Если же оказывается, что новое сообщение невозможно буферизовать в ядре по причине превышения верхнего предела суммарной длины сообщений, находящихся в данной очереди

сообщений (флаг IPC_NOWAIT при этом отсутствует), то обратившийся процесс откладывается (усыпляется) до тех пор, пока очередь сообщений не разгрузится процессами, ожидающими получения сообщений.

Для приема сообщения используется системный вызов msgrcv():

```
int msgrcv (int msgqid, void *msg, size_t size, long msg_type, int flag);
```

Системный вызов msgctl()

```
int msgctl (int msgqid, int command, struct msqid_ds *msg_stat)
```

используется

- для опроса состояния описателя очереди сообщений (command = IPC_STAT) и помещения его в структуру msg_stat (детали опускаем);

- изменения его состояния (command = IPC_SET), например, изменения прав доступа к очереди;

- для уничтожения указанной очереди сообщений (command = IPC_RMID).

Работа с разделяемой памятью.

Для работы с разделяемой памятью используются системные вызовы:

`shmget()` создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;

`shmat()` подключает сегмент с указанным описателем к виртуальной памяти обращающегося процесса;

`shmdt()` отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;

`shmctl()` служит для управления разнообразными параметрами, связанными с существующим сегментом.

Прототипы перечисленных системных вызовов описаны в файлах

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

После того, как сегмент разделяемой памяти подключен к виртуальной памяти процесса, этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи.

Системный вызов

```
int shmid = shmget (key_t key, size_t size, int flag)
```

на основании параметра size определяет желаемый размер сегмента в байтах. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является идентификатор существующего сегмента. В противном случае создается новый сегмент с размером не меньше установленного в системе минимального размера сегмента разделяемой памяти и не больше установленного максимального размера. Создание сегмента не означает немедленного выделения под него основной памяти и это действие откладывается до выполнения первого системного вызова подключения сегмента к

виртуальной памяти некоторого процесса. Флаги IPC_CREAT и IPC_EXCL аналогичны рассмотренным выше.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову shmat():

```
void *virtaddr = shmat(int shmid, void *daddr, int flags).
```

Параметр shmid - это ранее полученный идентификатор сегмента, а daddr - желаемый процессом виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является фактический виртуальный адрес начала сегмента. Если значением daddr является NULL, ядро выбирает наиболее удобный виртуальный адрес начала сегмента. Флаги системного вызова shmat() приведены ниже в таблице.

Таблица Флаги системного вызова shmat()

SHM_RDONLY	ядро подключает участок памяти только для чтения;
SHM_RND	определяет, если возможно, способ обработки ненулевого значения daddr.

Для отключения сегмента от виртуальной памяти используется системный вызов shmdt():

```
int shmdt(*daddr);
```

где daddr - это виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный от системного вызова shmat().

Системный вызов shmctl:

```
int shmctl (int shmid, int command, struct shmid_ds *shm_stat);
```

по синтаксису и назначению системный вызов полностью аналогичен msgctl().

Механизмы межпроцессного взаимодействия ОС Windows

Ниже кратко описаны некоторые системные вызовы, которые могут потребоваться при выполнении лабораторной работы в ОС Windows.

Wait-функции

Wait-функции позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта ядра. Из всего семейства этих функций чаще всего используется WaitForSingleObject:

```
DWORD WaitForSingleObject( HANDLE hObject, DWORD dwMilliseconds);
```

Когда поток вызывает эту функцию, первый параметр, hObject, идентифицирует объект ядра, поддерживающий состояния «свободен-занят» Второй параметр, dwMilliseconds, указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта.

Следующий вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый описателем hProcess.

```
WaitForSingleObject(hProcess, INFINITE);
```

В данном случае константа INFINITE, передаваемая во втором параметре, подсказывает системе, что вызывающий поток готов ждать этого события целую вечность. Именно эта

константа обычно и передается функции WaitForSingleObject, но Вы можете указать любое значение в миллисекундах.

Вот пример, иллюстрирующий, как вызывать WaitForSingleObject со значением таймаута, отличным от INFINITE

```
DWORD dw = WaitForSlngleObject(hProcess, 5000);
switch (dw)
{
     case WAIT_OBJECT_0:
          // процесс завершается
          break;
     case WAIT_TIMEOUT:
          // процесс не завершился в течение 5000 мс
          break;
     case WAIT_FAILED:
          // неправильный вызов функции (неверный описатель?)
          break;
}
```

Функция WaitForMultipleObjects аналогична WaitForSingleObject с тем исключением, что позволяет ждать освобождения сразу нескольких объектов или какого-то одного из списка объектов:

```
DWORD WaitForMultipleObjects( DWOHD dwCount,
                              CONST HANDLE* phObjects,
                              BOOL fWaitAll,
                              DWORD dwMilliseconds);
```

Параметр dwCount определяет количество интересующих Вас объектов ядра Его значение должно быть в пределах от 1 до MAXIMUM_WAIT_OBJECTS (в заголовочных файлах Windows оно определено как 64). Параметр phObject — это указатель на массив описателей объектов ядра.

WaitForMultipleObjects приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр fWaitAll как раз и определяет, чего именно Вы хотите от функции. Если он равен TRUE, функция не даст потоку возобновить свою работу, пока не освободятся все объекты.

Параметр dwMilliseconds идентичен одноименному параметру функции WaitFor SingleObject Если Вы указываете конкретное время ожидания, то по его истечении функция в любом случае возвращает управление. И опять же, в этом параметре обычно передают INFINITE (будьте внимательны при написании кода, чтобы не создать ситуацию взаимной блокировки).

Возвращаемое значение функции WaitForMultipleObjects сообщает, почему возобновилось выполнение вызвавшего ее потока Значения WAIT_FAILED и WAIT_TIMEOUT никаких пояснений не требуют. Если Вы передали TRUE в параметре fWaitAll и все объекты перешли в свободное состояние, функция возвращает значение WAIT_OB JECT_0. Если fWaitAll

приравнен FALSE, она возвращает управление, как только ос вобождается любой из объектов. Вы, по-видимому, захотите выяснить, какой именно объект освободился В этом случае возвращается значение от WAIT_OBJECT_0 до WAIT_OBJECT_0 + dwCount - 1. Иначе говоря, если возвращаемое значение не равно WAIT_TIMEOUT или WAIT_FAILED, вычтите из него значение WAlT_OBJECT_0, и Вы получите индекс в массиве описателей, на который указывает второй параметр фун кции WaitForMultipleObjects.

События

События - самая примитивная разновидность объектов ядра. Они содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая — его состояние (свободен или занят).

События просто уведомляют об окончании какой-либо операции. Объекты - события бывают двух типов: со сбросом вручную (manual-reset events) и с автосбросом (auto-reset events). Первые позволяют возобновлять выполнение сразу нескольких ждущих потоков, вторые — только одного.

