**РОСЖЕЛДОР**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Ростовский государственный университет путей сообщения»**

**(ФГБОУ ВО РГУПС)**

Н.А. Москат

**ОПЕРАЦИОННЫЕ СИСТЕМЫ**

**ОПЕРАЦИОННЫЕ СИСТЕМЫ и кроссплатформенное програмирование**

Учебное пособие

Ростов-на-Дону

2017

УДК 681.5.012(07) + 06

Рецензенты: кандидат технических наук, доцент

И.Н. Герасимов (СКФ МТУСИ);

кандидат технических наук, профессор

М.Д. Линденбаум (РГУПС)

**Москат Н.А.**

Операционные системы. Операционные системы и кроссплатформенное программирование: [Электронный ресурс] учебное пособие / Н.А. Москат; ФГБОУ ВО РГУПС. – Ростов н/Д, 2017. – 147 с.

Учебное пособие разработано на основе лекционного курса Карпова В.Е. «Основы операционных систем». Учебное пособие содержит полный спектр материала, необходимого для изучения студентами.

Предназначено для студентов направлений 09.03.01 «Информатика и вычислительная техника» и 09.03.02 «Информационные системы и технологии», изучающих дисциплины «Операционные системы», «Операционные системы и кроссплатформенное программирование», «Операционные системы реального времени» «Операционная система LINUX», «Инструментальные средства информационных систем (Операционные системы)», а так же для студентов всех специальностей, изучающих указанные и смежные дисциплины.

Одобрено к внесению в «Электронный университет» кафедрой «Вы­числительная техника и автоматизированные системы управления».

|  |  |
| --- | --- |
|  | © Москат Н.А., 2017 |
|  | © «Электронный университет» ФГБОУ ВОРГУПС 2017 |

Оглавление

[1. Введение в операционные системы. Основные концептуальные понятия Операционных систем. 4](#_Toc487217080)

[1.1 Назначение и функции операционных систем 4](#_Toc487217081)

[1.2 Архитектура вычислительной системы. Особенности ОС. 13](#_Toc487217082)

[1.3 Классификация ОС 17](#_Toc487217083)

[1.4 Основные концептуальные понятия ОС. 19](#_Toc487217084)

[2. Управление процессами. 22](#_Toc487217085)

[2.1 Состояние и траектория процесса. Иерархия процессов 22](#_Toc487217086)

[2.2 Описание процесса 26](#_Toc487217087)

[2.3 Планирование процессов 32](#_Toc487217088)

[2.4 Средства коммуникации процессов 51](#_Toc487217089)

[2.5 Синхронизация траекторий процессов 61](#_Toc487217090)

[3. Управление памятью 82](#_Toc487217091)

[3.1 Организация памяти компьютера 82](#_Toc487217092)

[3.2 Виртуальная память. Средства поддержки виртуальной памяти. 94](#_Toc487217093)

[3.3 Стратегии управления страничной памятью 102](#_Toc487217094)

[4. Файловая система. 110](#_Toc487217095)

[4.1 Организация файлов 110](#_Toc487217096)

[4.2 Операции над файлами 114](#_Toc487217097)

[5. Управление устройствами 121](#_Toc487217098)

[Библиографический список 143](#_Toc487217099)

# Введение в операционные системы. Основные концептуальные понятия Операционных систем.

**Операционная система**, сокр. ОС ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) operating system, OS) – комплекс взаимосвязанных программ, предназначенных для управления ресурсами [компьютера](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80) и организации взаимодействия с пользователем[1].

## Назначение и функции операционных систем

Из чего состоит любая вычислительная система? Во-первых, из того, что в англоязычных странах принято называть словом hardware, или техническое обеспечение: процессор, память, монитор, дисковые устройства и т.д., объединенные магистральным соединением, которое называется шиной [2].

Во-вторых, вычислительная система состоит из программного обеспечения. Все программное обеспечение принято делить на две части: прикладное и системное. К прикладному программному обеспечению, как правило, относятся разнообразные банковские и прочие бизнес-программы, игры, текстовые процессоры и т. п. Под системным программным обеспечением обычно понимают программы, способствующие функционированию и разработке прикладных программ. Надо сказать, что деление на прикладное и системное программное обеспечение является отчасти условным и зависит от того, кто осуществляет такое деление. Так, обычный пользователь, неискушенный в программировании, может считать MicrosoftWord системной программой, а, с точки зрения программиста, это – приложение. Компилятор языка Си для обычного программиста – системная программа, а для системного – прикладная. Несмотря на эту нечеткую грань, данную ситуацию можно отобразить в виде последовательности слоев (рис. 1.1), выделив отдельно наиболее общую часть системного программного обеспечения – операционную систему:



Рисунок 1.1 – Слои программного обеспечения компьютерной системы

Место операционной системы в структуре компьютера может быть обозначено как представлено на рис.1.2

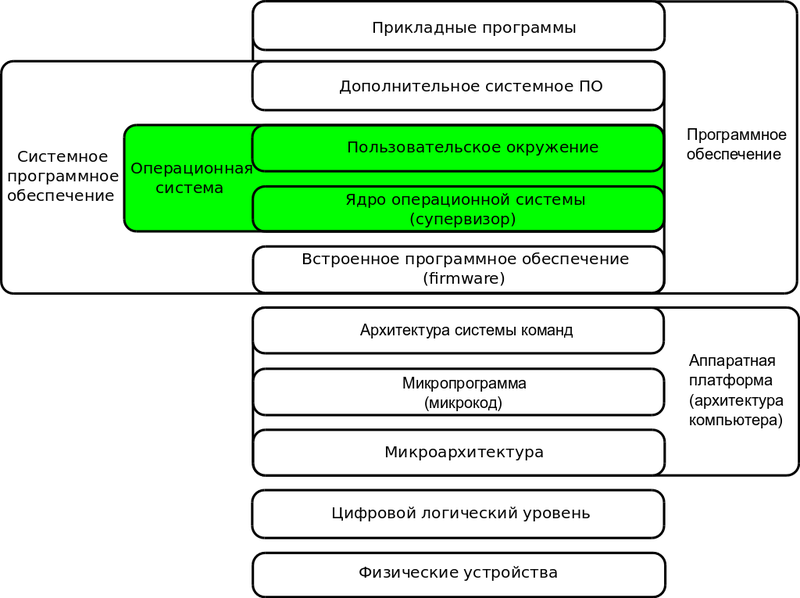


Рисунок 1.2 – Схема, иллюстрирующая место операционной системы в многоуровневой структуре компьютера

Для чего же предназначена операционная система? Каковы ее функции?