Объекты-события обычно используют в том случае, когда какой-то поток выполняет инициализацию, а затем сигнализирует другому потоку, что тот может продол жить работу. Инициализирующий поток переводит объект "событие» в занятое состояние и приступает к своим операциям. Закончив, он сбрасывает событие в свободное состояние. Тогда другой поток, который ждал перехода события в свободное состояние, пробуждается и вновь становится планируемым.

Объект ядра «событие" создается функцией CreateEvent:

```
HANDLE CreateEvent( PSECURITY_ATTRIBUTES psa,
                    BOOL fManualReset,
                    BOOL fInitialState,
                    PCTSTR pszName);
```

Параметр fManualReset (булева переменная) сообщает системе, хотите Вы создать событие со сбросом вручную (TRUE) или с автосбросом (FALSE). Параметру fInitialState определяет начальное состояние события — свободное (TRUE) или занятое (FALSE). После того как система создает объект событие, CreateEvent возвращает описатель события, специфичный для конкретного процесса. Потоки из других процессов могут получить доступ к этому объекту: 1) вызовом CreateEvent с тем же параметром pszName;, 2) наследованием описателя; 3) применением функции DuplicateHandle;, и 4) вызовом OpenEvent с передачей в параметре pszName имени, совпадающего с указанным в аналогичном параметре функции CreateEvent. Вот что представляет собой функция OpenEvent.

```
HANDLE OpenEvent( DWORD fdwAccess, BOOL fInhent, PCTSTR pszName);
```

Чтобы перевести объект в свободное состояние, Вы вызываете:

```
BOOL SetEventT(HANDLE hEvenet);
```

А чтобы поменять его на занятое

```
BOOL ResetEvent(HANDLE hEvent);
```

Ожидаемые таймеры

Ожидаемые таймеры (waitahle timers) ~ это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Чтобы создать ожидаемый таймер, достаточно вызвать функцию CreateWaitableTimer.

```
HANDLE CreateWaitableTimer(    PSECURITY_ATTRIBUTES psa,
                               BOOL fManualReset,
                               PCTSTR pszName);
HANDLE OpenWaitableTirrer(     DWORD dwDesiredAccess,
                               BOOL bInheritHandle,
                               PCTSTR pszName);
```

По аналогии с событиями параметр fManualReset определяет тип ожидаемого таймера: со сбросом вручную или с автосбросом. Когда освобождается таймер со сбросом вручную, возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом — лишь одного из потоков.

Объекты «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, вызови те функцию SetWaitableTimer.

```
BOOL SetWaitableTimer(    HANDLE hTimer,
                          const LARGE_INTEGER *pDueTime,
                          LONG lPeriod,
                          PTIMERAPCROUTINE pfnCompletionRoutine,
                          PVOID pvArgToCotnpletionRoutine,
                          BOOI fResume);
```

Эта функция принимает несколько параметров. Очевидно, что hTimer определяет нужный таймер. Следующие два параметра (pDieTite и lPeriod) используются совместно, первый из них задает, когда таймер должен сработать в первый раз, второй определяет, насколько часто это должно происходить в дальнейшем

Другая функция -

```
BOOL CancelWaitableTimer(HANDLE hTimer);
```

Эта очень простая функция принимает описатель таймера и отменяет его (таймер), после чего тот уже никогда не сработает, — если только Вы не переустановите его повторным вызовом SetWaitableTimer. Если Вам понадобится перенастроить таймер, то вызывать CancelWattableTimer перед повторным обращением к SetWaitableTimer не требуется; каждый вызов SetWaitableTimer автоматически отменяет предыдущие настройки перед установкой новых.

Семафоры

Объекты ядра «семафор» используются для учета ресурсов. Как и все объекты ядра, они содержат счетчик числа пользователей, но, кроме того, поддерживают два 32 битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), другое используется как счетчик текущего числа ресурсов

Объект ядра «семафор» создается вызовом CreateSemapbore

```
HANDLE CreateSemaphore(  PSECURITY_ATTRIBUTE psa,
                         LONG lInitialCount,
                         LONG lMaximumCount,
                         PCTRTR pszName);
```

Описатель существующего объекта «семафор» можно получить, вызвав OpenSemaphore

```
HANDLE OpenSemaphore(    DWORD fdwAccess,
                         BOOL bInhentHandle,
                         PCTSTR pszName);
```

Параметр lMaximumCount сообщает системе максимальное число ресурсов, обрабатываемое Вашим приложением. Поскольку это 32-битное значение со знаком, пре дельное число ресурсов может достигать 2 147 483 647 Параметр lInitia1Coimt указывает, сколько из этих ресурсов доступно изначально (на данный момент).

Поток получает доступ к ресурсу, вызывая одну из Wait-функций и передавая ей описатель семафора, который охраняет этот ресурс Wait-функция проверяет у сема фора счетчик текущего числа ресурсов если его значение больше 0 (семафор свободен), уменьшает значение этого счетчика на 1, и вызывающий поток остается планируемым Очень важно, что семафоры выполняют эту операцию проверки и присвоения на уровне атомарного доступа; иначе говоря, когда Вы запрашиваете у семафора какой-либо ресурс, операционная система проверяет, доступен ли этот ресурс, и, если да, уменьшает счетчик текущего числа ресурсов, не позволяя вмешиваться в эту операцию другому потоку. Только после того как счетчик ресурсов будет уменьшен на 1, доступ к ресурсу сможет запросить другой поток

Если Wait-функция определяет, что счетчик текущего числа ресурсов равен 0 (семафор занят), система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время (а он, захватив ресурс, уменьшит значение счетчика на 1).

Поток увеличивает значение счетчика текущего числа ресурсов, вызывая функцию ReleaseSemaphore

```
BOOL ReleaseSemaphore(   HANDLE hSem,
                         LONG lReleaseCount,
                         PLONG p]PreviousCount);
```

Она просто складывает величину lReleaseCount со значением счетчика текущего числа ресурсов. Обычно в параметре lReleaseCount передают 1, но это вовсе не обязательно. Функция возвращает исходное значение счетчика ресурсов в *plPreviousCount. Если Вас не интересует это, передайте в параметре plPre viousCount значение NULL.

Мьютексы

Объекты ядра «мьютексы» гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Мьютексы ведут себя точно так же, как и критические секции. Однако, если последние являются объектами пользовательского режима, то мьютексы — объектами ядра. Кроме того, единственный объект-мьютекс позволяет синхронизировать доступ к ресурсу нескольких потоков из разных процессов; при этом можно задать максимальное время ожидания доступа к ресурсу.

Идентификатор потока определяет, какой поток захватил мьютекс, а счетчик рекурсий — сколько раз. У мьютексов много применений, и это наиболее часто используемые объекты ядра. Как правило, с их помощью защищают блок памяти, к которому обращается множество потоков. Если бы потоки одновременно использовали какой-то блок памяти, данные в нем были бы повреждены. Мьютексы гарантируют, что любой поток получает монопольный доступ к блоку памяти, и тем самым обеспечивают целостность данных.

Для использования объекта-мьютекса один из процессов должен сначала создать его вызовом CreateMutex:

```
HANDLE CreateMutex( PSECURITY_ATTRIBUTES psa,
                    BOOL fIniLialOwner,
                    PCTSTR pszName);
```

Разумеется, любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «мьютекс», вызвав OpenMutex:

```
HANDLE OpenMutex(   DWORD fdwAccess,
                    B00L bInheritHandle,
                    PCTSTR pszName);
```

Параметр fInitialOwner определяет начальное состояние мъютекса. Если в нем передается FALSE (что обычно и бывает), объект-мьютекс не принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока и счетчик рекурсии равны 0. Если же в нем передается TRUE, идентификатор потока, принадлежащий мьютексу, приравнивается идентификатору вызывающего потока, а счетчик рекурсии получает значение 1. Поскольку теперь идентификатор потока отличен от 0, мьютекс изначально находится в занятом состоянии.

Поток получает доступ к разделяемому ресурсу, вызывая одну из Wait-функций и передавая ей описатель мьютекса, который охраняет этот ресурс. Wait-функция проверяет у мьютекса идентификатор потока, если его значение не равно 0, мьютекс свободен, в ином случае оно принимает значение идентификатора вызывающего потока, и этот поток остается планируемым.

Если Wait-функция определяет, что у мьютекса идентификатор потока не равен 0 (мьютекс занят), вызывающий поток переходит в состояние ожидания. Система запоминает это и, когда идентификатор обнуляется, записывает в него идентификатор ждущего потока, а счетчику рекурсии присваивает значение 1, после чего ждущий поток вновь становится планируемым. Все проверки и изменения состояния объекта-мьютекса выполняются на уровне атомарного доступа.

Когда ожидание мьютекса потоком успешно завершается, последний получает монопольный доступ к защищенному ресурсу. Все остальные потоки, пытающиеся обратиться к этому ресурсу, переходят в состояние ожидания Когда поток, занимающий ресурс, заканчивает с ним работать, он должен освободить мьютекс вызовом функции ReleaseMutex

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Эта функция уменьшает счетчик рекурсии в объекте-мьютексе на 1. Если данный объект передавался во владение потоку неоднократно, поток обязан вызвать ReleaseMutex столько раз, сколько необходимо для обнуления счетчика рекурсии. Как только счетчик станет равен 0, переменная, хранящая идентификатор потока, тоже обнулится, и объект-мьютекс освободится. После этого система проверит, ожидают ли освобождения мьютекса какие-

нибудь другие потоки. Если да, система «по-честному» выберет один из ждущих потоков и передаст ему во владение объект-мьютекс.

Литература по лабораторным работам 3-4

1. Дж. Рихтер. Windows для профессионалов (Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows). Четвертое издание. М: "Русская Редакция"; пер. с англ. - СПб: "Питер", 2001.

2. Дж. Рихтер. Windows для профессионалов (Программирование в Win32 API для Windows NT 3.5 и Windows 95). Второе издание. М: "Русская Редакция", 1995.

3. Д. Соломон, М. Руссинович. Внутреннее устройство Microsoft Windows 2000. М: "Русская Редакция"; пер. с англ. - СПб: "Питер", 2001.

4. Дж. Рихтер, Дж. Кларк. Программирование серверных приложений для Microsoft Windows 2000. М: "Русская Редакция"; пер. с англ. - СПб: "Питер", 2001.

5. Ал Вильямс Системное программирование в Windows 2000 для профессионалов.СПб: Питер, 2001.

6. А. М. Робачевский. Операционная система UNIX. СПб: BHV - Санкт-Петербург, 1998.

7. A.D. Marshall. Programming in C UNIX System Calls and Subroutines using C. 1999. (Электронный источник – http://www.cs.cf.ac.uk/Dave/C/CE.html)

Лабораторная работа 5. Файловые системы

Цель работы – реализация доступа к линейному пространству данных как к иерархической файловой системе.

Данная работа предполагает два варианта исполнения:

а) создание программного блока, обеспечивающего работу с файловой системой, хранящейся в файле;

б) создание модулей для ОС UNIX, обеспечивающих подключение к иерархии файловой системы раздела с файловой системой, хранящегося в файле.

Постановка задачи для конкретного студента включает выбор варианта исполнения и параметры файловой системы.

В ходе выполнения лабораторной работы студент должен решить следующие задачи.

1. Изучить архитектуру драйвера файловой системы и его место в подсистеме управления файлами.

2. Определить обрабатываемые драйвером запросы и используемые функции доступа к линейному пространству данных.

3. Изучить предложенный тип файловой системы.

4. Реализовать модуль драйвера файловой системы.

5. Реализовать модуль инициализации файловой системы.

6. Выполнить компиляцию и сборку модулей.

7. Провести сравнительные эксперименты работы с файловыми системами различных типов с использованием соответствующих модулей.

Предлагаемые к реализации файловые системы

Во всех рассматриваемых файловых системах место на диске выделяется блоками. Размер блока, как правило, совпадает с аппаратным размером сектора (512 байт у большинства дисковых устройств), однако многие файловые системы могут использовать логические блоки, состоящие из нескольких секторов (так называемые кластеры).

Использование блоков и кластеров вместо адресации с точностью до байта обусловлено двумя причинами. Во-первых, у большинства устройств произвольного доступа доступ произволен лишь с точностью до сектора, т.е. нельзя произвольно считать или записать любой байт – нужно считывать или записывать весь сектор целиком. Во вторых, такой способ позволяет значительно расширить адресуемое пространство.

Файлы, хранящиеся в файловой системе можно разделить на два типа – пользовательские файлы и системные файлы (к которым относятся, например, директории).

Файловая система 1

Наиболее простой файловой системой можно считать структуру представленную на рисунке.