**Операционная система как виртуальная машина****.** При разработке ОС широко применяется абстрагирование, которое является важным методом упрощения и позволяет сконцентрироваться на взаимодействии высокоуровневых компонентов системы, игнорируя детали их реализации. В этом смысле ОС представляет собой интерфейс между пользователем и компьютером.Архитектура большинства компьютеров на уровне машинных команд очень неудобна для использования прикладными программами. Например, работа с диском предполагает знание внутреннего устройства его электронного компонента – контроллера для ввода команд вращения диска, поиска и форматирования дорожек, чтения и записи секторов и т. д. Ясно, что средний программист не в состоянии учитывать все особенности работы оборудования (в современной терминологии – заниматься разработкой драйверов устройств), а должен иметь простую высокоуровневую абстракцию, скажем, представляя информационное пространство диска как набор файлов. Файл можно открывать для чтения или записи, использовать для получения или сброса информации, а потом закрывать. Это концептуально проще, чем заботиться о деталях перемещения головок дисков или организации работы мотора. Аналогичным образом, с помощью простых и ясных абстракций, скрываются от программиста все ненужные подробности организации прерываний, работы таймера, управления памятью и т. д. Более того, на современных вычислительных комплексах можно создать иллюзию неограниченного размера оперативной памяти и числа процессоров. Всем этим занимается операционная система. Таким образом, операционная система представляется пользователю виртуальной машиной, с которой проще иметь дело, чем непосредственно с оборудованием компьютера.

**Операционная система как менеджер ресурсов****.** Операционная система предназначена для управления всеми частями весьма сложной архитектуры компьютера. Представим, к примеру, что произойдет, если несколько программ, работающих на одном компьютере, будут пытаться одновременно осуществлять вывод на принтер. Мы получили бы смесь строчек и страниц, выведенных различными программами. Операционная система предотвращает такого рода хаос за счет буферизации информации, предназначенной для печати, на диске и организации очереди на печать. Для многопользовательских компьютеров необходимость управления ресурсами и их защиты еще более очевидна. Следовательно, операционная система, как менеджер ресурсов, осуществляет упорядоченное и контролируемое распределение процессоров, памяти и других ресурсов между различными программами.

**Операционная система как защитник пользователей и программ.** Если вычислительная система допускает совместную работу нескольких пользователей, то возникает проблема организации их безопасной деятельности. Необходимо обеспечить сохранность информации на диске, чтобы никто не мог удалить или повредить чужие файлы. Нельзя разрешить программам одних пользователей произвольно вмешиваться в работу программ других пользователей. Нужно пресекать попытки несанкционированного использования вычислительной системы. Всю эту деятельность осуществляет операционная система как организатор безопасной работы пользователей и их программ. С такой точки зрения операционная система представляется системой безопасности государства, на которую возложены полицейские и контрразведывательные функции.

**Операционная система как постоянно функционирующее ядро****.** Наконец, можно дать и такое определение: операционная система – это программа, постоянно работающая на компьютере и взаимодействующая со всеми прикладными программами. Казалось бы, это абсолютно правильное определение, но, как мы увидим дальше, во многих современных операционных системах постоянно работает на компьютере лишь часть операционной системы, которую принято называть ее ядром.

Как мы видим, существует много точек зрения на то, что такое операционная система. Невозможно дать ей адекватное строгое определение. Проще сказать не что есть операционная система, а для чего она нужна и что она делает.

Далее следовало бы рассказать об **эволюции операционных систем**. Однако эволюция ОС напрямую связано с эволюцией вычислительных систем, с которой студенты наверняка знакомы как из курса школьной информатики, так и из ранее читаемых дисциплин. В рамках данного учебного пособия остановимся только на небольших особенностях эволюции ОС, связанных с развитием вычислительных систем.

Понятно, что hardware и программное обеспечение эволюционировали совместно, оказывая взаимное влияние друг на друга. Появление новых технических возможностей приводило к прорыву в области создания удобных, эффективных и безопасных программ, а свежие идеи в программной области стимулировали поиски новых технических решений. Именно эти критерии – удобство, эффективность и безопасность – играли роль факторов естественного отбора при эволюции вычислительных систем.

Для *первого периода (1945–1955 гг.)* характерна скорее научно-исследовательская работа в области вычислительной техники, а не регулярное использование компьютеров в качестве инструмента Ламповые машины. Операционных систем нет. Вычислительная система выполняла одновременно только одну операцию (ввод-вывод или собственно вычисления). Отладка программ велась с пульта управления с помощью изучения состояния памяти и регистров машины. В конце этого периода появляется первое системное программное обеспечение: в 1951–1952 гг. возникают прообразы первых компиляторов с символических языков (Fortran и др.), а в 1954 г. NatRochester разрабатывает Ассемблер для IBM-701.

Существенная часть времени уходила на подготовку запуска программы, а сами программы выполнялись строго последовательно. Такой режим работы называется последовательной обработкой данных. В целом первый период характеризуется крайне высокой стоимостью вычислительных систем, их малым количеством и низкой эффективностью использования.

*Второй период (1955 г.–начало 60-х).* Компьютеры на основе транзисторов. Пакетные операционные системы.

Теперь машины могут непрерывно работать достаточно долго, чтобы на них можно было возложить выполнение практически важных задач. Снижается потребление вычислительными машинами электроэнергии, совершенствуются системы охлаждения. Размеры компьютеров уменьшились. Снизилась стоимость эксплуатации и обслуживания вычислительной техники. Началось использование ЭВМ коммерческими фирмами. Одновременно наблюдается бурное развитие алгоритмических языков (LISP, COBOL, ALGOL-60, PL-1 и т.д.). Появляются первые настоящие компиляторы, редакторы связей, библиотеки математических и служебных подпрограмм. Упрощается процесс программирования. Пропадает необходимость взваливать на одних и тех же людей весь процесс разработки и использования компьютеров. Именно в этот период происходит разделение персонала на программистов и операторов, специалистов по эксплуатации и разработчиков вычислительных машин.

Изменяется сам процесс прогона программ. Теперь пользователь приносит программу с входными данными в виде колоды перфокарт и указывает необходимые ресурсы. Такая колода получает название задания. Оператор загружает задание в память машины и запускает его на исполнение. Полученные выходные данные печатаются на принтере, и пользователь получает их обратно через некоторое (довольно продолжительное) время.

Смена запрошенных ресурсов вызывает приостановку выполнения программ, в результате процессор часто простаивает. Для повышения эффективности использования компьютера задания с похожими ресурсами начинают собирать вместе, создавая пакет заданий.

Появляются первые системы пакетной обработки, которые просто автоматизируют запуск одной программы из пакета за другой и тем самым увеличивают коэффициент загрузки процессора . При реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Системы пакетной обработки стали прообразом современных операционных систем, они были первыми системными программами, предназначенными для управления вычислительным процессом.