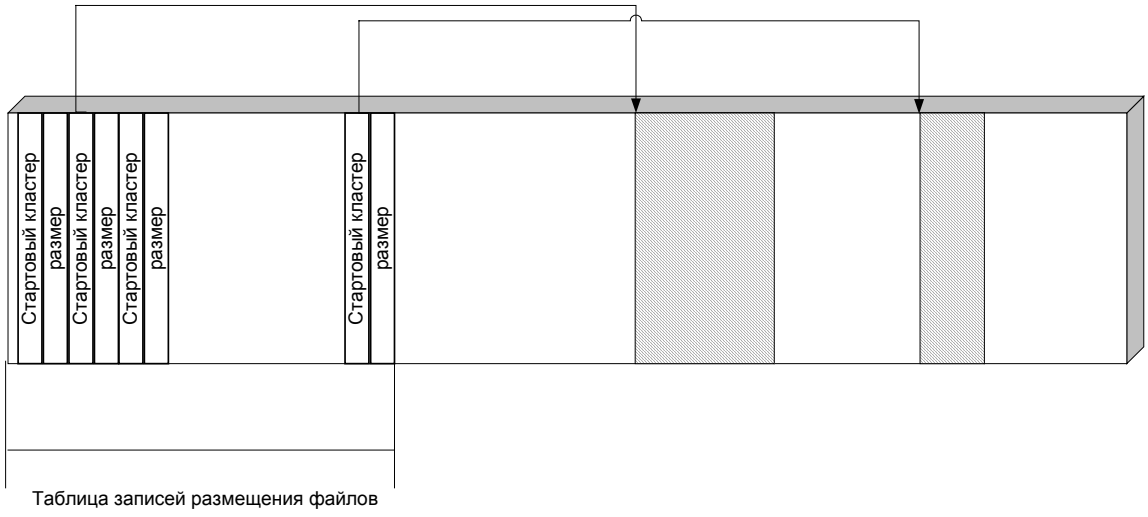


Рис. 58 Файловая система 1

В начале диска располагается таблица записей размещения файлов. Количество записей таблицы задается при форматировании файловой системы и не может быть изменено. Количество записей в таблице определяет число файлов, которые могут быть созданы. Каждому файлу выделяется непрерывная область на диске. Благодаря этому в записи таблицы, соответствующей файлу, достаточно хранить адрес первого кластера файла и его длину, также измеренную в кластерах.

Нулевая и первая запись в таблице зарезервированы. Нулевая запись представляет собой файл корневого каталога. Первая – файл, содержащий список элементов таблицы ссылающихся на свободные участки диска.

Каталог (директория) является файлом специального вида. Он содержит записи описывающие файлы входящие в каталог. Каждая запись представляет собой 32-байтовую структуру вида:

байт	0-7	имя файла
	8-10	расширение файла
	11	атрибут файла
	12-21	зарезервировано
	22-23	время последнего доступа к файлу
	24-25	дата последнего доступа к файлу
	26-27	индекс в таблице записей размещения файлов
	28-31	размер файла

Атрибут файла в частности определяет является ли файл подкаталогом (если младший бит = 1, то это подкаталог). Размер файла указывает точную длину файла в байтах (поскольку файл обычно не целиком занимает отведенный ему последний кластер).

Каталог свободных элементов имеет такую же структуру, однако в ней используется только индекс в таблице записей размещения файлов. В случае если свободного места нет, то элемент таблицы размещения файлов с индексом один будет содержать ноль в поле размер (каталог свободных элементов пуст).

Файловая система 2

Для работы этой файловой системы используется таблица, обычно называемая FAT (File Allocation Table, таблица размещения файлов), расположенная в начале диска. Таблица создается при форматировании файловой системы. В этой таблице каждому блоку (кластеру), предназначенному для хранения данных, соответствует 16-битовое значение. Если блок свободен, то значение будет нулевым. Если же блок принадлежит файлу, то значение равно индексу следующего блока этого файла в таблице размещения файлов. Если это последний блок в файле, то значение – FFFF. Запись с индексом ноль является первой записью файла, содержащего корневую директорию.

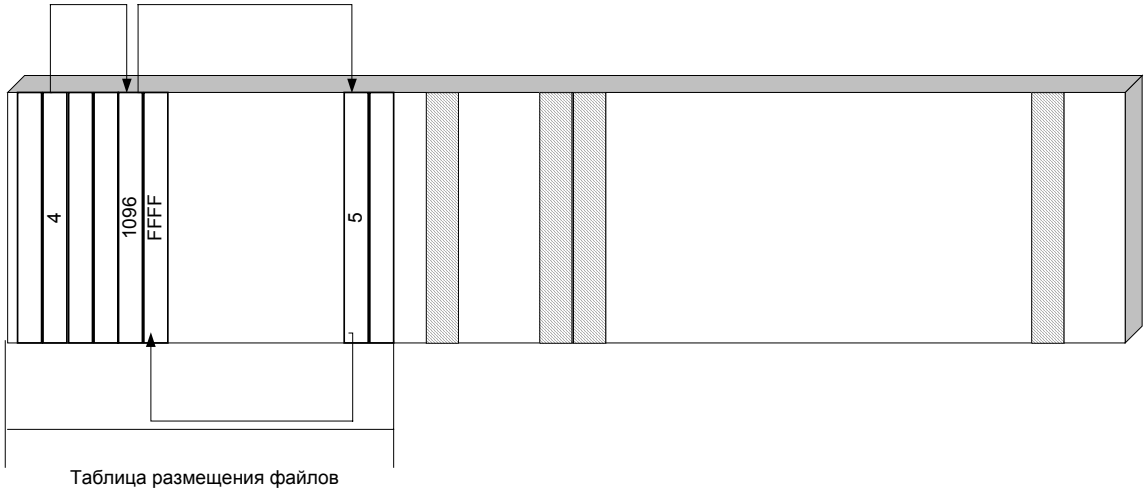


Рис. 59 Файловая система 2

Каталог (директория) является файлом специального вида. Он содержит записи описывающие файлы входящие в каталог. Каждая запись представляет собой 32-байтовую структуру вида:

байт	0-7	имя файла
	8-10	расширение файла
	11	атрибут файла
	12-21	зарезервировано
	22-23	время последнего доступа к файлу
	24-25	дата последнего доступа к файлу
	26-27	индекс первого кластера файла в таблице размещения файлов
	28-31	размер файла

Атрибут файла в частности определяет является ли файл подкаталогом (если младший бит = 1, то это подкаталог). Размер файла указывает точную длину файла в байтах (поскольку файл обычно не целиком занимает отведенный ему последний кластер).

Файловая система 3

Предполагается, что пространство разбито на участки равного размера (кластеры). Файл может занимать несколько несмежных областей. Области, занимаемые файлом, однозначно определяются *файловой записью*. Файловая запись представляет структуру следующего вида:

байт		экстент1
		экстент2
		экстент16

Каждый экстент представляет собой запись из двух полей – первое поле – адрес начала области (номер кластера), которую занимает файл, второе поле – размер этой области в кластерах. Файловые записи собраны в таблицу, размещаемую в начале диска. Размер таблицы задается при форматировании файловой системы и не изменяется.

В случае, если шестнадцати областей оказывается недостаточно для размещения файла, то из экстента может быть организована ссылка на файловую запись, содержащую "продолжение" файла. Для этого поле размер устанавливается в 0, при этом значение в первом поле трактуется как индекс в таблице файловых записей. Таким образом, массивы экстентов, описывающих размещение файлов, организованы в виде дерева.