*Третий период (начало 60-х – 1980 г.).* Компьютеры на основе интегральных микросхем. Первые многозадачные ОС

Следующий важный период развития вычислительных машин относится к началу 60-х – 1980 г. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам. Вычислительная техника становится более надежной и дешевой. Растет сложность и количество задач, решаемых компьютерами. Повышается производительность процессоров.

Повышению эффективности использования процессорного времени мешает низкая скорость работы механических устройств ввода-вывода (быстрый считыватель перфокарт мог обработать 1200 перфокарт в минуту, принтеры печатали до 600 строк в минуту). Вместо непосредственного чтения пакета заданий с перфокарт в память начинают использовать его предварительную запись, сначала на магнитную ленту, а затем и на диск. Когда в процессе выполнения задания требуется ввод данных, они читаются с диска. Точно так же выходная информация сначала копируется в системный буфер и записывается на ленту или диск, а печатается только после завершения задания. Вначале действительные операции ввода-вывода осуществлялись в режиме off-line, то есть с использованием других, более простых, отдельно стоящих компьютеров. В дальнейшем они начинают выполняться на том же компьютере, который производит вычисления, то есть в режиме on-line. Такой прием получает название spooling (сокращение от SimultaneousPeripheralOperationOnLine) или подкачки-откачки данных. Введение техники подкачки-откачки в пакетные системы позволило совместить реальные операции ввода-вывода одного задания с выполнением другого задания, но потребовало разработки аппарата прерываний для извещения процессора об окончании этих операций.

Магнитные ленты были устройствами последовательного доступа, то есть информация считывалась с них в том порядке, в каком была записана. Появление магнитного диска, для которого не важен порядок чтения информации, то есть устройства прямого доступа, привело к дальнейшему развитию вычислительных систем. При обработке пакета заданий на магнитной ленте очередность запуска заданий определялась порядком их ввода. При обработке пакета заданий на магнитном диске появилась возможность выбора очередного выполняемого задания. Пакетные системы начинают заниматься планированием заданий: в зависимости от наличия запрошенных ресурсов, срочности вычислений и т.д. на счет выбирается то или иное задание.

Дальнейшее повышение эффективности использования процессора было достигнуто с помощью мультипрограммирования. Идея мультипрограммирования заключается в следующем: пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при однопрограммном режиме, а выполняет другую программу. Когда операция ввода-вывода заканчивается, процессор возвращается к выполнению первой программы. Эта идея напоминает поведение преподавателя и студентов на экзамене. Пока один студент (программа) обдумывает ответ на вопрос (операция ввода-вывода), преподаватель ( процессор ) выслушивает ответ другого студента (вычисления). Естественно, такая ситуация требует наличия в комнате нескольких студентов. Точно так же мультипрограммирование требует наличия в памяти нескольких программ одновременно. При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом, и не должна влиять на выполнение другой программы. (Студенты сидят за отдельными столами и не подсказывают друг другу.)

Появление мультипрограммирования требует настоящей революции в строении вычислительной системы. Особую роль здесь играет аппаратная поддержка (многие аппаратные новшества появились еще на предыдущем этапе эволюции), наиболее существенные особенности которой перечислены ниже.

* Реализация защитных механизмов. Программы не должны иметь самостоятельного доступа к распределению ресурсов, что приводит к появлению привилегированных и непривилегированных команд. Привилегированные команды, например команды ввода-вывода, могут исполняться только операционной системой. Говорят, что она работает в привилегированном режиме. Переход управления от прикладной программы к ОС сопровождается контролируемой сменой режима. Кроме того, это защита памяти, позволяющая изолировать конкурирующие пользовательские программы друг от друга, а ОС – от программ пользователей.
* Наличие прерываний. Внешние прерывания оповещают ОС о том, что произошло асинхронное событие, например завершилась операция ввода-вывода. Внутренние прерывания (сейчас их принято называть исключительными ситуациями) возникают, когда выполнение программы привело к ситуации, требующей вмешательства ОС, например деление на ноль или попытка нарушения защиты.
* Развитие параллелизма в архитектуре. Прямой доступ к памяти и организация каналов ввода-вывода позволили освободить центральный процессор от рутинных операций.

Не менее важна в организации мультипрограммирования роль операционной системы. Она отвечает за следующие операции.

* Организация интерфейса между прикладной программой и ОС при помощи системных вызовов.
* Организация очереди из заданий в памяти и выделение процессора одному из заданий потребовало планирования использования процессора.
* Переключение с одного задания на другое требует сохранения содержимого регистров и структур данных, необходимых для выполнения задания, иначе говоря, контекста для обеспечения правильного продолжения вычислений.
* Поскольку память является ограниченным ресурсом, нужны стратегии управления памятью, то есть требуется упорядочить процессы размещения, замещения и выборки информации из памяти.
* Организация хранения информации на внешних носителях в виде файлов и обеспечение доступа к конкретному файлу только определенным категориям пользователей.
* Поскольку программам может потребоваться произвести санкционированный обмен данными, необходимо их обеспечить средствами коммуникации.
* Для корректного обмена данными необходимо разрешать конфликтные ситуации, возникающие при работе с различными ресурсами и предусмотреть координацию программами своих действий, т.е. снабдить систему средствами синхронизации.

Мультипрограммные системы обеспечили возможность более эффективного использования системных ресурсов (например, процессора, памяти, периферийных устройств), но они еще долго оставались пакетными. Пользователь не мог непосредственно взаимодействовать с заданием и должен был предусмотреть с помощью управляющих карт все возможные ситуации. Отладка программ по-прежнему занимала много времени и требовала изучения многостраничных распечаток содержимого памяти и регистров или использования отладочной печати.

Появление электронно-лучевых дисплеев и переосмысление возможностей применения клавиатур поставили на очередь решение этой проблемы. Логическим расширением систем мультипрограммирования стали time-sharing системы, или системы разделения времени. В них процессор переключается между задачами не только на время операций ввода-вывода, но и просто по прошествии определенного времени. Эти переключения происходят так часто, что пользователи могут взаимодействовать со своими программами во время их выполнения, то есть интерактивно. В результате появляется возможность одновременной работы нескольких пользователей на одной компьютерной системе. У каждого пользователя для этого должна быть хотя бы одна программа в памяти. Чтобы уменьшить ограничения на количество работающих пользователей, была внедрена идея неполного нахождения исполняемой программы в оперативной памяти. Основная часть программы находится на диске, и фрагмент, который необходимо в данный момент выполнять, может быть загружен в оперативную память, а ненужный – выкачан обратно на диск. Это реализуется с помощью механизма виртуальной памяти. Основным достоинством такого механизма является создание иллюзии неограниченной оперативной памяти ЭВМ.

В системах разделения времени пользователь получил возможность эффективно производить отладку программы в интерактивном режиме и записывать информацию на диск, не используя перфокарты, а непосредственно с клавиатуры. Появление on-line-файлов привело к необходимости разработки развитых файловых систем.

Параллельно внутренней эволюции вычислительных систем происходила и внешняя их эволюция. До начала этого периода вычислительные комплексы были, как правило, несовместимы. Каждый имел собственную операционную систему, свою систему команд и т. д. В результате программу, успешно работающую на одном типе машин, необходимо было полностью переписывать и заново отлаживать для выполнения на компьютерах другого типа. В начале третьего периода появилась идея создания семейств программно совместимых машин, работающих под управлением одной и той же операционной системы. Первым семейством программно совместимых компьютеров, построенных на интегральных микросхемах, стала серия машин IBM/360. Разработанное в начале 60-х годов, это семейство значительно превосходило машины второго поколения по критерию цена/производительность. За ним последовала линия компьютеров PDP, несовместимых с линией IBM, и лучшей моделью в ней стала PDP-11.

Сила «одной семьи» была одновременно и ее слабостью. Широкие возможности этой концепции (наличие всех моделей: от мини-компьютеров до гигантских машин; обилие разнообразной периферии; различное окружение; различные пользователи) порождали сложную и громоздкую операционную систему. Миллионы строчек Ассемблера, написанные тысячами программистов, содержали множество ошибок, что вызывало непрерывный поток публикаций о них и попыток исправления. Только в операционной системе OS/360 содержалось более 1000 известных ошибок. Тем не менее идея стандартизации операционных систем была широко внедрена в сознание пользователей и в дальнейшем получила активное развитие.

*Четвертый период (с 1980 г. по настоящее время).* Персональные компьютеры. Классические, сетевые и распределенные системы

Следующий период в эволюции вычислительных систем связан с появлением больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и снижение стоимости микросхем. Компьютер, не отличающийся по архитектуре от PDP-11, по цене и простоте эксплуатации стал доступен отдельному человеку, а не отделу предприятия или университета. Наступила эра персональных компьютеров. Первоначально персональные компьютеры предназначались для использования одним пользователем в однопрограммном режиме, что повлекло за собой деградацию архитектуры этих ЭВМ и их операционных систем (в частности, пропала необходимость защиты файлов и памяти, планирования заданий и т. п.).

Компьютеры стали использоваться не только специалистами, что потребовало разработки «дружественного» программного обеспечения.

Однако рост сложности и разнообразия задач, решаемых на персональных компьютерах, необходимость повышения надежности их работы привели к возрождению практически всех черт, характерных для архитектуры больших вычислительных систем.

В середине 80-х стали бурно развиваться сети компьютеров, в том числе персональных, работающих под управлением сетевых или распределенных операционных систем.

В сетевых операционных системах пользователи могут получить доступ к ресурсам другого сетевого компьютера, только они должны знать об их наличии и уметь это сделать. Каждая машина в сети работает под управлением своей локальной операционной системы, отличающейся от операционной системы автономного компьютера наличием дополнительных средств (программной поддержкой для сетевых интерфейсных устройств и доступа к удаленным ресурсам), но эти дополнения не меняют структуру операционной системы.

Распределенная система, напротив, внешне выглядит как обычная автономная система. Пользователь не знает и не должен знать, где его файлы хранятся – на локальной или удаленной машине – и где его программы выполняются. Он может вообще не знать, подключен ли его компьютер к сети. Внутреннее строение распределенной операционной системы имеет существенные отличия от автономных систем.

В дальнейшем автономные операционные системы мы будем называть классическими операционными системами.

Выделяют также *Пятое поколение (с 1990 г. По наши дни).* Мобильные компьютеры. Особенности операционных систем данного поколения связаны с организацией «мобильности», поддержки широкого ряда графических приложений, быстрого взаимодействия с сетями передачи данных, поддержки целого ряда мобильных приложений. Следует отметить, что в настоящее время процесс развития hardware опережает соответствующее развитие программного обеспечения. Это особенно заметно на классе многопроцессорных систем, где задача грамотного распараллеливания задач является наиболее актуальной.

Просмотрев этапы развития вычислительных систем, мы можем выделить шесть основных функций, которые выполняли классические операционные системы в процессе эволюции:

* Планирование заданий и использования процессора.
* Обеспечение программ средствами коммуникации и синхронизации.
* Управление памятью.
* Управление файловой системой.
* Управление вводом-выводом.
* Обеспечение безопасности

Каждая из приведенных функций обычно реализована в виде подсистемы, являющейся структурным компонентом ОС. В каждой операционной системе эти функции, конечно, реализовывались по-своему, в различном объеме. Они не были изначально придуманы как составные части операционных систем, а появились в процессе развития, по мере того как вычислительные системы становились все более удобными, эффективными и безопасными. Эволюция вычислительных систем, созданных человеком, пошла по такому пути, но никто еще не доказал, что это единственно возможный путь их развития. Операционные системы существуют потому, что на данный момент их существование – это разумный способ использования вычислительных систем. Рассмотрение общих принципов и алгоритмов реализации их функций и составляет содержание большей части нашего курса, в котором будут последовательно описаны перечисленные подсистемы.

## Архитектура вычислительной системы. Особенности ОС.

До сих пор мы говорили о взгляде на операционные системы извне, о том, что делают операционные системы. Дальнейший наш курс будет посвящен тому, как они это делают. Но мы пока ничего не сказали о том, что они представляют собой изнутри, какие подходы существуют к их построению.

**Монолитное ядро**. По сути дела, операционная система – это обычная программа, поэтому было бы логично и организовать ее так же, как устроено большинство программ, то есть составить из процедур и функций. В этом случае компоненты операционной системы являются не самостоятельными модулями, а составными частями одной большой программы. Такая структура операционной системы называется монолитным ядром (monolithickernel). Монолитное ядро представляет собой набор процедур, каждая из которых может вызвать каждую. Все процедуры работают в привилегированном режиме. Таким образом, монолитное ядро – это такая схема операционной системы, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур. Для монолитной операционной системы ядро совпадает со всей системой.

Во многих операционных системах с монолитным ядром сборка ядра, то есть его компиляция, осуществляется отдельно для каждого компьютера, на который устанавливается операционная система. При этом можно выбрать список оборудования и программных протоколов, поддержка которых будет включена в ядро. Так как ядро является единой программой, перекомпиляция – это единственный способ добавить в него новые компоненты или исключить неиспользуемые. Следует отметить, что присутствие в ядре лишних компонентов крайне нежелательно, так как ядро всегда полностью располагается в оперативной памяти. Кроме того, исключение ненужных компонентов повышает надежность операционной системы в целом.

Монолитное ядро – старейший способ организации операционных систем. Примером систем с монолитным ядром является большинство Unix-систем.