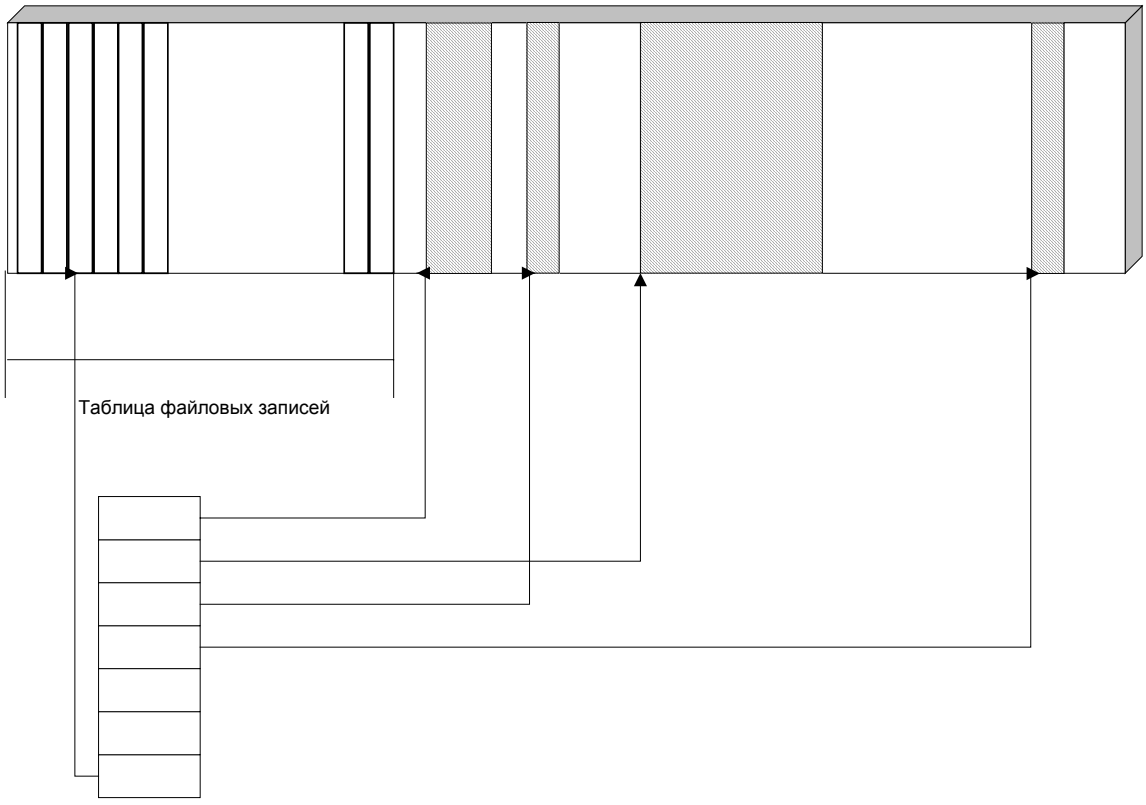


Рис. 60 Файловая система 3

Каталог (директория) является файлом специального вида. Он содержит записи, описывающие файлы, входящие в каталог. Каждая запись представляет собой 32-байтовую структуру вида:

байт	0-7	имя файла
	8-10	расширение файла
	11	атрибут файла
	12-21	зарезервировано

22-23	время последнего доступа к файлу
24-25	дата последнего доступа к файлу
26-27	индекс первой файловой записи в таблице файловых записей
28-31	размер файла

Атрибут файла в частности определяет является ли файл подкаталогом (если младший бит = 1, то это подкаталог). Размер файла указывает точную длину файла в байтах (поскольку файл обычно не целиком занимает отведенный ему последний кластер).

Файловая запись с индексом ноль содержит корневой каталог.

Симулятор работы с файловой системой

Постановка задачи

В рамках лабораторной работы ставится задача создания файловой системы, поддерживающей следующие операции:

Операции над файлами

- Create. Создание файла, не содержащего данных. Смысл данного вызова - объявить, что файл существует и присвоить ему ряд атрибутов по умолчанию.

- Delete. Удаление файла и освобождение занятого им дискового пространства.

- Open. Перед использованием файла процесс должен его открыть. Цель данного системного вызова разрешить системе проанализировать атрибуты файла и проверить права доступа к файлу, а также считать в оперативную память список адресов блоков файла для быстрого доступа к его данным.

- Close. Если работа с файлом завершена, его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы.

- Seek. Дает возможность специфицировать место внутри файла, откуда будет производиться считывание (или запись) данных, то есть задать текущую позицию.

- Read. Чтение данных из файла. Обычно это происходит с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить буфер для них.

- Write. Запись данных в файл с текущей позиции. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

- Get attributes. Предоставляет процессам нужные им сведения об атрибутах файла

- Set attributes. Дает возможность пользователю установить некоторые атрибуты. Наиболее очевидный пример - установка режима доступа к файлу.

- Rename. Данная операция может быть смоделирована копированием данного файла в файл с новым именем и последующим его удалением.

Операции над директориями

- Create. Создание директории.

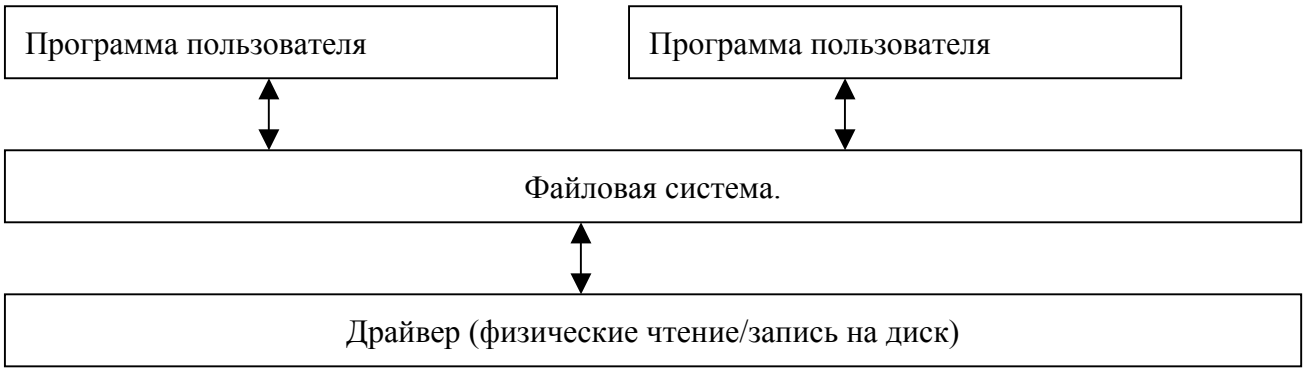
- Delete. Удаление директории. Удалена может быть только пустая директория.