Даже в монолитных системах можно выделить некоторую структуру. Как в бетонной глыбе можно различить вкрапления щебенки, так и в монолитном ядре выделяются вкрапления сервисных процедур, соответствующих системным вызовам. Сервисные процедуры выполняются в привилегированном режиме, тогда как пользовательские программы – в непривилегированном. Для перехода с одного уровня привилегий на другой иногда может использоваться главная сервисная программа, определяющая, какой именно системный вызов был сделан, корректность входных данных для этого вызова и передающая управление соответствующей сервисной процедуре с переходом в привилегированный режим работы. Иногда выделяют также набор программных утилит, которые помогают выполнять сервисные процедуры.

**Многоуровневые системы (Layeredsystems).** Продолжая структуризацию, можно разбить всю вычислительную систему на ряд более мелких уровней с хорошо определенными связями между ними, так чтобы объекты уровня N могли вызывать только объекты уровня N-1. Нижним уровнем в таких системах обычно является hardware, верхним уровнем – интерфейс пользователя. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне. Впервые такой подход был применен при создании системы THE (TechnisheHogeschoolEindhoven) Дейкстрой (Dijkstra) и его студентами в 1968 г. Эта система имела следующие уровни , представленные на рис.1.3:

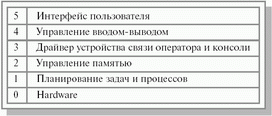


Рисунок 1.3 – Слоеная система THE

Слоеные системы хорошо реализуются. При использовании операций нижнего слоя не нужно знать, как они реализованы, нужно лишь понимать, что они делают. Слоеные системы хорошо тестируются. Отладка начинается с нижнего слоя и проводится послойно. При возникновении ошибки мы можем быть уверены, что она находится в тестируемом слое. Слоеные системы хорошо модифицируются. При необходимости можно заменить лишь один слой, не трогая остальные. Но слоеные системы сложны для разработки: тяжело правильно определить порядок слоев и что к какому слою относится. Слоеные системы менее эффективны, чем монолитные. Так, например, для выполнения операций ввода-вывода программе пользователя придется последовательно проходить все слои от верхнего до нижнего.

**Виртуальные машины.** В начале лекции мы говорили о взгляде на операционную систему как на виртуальную машину, когда пользователю нет необходимости знать детали внутреннего устройства компьютера. Он работает с файлами, а не с магнитными головками и двигателем; он работает с огромной виртуальной, а не ограниченной реальной оперативной памятью; его мало волнует, единственный он на машине пользователь или нет. Рассмотрим несколько иной подход. Пусть операционная система реализует виртуальную машину для каждого пользователя, но не упрощая ему жизнь, а, наоборот, усложняя. Каждая такая виртуальная машина предстает перед пользователем как голое железо – копия всего hardware в вычислительной системе, включая процессор, привилегированные и непривилегированные команды, устройства ввода-вывода, прерывания и т.д. И он остается с этим железом один на один. При попытке обратиться к такому виртуальному железу на уровне привилегированных команд в действительности происходит системный вызов реальной операционной системы, которая и производит все необходимые действия. Такой подход позволяет каждому пользователю загрузить свою операционную систему на виртуальную машину и делать с ней все, что душа пожелает.

Первой реальной системой такого рода была система CP/CMS, или VM/370, как ее называют сейчас, для семейства машин IBM/370.

Недостатком таких операционных систем является снижение эффективности виртуальных машин по сравнению с реальным компьютером. Преимущество же заключается в использовании на одной вычислительной системе программ, написанных для разных операционных систем.

**Микроядерная архитектура**. Современная тенденция в разработке операционных систем состоит в перенесении значительной части системного кода на уровень пользователя и одновременной минимизации ядра. Речь идет о подходе к построению ядра, называемом микроядерной архитектурой (microkernelarchitecture) операционной системы, когда большинство ее составляющих являются самостоятельными программами. В этом случае взаимодействие между ними обеспечивает специальный модуль ядра, называемый микроядром. Микроядро работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью.

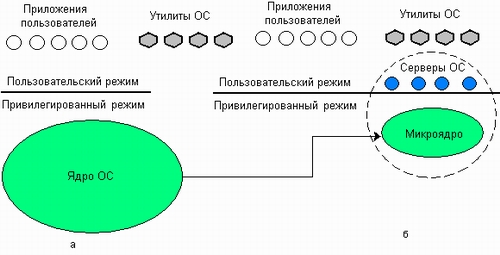


Рисунок 1.4 – Микроядерная архитектура операционной системы

Остальные компоненты системы взаимодействуют друг с другом путем передачи сообщений через микроядро.

Основное достоинство микроядерной архитектуры – высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонентов. В микроядернойоперационной системе можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т. д. Существенно упрощается процесс отладки компонентов ядра, так как новая версия драйвера может загружаться без перезапуска всей операционной системы. Компоненты ядра операционной системы ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства. Микроядерная архитектура повышает надежность системы, поскольку ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра.

В то же время микроядерная архитектураоперационной системы вносит дополнительные накладные расходы, связанные с передачей сообщений, что существенно влияет на производительность. Для того чтобы микроядернаяоперационная система по скорости не уступала операционным системам на базе монолитного ядра, требуется очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними. Таким образом, основная сложность при создании микроядерныхоперационных систем – необходимость очень аккуратного проектирования.

**Смешанные системы.** Все рассмотренные подходы к построению операционных систем имеют свои достоинства и недостатки. В большинстве случаев современные операционные системы используют различные комбинации этих подходов. Так, например, ядро операционной системыLinux представляет собой монолитную систему с элементами микроядерной архитектуры. При компиляции ядра можно разрешить динамическую загрузку и выгрузку очень многих компонентов ядра – так называемых модулей. В момент загрузки модуля его код загружается на уровне системы и связывается с остальной частью ядра. Внутри модуля могут использоваться любые экспортируемые ядром функции. Другим примером смешанного подхода может служить возможность запуска операционной системы с монолитным ядром под управлением микроядра. Так устроены 4.4BSD и MkLinux, основанные на микроядре Mach. Микроядро обеспечивает управление виртуальной памятью и работу низкоуровневых драйверов. Все остальные функции, в том числе взаимодействие с прикладными программами, осуществляется монолитным ядром. Данный подход сформировался в результате попыток использовать преимущества микроядерной архитектуры, сохраняя по возможности хорошо отлаженный код монолитного ядра.

Наиболее тесно элементы микроядерной архитектуры и элементы монолитного ядра переплетены в ядре Windows NT. Хотя Windows NT часто называют микроядернойоперационной системой, это не совсем так. Микроядро NT слишком велико (более 1 Мбайт), чтобы носить приставку "микро". Компоненты ядра Windows NT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений, как и положено в микроядерныхоперационных системах. В то же время все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром. По мнению специалистов Microsoft, причина проста: чисто микроядерный дизайн коммерчески невыгоден, поскольку неэффективен. Таким образом, Windows NT можно с полным правом назвать гибридной операционной системой.

## Классификация ОС

Существует несколько схем классификации операционных систем. Ниже приведена классификация по некоторым признакам с точки зрения пользователя.

**Реализация многозадачности.** По числу одновременно выполняемых задач операционные системы можно разделить на два класса:

* многозадачные (Unix, OS/2, Windows);
* однозадачные (например, MS-DOS).

Многозадачная ОС, решая проблемы распределения ресурсов и конкуренции, полностью реализует мультипрограммный режим в соответствии с требованиями раздела "Основные понятия, концепции ОС ".

Многозадачный режим, который воплощает в себе идею разделения времени, называется вытесняющим (preemptive). Каждой программе выделяется квант процессорного времени, по истечении которого управление передается другой программе. Говорят, что первая программа будет вытеснена. В вытесняющем режиме работают пользовательские программы большинства коммерческих ОС.

В некоторых ОС (Windows 3.11, например) пользовательская программа может монополизировать процессор, то есть работать в невытесняющем режиме. Как правило, в большинстве систем не подлежит вытеснению код собственно ОС. Ответственные программы, в частности задачи реального времени, также не вытесняются. Более подробно об этом рассказано в лекции, посвященной планированию работы процессора.

По приведенным примерам можно судить о приблизительности классификации. Так, в ОС MS-DOS можно организовать запуск дочерней задачи и наличие в памяти двух и более задач одновременно. Однако эта ОС традиционно считается однозадачной, главным образом из-за отсутствия защитных механизмов и коммуникационных возможностей.

**Поддержка многопользовательского режима**

По числу одновременно работающих пользователей ОС можно разделить на:

* однопользовательские (MS-DOS, Windows 3.x);
* многопользовательские (Windows NT, Unix).

Наиболее существенное отличие между этими ОС заключается в наличии у многопользовательских систем механизмов защиты персональных данных каждого пользователя.

**Многопроцессорная обработка.** Вплоть до недавнего времени вычислительные системы имели один центральный процессор. В результате требований к повышению производительности появились многопроцессорные системы, состоящие из двух и более процессоров общего назначения, осуществляющих параллельное выполнение команд. Поддержка мультипроцессирования является важным свойством ОС и приводит к усложнению всех алгоритмов управления ресурсами. Многопроцессорная обработка реализована в таких ОС, как Linux, Solaris, Windows NT, и ряде других.

Многопроцессорные ОС разделяют на симметричные и асимметричные. В симметричных ОС на каждом процессоре функционирует одно и то же ядро, и задача может быть выполнена на любом процессоре, то есть обработка полностью децентрализована. При этом каждому из процессоров доступна вся память.

В асимметричных ОС процессоры неравноправны. Обычно существует главный процессор (master) и подчиненные (slave), загрузку и характер работы которых определяет главный процессор.

**Системы реального времени.** В разряд многозадачных ОС, наряду с пакетными системами и системами разделения времени, включаются также системы реального времени, не упоминавшиеся до сих пор.

Они используются для управления различными техническими объектами или технологическими процессами. Такие системы характеризуются предельно допустимым временем реакции на внешнее событие, в течение которого должна быть выполнена программа, управляющая объектом. Система должна обрабатывать поступающие данные быстрее, чем они могут поступать, причем от нескольких источников одновременно.

Столь жесткие ограничения сказываются на архитектуре систем реального времени, например, в них может отсутствовать виртуальная память, поддержка которой дает непредсказуемые задержки в выполнении программ. (См. также разделы, связанные с планированием процессов и реализацией виртуальной памяти.)

Приведенная классификация ОС не является исчерпывающей. Более подробно особенности применения современных ОС рассмотрены в [3].

## Основные концептуальные понятия ОС.

В процессе эволюции возникло несколько важных концепций, которые стали неотъемлемой частью теории и практики ОС. Рассматриваемые в данном разделе понятия будут встречаться и разъясняться на протяжении всего курса. Здесь дается их краткое описание.

В любой операционной системе поддерживается механизм, который позволяет пользовательским программам обращаться к услугам ядра ОС. В операционных системах наиболее известной советской вычислительной машины БЭСМ-6 соответствующие средства "общения" с ядром назывались экстракодами, в операционных системах IBM они назывались системными макрокомандами и т.д. В ОСUnix такие средства называют системными вызовами.

***Системные вызовы (******systemcalls)*** – это интерфейс между операционной системой и пользовательской программой. Они создают, удаляют и используют различные объекты, главные из которых – процессы и файлы. Пользовательская программа запрашивает сервис у операционной системы, осуществляя системный вызов. Имеются библиотеки процедур, которые загружают машинные регистры определенными параметрами и осуществляют прерывание   процессора, после чего управление передается обработчику данного вызова, входящему в ядро операционной системы. Цель таких библиотек – сделать системный вызов похожим на обычный вызов подпрограммы.

Основное отличие состоит в том, что при системном вызове задача переходит в привилегированный режим или режим ядра (kernelmode). Поэтому системные вызовы иногда еще называют программными прерываниями, в отличие от аппаратных прерываний, которые чаще называют просто прерываниями.

В этом режиме работает код ядра операционной системы, причем исполняется он в адресном пространстве и в контексте вызвавшей его задачи. Таким образом, ядро операционной системы имеет полный доступ к памяти пользовательской программы, и при системном вызове достаточно передать адреса одной или нескольких областей памяти с параметрами вызова и адреса одной или нескольких областей памяти для результатов вызова.

В большинстве операционных систем системный вызов осуществляется командой программного прерывания (INT). Программное прерывание – это синхронное событие, которое может быть повторено при выполнении одного и того же программного кода.

***Прерывание (******hardwareinterrupt)*** – это событие, генерируемое внешним (по отношению к процессору) устройством. Посредством аппаратных прерываний аппаратура либо информирует центральный процессор о том, что произошло какое-либо событие, требующее немедленной реакции (например, пользователь нажал клавишу), либо сообщает о завершении асинхронной операции ввода-вывода (например, закончено чтение данных с диска в основную память). Важный тип аппаратных прерываний – прерывания таймера, которые генерируются периодически через фиксированный промежуток времени. Прерывания таймера используются операционной системой при планировании процессов. Каждый тип аппаратных прерываний имеет собственный номер, однозначно определяющий источник прерывания. Аппаратное прерывание – это асинхронное событие, то есть оно возникает вне зависимости от того, какой код исполняется процессором в данный момент. Обработка аппаратного прерывания не должна учитывать, какой процесс является текущим.