- Files. Возвращает список файлов (и директорий), содержащихся в данной директории

- Rename. Имена директорий можно менять, также как и имена файлов.

Требования к лабораторной работе

Лабораторная работа должна быть выполнена с использованием программной лаборатории «Симулятор файловых систем». Программная лаборатория представляет собой набор подпрограмм и классов, реализованных на языке C++. Если рассмотреть пример типичного использования файловой системы, то можно выделить три уровня модулей, участвующих во взаимодействии



Целью лабораторной работы является реализация второго уровня – файловой системы, в то время как первый уровень (программы пользователя) и третий уровень («драйвер») поставляются в рамках программной лаборатории.

Пользовательские программы, включенные в программную лабораторию можно разделить на два типа – к первому относятся тесты файловой системы, каждый из которых проверяет тот или иной аспект функционирования файловой системы. К другому типу относится броузер файловой системы, который позволяет просмотреть структуру директорий. Тесты представляют собой консольные приложения. Броузер реализован в виде оконного приложения, представляющего структуру директорий в виде дерева. Сбор статистики по основным операциям (количество позиционирований, объем считанных/записанных байтов) производится «драйвером».

Таким образом, задача слушателя состоит в том, чтобы реализовать собственный класс, реализующий функциональность файловой системы определенного типа. Класс файловой системы должен быть потомком абстрактного класса VFS (`class VFS`). Описание класса предоставлено в сопровождающей документации.

Работа с программной лабораторией может осуществляться по следующему сценарию:

1. Получение задания на разработку файловой системы того или иного типа.

2. Реализация и оптимизация класса, реализующего функциональность файловой системы. Класс – потомок абстрактного класса VFS.Устранение ошибок.

3. Компиляция проекта, устранение ошибок.

4. Компиляция тестов.

5. Запуск тестов. В случае отрицательного результата теста – возврат к шагу 2.

6. Изучение файла статистики дисковых операций по результатам тестов. Сравнение с эталонными результатами.

7. Компиляция броузера реализованной файловой системы.

Архитектура программной лаборатории

Программная лаборатория предоставляет набор классов и подпрограмм, предназначенных для использования слушателем при создании симулятора файловой системы. Кроме того, она включает в себя средства тестирования и изучения файловых систем, реализованных слушателями.

Функционально программная лаборатория состоит из следующих логических блоков:

1) Блок визуализации – позволяет просмотреть структуру директорий предоставленной файловой системы.
2) Блок симуляции– реализуется слушателем. Представляет собой модуль, реализующий стандартные операции файловой системы.
3) Блок тестирования – включает в себя набор тестов для проверки правильности работы предоставленного модуля файловой системы
4) Блок статистики – отвечает за накопление результатов тестов.

Сценарий работы с программной лабораторией.

Программная лаборатория поставляется в виде структуры директорий, содержащих исходные тесты.

FSLab

|_____lab_viewer // Содержит код приложения, визуализирующего структуру директорий

|_____Tests // Содержит набор тестов

|_____FS // Содержит набор абстрактных классов FS, модуль работы с диском и модуль,

| //собирающий статистику.

|_____Docs // Некоторая документация

Задача слушателя состоит в том, чтобы реализовать собственный класс, реализующий функциональность файловой системы определенного типа. После чего осуществляется перекомпиляция исходных кодов программной лаборатории вместе с реализованным слушателем модулем.

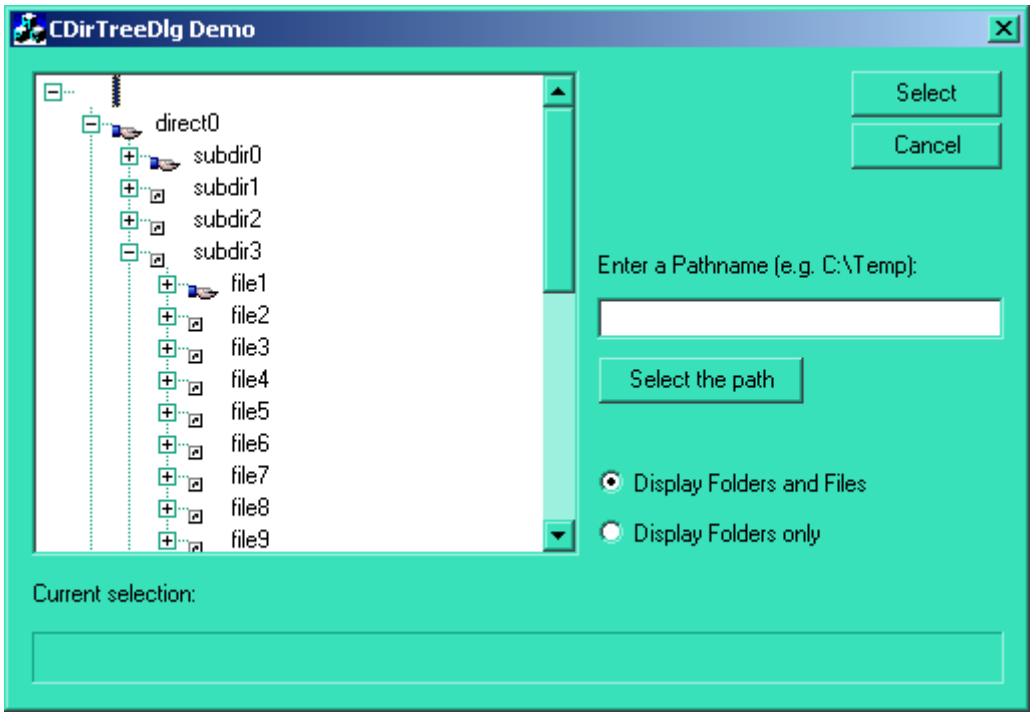


Рис. 61Окно программы – броузера файловой системы

Полученный симулятор файловый системы должен быть протестирован с помощью поставляемого вместе с лабораторией набора тестов на корректность работы. Кроме того, в результате тестирования собирается статистка по операциям чтения/записи, выполняемым модулем файловой системы – что может использоваться для оценки эффективности работы реализованного слушателем модуля.

Каталог FS содержит следующие файлы:

Driver.h	Содержит описание класса Driver, используемого для реализации операций чтения/записи на диск
Driver.cpp	Реализация класса Driver
VFS1.h	Содержит описание абстрактного класса VFS, который является базовым для реализуемых файловых систем

Класс слушателя, реализующий файловую систему, должен называться FS и располагаться в файлах FS.h и FS.cpp соответственно.

Каталог tests содержит следующие тесты

Тест1. Запись файла в файловую систему, чтение файла из файловой системы, сравнение записанного и прочитанного файлов.

Тест2. Последовательное создание 10 директорий, в каждой из директорий 10 поддиректорий, в каждой из них – 10 файлов размером по 50 байт. Чтение списков директорий. Сравнения списка записанных и прочитанных файлов.