***Исключительная ситуация (exception)*** – событие, возникающее в результате попытки выполнения программой команды, которая по каким-то причинам не может быть выполнена до конца. Примерами таких команд могут быть попытки доступа к ресурсу при отсутствии достаточных привилегий или обращения к отсутствующей странице памяти. Исключительные ситуации, как и системные вызовы, являются синхронными событиями, возникающими в контексте текущей задачи. Исключительные ситуации можно разделить на исправимые и неисправимые. К исправимым относятся такие исключительные ситуации, как отсутствие нужной информации в оперативной памяти. После устранения причины исправимой исключительной ситуации программа может выполняться дальше. Возникновение в процессе работы операционной системы исправимых исключительных ситуаций считается нормальным явлением. Неисправимые исключительные ситуации чаще всего возникают в результате ошибок в программах (например, деление на ноль). Обычно в таких случаях операционная система реагирует завершением программы, вызвавшей исключительную ситуацию.

***Файлы*** предназначены для хранения информации на внешних носителях, то есть принято, что информация, записанная, например, на диске, должна находиться внутри файла. Обычно под файлом понимают именованную часть пространства на носителе информации.

Главная задача файловой системы (filesystem) – скрыть особенности ввода-вывода и дать программисту простую абстрактную модель файлов, независимых от устройств. Для чтения, создания, удаления, записи, открытия и закрытия файлов также имеется обширная категория системных вызовов (создание, удаление, открытие, закрытие, чтение и т.д.). Пользователям хорошо знакомы такие связанные с организацией файловой системы понятия, как каталог, текущий каталог, корневой каталог, путь. Для манипулирования этими объектами в операционной системе имеются системные вызовы.

***Процессы, нити***

Концепция процесса в ОС одна из наиболее фундаментальных. Процессы подробно рассмотрены в последующих разделах данного пособия. Там же описаны нити, или легковесные процессы.

# Управление процессами.

## Состояние и траектория процесса. Иерархия процессов

Начиная с данного раздела, мы будем знакомиться с внутренним устройством и механизмами действия операционных систем, разбирая одну за другой их основные функции по отдельности и во взаимосвязи. Фундаментальным понятием для изучения работы операционных систем является понятие процессов как основных динамических объектов, над которыми системы выполняют определенные действия. Данный раздел посвящен описанию таких объектов, их состояний и свойств, их представлению в вычислительных системах, а также операциям, которые могут проводиться над ними.

**Понятие процесса.** В первом разделе пособия, поясняя понятие «операционная система» и описывая способы построения операционных систем, мы часто применяли слова «программа» и «задание». Мы говорили: вычислительная система исполняет одну или несколько программ, операционная система планирует задания, программы могут обмениваться данными и т. д. Мы использовали эти термины в некотором общеупотребительном, житейском смысле, предполагая, что все читатели одинаково представляют себе, что подразумевается под ними в каждом конкретном случае. При этом одни и те же слова обозначали и объекты в статическом состоянии, необрабатывающиеся вычислительной системой (например, совокупность файлов на диске), и объекты в динамическом состоянии, находящиеся в процессе исполнения. Это было возможно, пока мы говорили об общих свойствах операционных систем, не вдаваясь в подробности их внутреннего устройства и поведения, или о работе вычислительных систем первого-второго поколений, которые не могли обрабатывать более одной программы или одного задания одновременно, по сути дела не имея операционных систем. Но теперь мы начинаем знакомиться с деталями функционирования современных компьютерных систем, и нам придется уточнить терминологию.

Рассмотрим следующий пример. Два студента запускают программу извлечения квадратного корня. Один хочет вычислить квадратный корень из 4, а второй – из 1. С точки зрения студентов, запущена одна и та же программа; с точки зрения компьютерной системы, ей приходится заниматься двумя различными вычислительными процессами, так как разные исходные данные приводят к разному набору вычислений. Следовательно, на уровне происходящего внутри вычислительной системы мы не можем использовать термин «программа» в пользовательском смысле слова.

Рассматривая системы пакетной обработки, мы ввели понятие «задание» как совокупность программы, набора команд языка управления заданиями, необходимых для ее выполнения, и входных данных. С точки зрения студентов, они, подставив разные исходные данные, сформировали два различных задания. Может быть, термин «задание» подойдет нам для описания внутреннего функционирования компьютерных систем? Чтобы выяснить это, давайте рассмотрим другой пример. Пусть оба студента пытаются извлечь корень квадратный из 1, то есть пусть они сформировали идентичные задания, но загрузили их в вычислительную систему со сдвигом по времени. В то время как одно из выполняемых заданий приступило к печати полученного значения, и ждет окончания операции ввода-вывода, второе только начинает исполняться. Можно ли говорить об идентичности заданий внутри вычислительной системы в данный момент? Нет, так как состояние процесса их выполнения различно. Следовательно, и слово «задание» в пользовательском смысле не может применяться для описания происходящего в вычислительной системе.

Это происходит потому, что термины «программа» и «задание» предназначены для описания статических, неактивных объектов. Программа же в процессе исполнения является динамическим, активным объектом. По ходу ее работы компьютер обрабатывает различные команды и преобразует значения переменных. Для выполнения программы операционная система должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ввода-вывода или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть зарезервировать определенные ресурсы из общего числа ресурсов всей вычислительной системы. Их количество и конфигурация с течением времени могут изменяться. Для описания таких активных объектов внутри компьютерной системы вместо терминов «программа» и «задание» мы будем использовать новый термин – «процесс».

Иногда для простоты предлагается рассматривать процесс как абстракцию, характеризующую программу во время выполнения. Эта рекомендация не совсем корректна. Понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы. Не существует взаимно-однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами. В некоторых операционных системах для работы определенных программ может организовываться более одного процесса или один и тот же процесс может исполнять последовательно несколько различных программ. Более того, даже в случае обработки только одной программы в рамках одного процесса нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле!), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке внешних прерываний).

**Состояния процесса.** При использовании такой абстракции все, что выполняется в вычислительных системах (не только программы пользователей, но и, возможно, определенные части операционных систем), организовано как набор процессов. Понятно, что реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди.

Как видим, каждый процесс может находиться как минимум в двух состояниях: процесс исполняется и процесс не исполняется. Диаграмма состояний процесса в такой модели изображена на рис. 2.1.



Рисунок 2.1 – Простейшая диаграмма состояний процесса

Процесс, находящийся в состоянии процесс исполняется, через некоторое время может быть завершен операционной системой или приостановлен и снова переведен в состояние процесс не исполняется. Приостановка процесса происходит по двум причинам: для его дальнейшей работы потребовалось какое-либо событие (например, завершение операции ввода-вывода) или истек временной интервал, отведенный операционной системой для работы данного процесса. После этого операционная система по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии процесс не исполняется, и переводит его в состояние процесс исполняется. Новый процесс, появляющийся в системе, первоначально помещается в состояние процесс не исполняется.