Тест3. Несколько циклов создания/удаления большого количества мелких файлов.

Обзор архитектуры модуля поддержки файловой системы в Linux

Роль драйвера файловой системы состоит в выполнении низкоуровневых задач, используемых для распределения высокоуровневых операций VFS на физических устройствах. Интерфейс VFS достаточно универсален.

Драйвер создается из следующих компонентов, принадлежащих каталогу исходных текстов файловой системы или каталогу исходных текстов VFS:

1. Описание суперблока файловой системы - include/linux/type_fs.h описывает информацию в том виде, в каком она хранится на диске, include/linux/type_fs_sb.h содержит описание структуры, которая добавляется к стандартной структуре ядра, описывающей суперблок.

2. Описание inode - include/linux/type_fs_i.h содержит описание структуры inode на диске и структуры, которая добавляется к стандартной структуре ядра, описывающей inode.

3. Основной include файл - include/linux/type_fs.h.

4. Исходные тексты функций работы с inode - fs/fstype/inode.c. Структура inode_operations используется для операций с inode функциями VFS.

5. Исходные тексты функций работы с объектами файловой системы разных типов – файлами, каталогами, символьными ссылками и т.д. - fs/fstype/file.c, dir.c, symlink.c и т.д. Структуры типа struct file_operations используется для этих операций функциями VFS.

6. Операции, управляющие распределением и освобождением inode и блоков хранения данных - fs/fstype/bitmap.c

7. Операции с inode, связанные с именами, такие как создание и удаление имени, относящегося к inode, переименование и т.п.

Во время подключения файловой системы fstype_read_super заполняет структуру super_block информацией, полученной с поддерживаемого устройства. Поле s_op структуры содержит указатель на fstype_sops, используемый кодом VFS для выполнения операций над суперблоком.

Отключение происходит с помощью do_umount, включающем запуск fstype_put_super.

При доступе к файлу, fstype_read_inode заполняет общую системную inode структуру полями из fstype_inode. Поле inode.i_op заполняется указателями на реализации функций обработки inode для данной файловой системы.

Рассмотрение пунктов проще всего произвести на примере. Используем для этого файловую систему Minix.

Модули драйвера файловой системы Minix

Описание суперблока файловой системы

Формат хранения суперблока файловой системы на внешнем носителе содержится в файле include/linux/minix_fs.h. Суперблок содержит общую информацию о файловой системе – число файловых дескрипторов и т.д.

```
struct minix_super_block {
        __u16 s_ninodes;
        __u16 s_nzones;
        __u16 s_imap_blocks;
        __u16 s_zmap_blocks;
        __u16 s_firstdatazone;
        __u16 s_log_zone_size;
        __u32 s_max_size;
```

```
        __u16 s_magic;
        __u16 s_state;
        __u32 s_zones;
};
```

При монтировании файловой системы в ядре создается суперблок VFS для управления файловой системой. Его формат определен в файле linux/fs.h

```
struct super_block {
        struct list_head        s_list;
        kdev_t                  s_dev;
        unsigned long           s_blocksize;
        unsigned char           s_blocksize_bits;
        unsigned char           s_dirt;
          …
        union {
                struct minix_sb_info    minix_sb;
                struct ext2_sb_info     ext2_sb;
                struct ext3_sb_info     ext3_sb;
                struct msdos_sb_info    msdos_sb;
                    …
        } u;
};
```

Одним из полей структуры является объединение (union u;), предназначенное для хранения информации, зависимой от конкретного типа файловой системы. Состав и формат этих данных определяются в файле include/linux/ minix_fs_sb.h

```
struct minix_sb_info {
        unsigned long s_ninodes;
        unsigned long s_nzones;
        unsigned long s_imap_blocks;
        unsigned long s_zmap_blocks;
        unsigned long s_firstdatazone;
        unsigned long s_log_zone_size;
        unsigned long s_max_size;
        int s_dirsize;
        int s_namelen;
        int s_link_max;
        struct buffer_head ** s_imap;
        struct buffer_head ** s_zmap;
```

```
        struct buffer_head * s_sbh;
        struct minix_super_block * s_ms;
        unsigned short s_mount_state;
        unsigned short s_version;
};
```

Описание индексного дескриптора (inode)

Файл include/linux/minix_fs_i.h должен содержит описание структуры, которая добавляется к стандартной структуре ядра, описывающей индексный дескриптор.

```
struct minix_inode_info {
        union {
                __u16 i1_data[16];
                __u32 i2_data[16];
        } u;
};
```

Основной файл заголовков

Основной файл заголовков include/linux/minix_fs.h содержит следующие элементы

- определения констант, используемых при работе с файловой системой;

- описание формата хранения индексного дескриптора на диске

```
struct minix2_inode {
        __u16 i_mode;          // права доступа
        __u16 i_nlinks;        // число жестких ссылок (имен)
        __u16 i_uid;           // идентификатор владельца
        __u16 i_gid;           //идентификатор группового владельца
        __u32 i_size;          // размер
        __u32 i_atime;         // время последнего доступа
        __u32 i_mtime;         // время последнего изменения inode
        __u32 i_ctime;         // время последнего изменения данных
        __u32 i_zone[10];      // описания расположения файла на ВН
};
```

- описание формата хранения суперблока файловой системы на внешнем носителе (как мы уже сказали выше);

- описание формата хранения записи о дочернем объекта в директории (обратите внимание: в директории хранится только имя объекта файловой системы и номер его индексного дескриптора, все остальное хранится в индексном дескрипторе)

```
struct minix_dir_entry {
        __u16 inode;
```

```
        char name[0];
};
```

- прототипы всех функций работы с файловой системой, которые могут потребоваться подсистеме VFS;

- объявление структур, содержащих адреса функций работы с элементами файловой системы; эти структуры объявляются в файлах include/linux/file.c, include/linux/dir.c, include/linux/namei.c, используются при присвоении значений полям индексного дескриптора VFS: inode->i_op (операции над inode) и inode->f_op (операции над объектом файловой системы; впоследствии указанные поля используются подсистемой VFS для вызова функций чтения/записи файла и т.д.;

```
extern struct inode_operations minix_file_inode_operations;
extern struct inode_operations minix_dir_inode_operations;
extern struct file_operations minix_file_operations;
extern struct file_operations minix_dir_operations;
extern struct dentry_operations minix_dentry_operations;
```

Исходные тексты функций работы с индексным дескриптором

Файл fs/minix/inode.c содержит определение функций работы с индексным дескриптором (создание, удаление, чтение, запись и т.д.), функции для работы с суперблоком и определение структуры, в которой эти функции регистрируются.

```
static struct super_operations minix_sops = {
        read_inode:     minix_read_inode,
        write_inode:    minix_write_inode,
        delete_inode:   minix_delete_inode,
        put_super:      minix_put_super,
        write_super:    minix_write_super,
        statfs:         minix_statfs,
        remount_fs:     minix_remount,
};
```

Также в данном файле определены функции блочного ввода-вывода и структура с их перечислением.

```
static struct address_space_operations minix_aops = {
        readpage: minix_readpage,
        writepage: minix_writepage,
        sync_page: block_sync_page,
        prepare_write: minix_prepare_write,
        commit_write: generic_commit_write,
        bmap: minix_bmap
};
```

Исходные тексты функций работы с объектами ФС разных типов

Исходные тексты функций работы с файлами содержатся в файле fs/minix/file.c, каталогами - fs/minix/dir.c и fs/minix/namei.c. Набор реализованных операций является типичным.

```
struct file_operations minix_file_operations = {
        llseek:         generic_file_llseek,
        read:           generic_file_read,
        write:          generic_file_write,
        mmap:           generic_file_mmap,
        fsync:          minix_sync_file,
};
struct file_operations minix_dir_operations = {
        read:           generic_read_dir,
        readdir:        minix_readdir,
        fsync:          minix_sync_file,
};
struct inode_operations minix_dir_inode_operations = {
        create:         minix_create,
        lookup:         minix_lookup,
        link:           minix_link,
        unlink:         minix_unlink,
        symlink:        minix_symlink,
        mkdir:          minix_mkdir,
        rmdir:          minix_rmdir,
        mknod:          minix_mknod,
        rename:         minix_rename,
};
```

Другие модули

Функции, управляющие распределением и освобождением inode и блоков хранения данных определены в fs/minix/bitmap.c.

Операции с inode, связанные с именами, такие как создание и удаление имени, относящегося к inode, переименование и им подобные определены в fs/minix/namei.c.

Реализация драйвера файловой системы

При реализации драйвера файловой системы вы должны разработать структуру хранения ФС на диске и реализовать поддерживающий ее код. Простейший путь решения подобной задачи – модификация уже существующей реализации.

Объем исходных текстов драйвера файловой системы Minix составляет примерно 75Kb (~2000 строк, включая комментарии). Он поддерживает практически минимальный набор операций над объектами ФС и его код достаточно легок для понимания. Вы можете даже не добавлять новый тип файловой системы, а просто заменить реализацию MINIX.

Литература по лабораторной работе 5

1. Э. Таненбаум. Современные операционные системы. 2-е издание. СПб: Питер, 2002.

2. К. Грегори «Использование Visual C++ 6. Специальное издание»: Пер. с англ. – М.; СПб.; К.: Издательский дом «Вильямс», 2002.

3. David Rusling. The Linux Kernel (Электронный источник - www.linuxdoc.org/LDP/tlk/)

4. Cross-Referencing Linux (Электронный источник - lxr.linux.no)

Литература

Основная литература

1. Э. Таненбаум. Современные операционные системы. 2-е издание. СПб: Питер, 2002.
2. В.Е. Карпов, К.А. Коньков. Введение в операционные системы. Курс лекций. (Электронный источник – http://cs.mipt.ru/docs/courses/osstud/os.html)
3. А. В. Гордеев, А. Ю. Молчанов. Системное программное обеспечение. СПб: "Питер", 2001.
4. А.В. Гордеев. Операционные системы. СПб: «Питер», 2004.
5. В. Г. Олифер, Н. А. Олифер. Сетевые операционные системы. СПб: "Питер", 2001.
6. Дж. Рихтер. Windows для профессионалов (Программирование в Win32 API для Windows NT 3.5 и Windows 95). Второе издание. М: "Русская Редакция", 1995.
7. Дж. Рихтер. Windows для профессионалов (Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows). Четвертое издание. М: "Русская Редакция"; пер. с англ. - СПб: "Питер", 2001.
8. Дж. Рихтер, Дж. Кларк. Программирование серверных приложений для Microsoft Windows 2000. М: "Русская Редакция"; пер. с англ. - СПб: "Питер", 2001.
9. Ал Вильямс Системное программирование в Windows 2000 для профессионалов.СПб: Питер, 2001.А. М. Робачевский. Операционная система UNIX. СПб: BHV - Санкт-Петербург, 1998.
10. Д. Соломон, М. Руссинович. Внутреннее устройство Microsoft Windows 2000. М: "Русская Редакция"; пер. с англ. - СПб: "Питер", 2001.
11. Mel Gorman. Understanding The Linux Virtual Memory Manager. 2003. (Электронный источник – http://www.csn.ul.ie/~mel/projects/vm/guide/html/understand/)
12. Robert Love. Linux Kernel Development. SAMS, 2003.
13. David Rusling. The Linux Kernel (Электронный источник – http://www.linuxdoc.org/LDP/tlk/)
14. Скотт Максвелл. Ядро Linux в комментариях. К. Издательство «ДиаСофт», 2000.
15. A.D. Marshall. Programming in C UNIX System Calls and Subroutines using C. 1999. (Электронный источник – http://www.cs.cf.ac.uk/Dave/C/CE.html)

Дополнительная литература

16. Дмитрий Иртегов. Введение в операционные системы. Учебное пособие. СПб: БХВ-Петербург, 2002.
17. А. В. Фролов, Г. В. Фролов. "Операционная система Windows 3.1 для программиста", Библиотека системного программиста, т.т. 11-14, 17. М: "Диалог-МИФИ", 1994-95.
18. В. Л. Григорьев. 80486. Архитектура и программирование (в 4-х книгах). М: "МИКАП", 1993.

19. Б. В. Керниган, Р. Пайк. UNIX - универсальная среда программирования. М: "Финансы и статистика", 1992.
20. З. В. Алферова, Г. Н. Лихачева, В. В. Шураков. Математическое обеспечение ЭВМ. М: "Статистика", 1974.
21. UNIX. Руководство системного администратора. Немет Э. и др. Киев: BHV, 1997.
22. Б. Страуструп. Язык программирования C++. М: Бином, 2002.
23. А.Я. Архангельский. Интегрированная среда разработки C++Builder 5. М: Бином, 2000.
24. А.Я. Архангельский. Разработка прикладных программ для Windows в C++Builder 5. М: Бином, 2000.
25. Разработка приложений Microsoft на Visual C++ 6.0. Учебный курс. М: "Русская Редакция", 2001.
26. К. Грегори «Использование Visual C++ 6. Специальное издание»: Пер. с англ. – М.; СПб.; К.: Издательский дом «Вильямс», 2002.
27. Cross-Referencing Linux (Электронный источник – http://lxr.linux.no)
28. The Linux Kernel Archives (www.kernel.org)
29. Материалы сайтов www.linux.org, www.linuxdoc.ru, www.opennet.ru, www.citforum.ru, www.sysinternals.com и многих других.