Это очень грубая модель, она не учитывает, в частности, то, что процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов. Для того чтобы избежать такой ситуации, разобьем состояние процесс не исполняется на два новых состояния: готовность и ожидание (рис. 2.2).

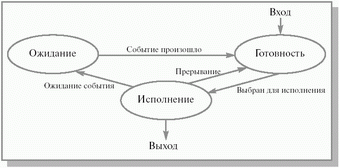


Рисунок 2.2 – Более подробная диаграмма состояний процесса

Всякий новый процесс, появляющийся в системе, попадает в состояние готовность. Операционная система, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние исполнение. В состоянии исполнение происходит непосредственное выполнение программного кода процесса. Выйти из этого состояния процесс может по трем причинам:

* операционная система прекращает его деятельность;
* он не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояниеожидание ;
* в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении предусмотренного времени выполнения) его возвращают в состояниеготовность.

Из состояния ожидание процесс попадает в состояние готовность после того, как ожидаемое событие произошло, и он снова может быть выбран для исполнения.

Наша новая модель хорошо описывает поведение процессов во время их существования, но она не акцентирует внимания на появлении процесса в системе и его исчезновении. Для полноты картины нам необходимо ввести еще два состояния процессов: рождениеи закончил исполнение (рис. 2.3).

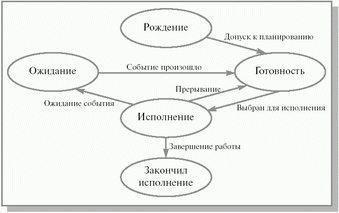


Рисунок 2.3 – Диаграмма(модель) процесса с пятью состояниями. «Классическая».

Теперь для появления в вычислительной системе процесс должен пройти через состояние рождение. При рождении процесс получает в свое распоряжение адресное пространство, в которое загружается программный код процесса ; ему выделяются стек и системные ресурсы; устанавливается начальное значение программного счетчика этого процесса и т. д. Родившийся процесс переводится в состояние готовность. При завершении своей деятельности процесс из состояния исполнение попадает в состояние закончил исполнение.

В конкретных операционных системах состояния процесса могут быть еще более детализированы, могут появиться некоторые новые варианты переходов из одного состояния в другое. Так, например, модель состояний процессов для операционной системы Windows NT содержит 7 различных состояний, а для операционной системы Unix – 9., наиболее значимые состояния для ОС Linux можно представить как изображенные на рис.2.4. Тем не менее, так или иначе все операционные системы подчиняются изложенной выше модели.

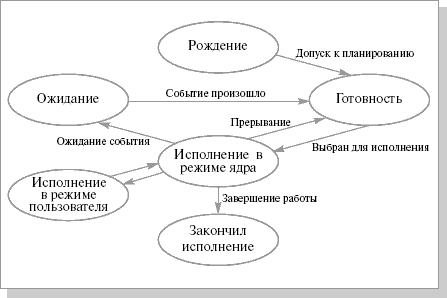


Рисунок 2.4 – Сокращенная диаграмма состояний процесса в LINUX

## Описание процесса

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается операционная система, совершая операции над ними. Количество таких операций в нашей модели пока совпадает с количеством стрелок на диаграмме состояний. Удобно объединить их в три пары:

* создание процесса – завершение процесса ;
* приостановка процесса (перевод из состояния исполнение в состояние готовность) – запуск процесса (перевод из состояния готовность в состояние исполнение );
* блокирование процесса (перевод из состояния исполнение в состояние ожидание) – разблокирование процесса (перевод из состояния ожидание в состояние готовность ).

В дальнейшем, когда мы будем говорить об алгоритмах планирования, в нашей модели появится еще одна операция, не имеющая парной: изменение приоритета процесса.

Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы при работе вычислительной системы не завершаются никогда). Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются многоразовыми. Рассмотрим подробнее, как операционная система выполняет операции над процессами.

**ProcessControlBlock и контекст процесса**. Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных. Эта структура содержит информацию, специфическую для данного процесса:

* состояние, в котором находится процесс ;
* программный счетчик процесса или, другими словами, адрес команды, которая должна быть выполнена для него следующей;
* содержимое регистров процессора;
* данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т. д.);
* учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);
* сведения об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов).

Ее состав и строение зависят, конечно, от конкретной операционной системы. Во многих операционных системах информация, характеризующая процесс, хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации. Для нас это не имеет значения. Для нас важно лишь то, что для любого процесса, находящегося в вычислительной системе, вся информация, необходимая для совершения операций над ним, доступна операционной системе. Для простоты изложения будем считать, что она хранится в одной структуре данных. Мы будем называть ее ***PCB (ProcessControlBlock)*** или ***блоком управления процессом***. Блок управления процессом является моделью процесса для операционной системы. Любая операция, производимая операционной системой над процессом, вызывает определенные изменения в PCB . В рамках принятой модели состояний процессов содержимое PCB между операциями остается постоянным.

Информацию, для хранения которой предназначен блок управления процессом, удобно для дальнейшего изложения разделить на две части. Содержимое всех регистров процессора (включая значение программного счетчика) будем называть ***регистровым контекстом процесса***, а все остальное – ***системным контекстом процесса***. Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его работой в операционной системе, совершая над ним операции. Однако этого недостаточно для того, чтобы полностью охарактеризовать процесс. Операционную систему не интересует, какими именно вычислениями занимается процесс, т. е. какой код и какие данные находятся в его адресном пространстве. С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства процесса, возможно, наряду с регистровым контекстом определяющее последовательность преобразования данных и полученные результаты. Код и данные, находящиеся в адресном пространстве процесса, будем называть его пользовательским контекстом. Совокупность регистрового, системного и пользовательского контекстов процесса для краткости принято называть просто контекстом процесса. В любой момент времени процесс полностью характеризуется своим контекстом.

**Одноразовые операции.** Сложный жизненный путь процесса в компьютере начинается с его рождения. Любая операционная система, поддерживающая концепцию процессов, должна обладать средствами для их создания. В очень простых системах (например, в системах, спроектированных для работы только одного конкретного приложения) все процессы могут быть порождены на этапе старта системы. Более сложные операционные системы создают процессы динамически, по мере необходимости. Инициатором рождения нового процесса после старта операционной системы может выступить либо процесс пользователя, совершивший специальный системный вызов, либо сама операционная система, то есть, в конечном итоге, тоже некоторый процесс. Процесс, инициировавший создание нового процесса, принято называть процессом-родителем (parentprocess), а вновь созданный процесс – процессом-ребенком (childprocess). Процессы-дети могут в свою очередь порождать новых детей и т. д., образуя, в общем случае, внутри системы набор генеалогических деревьев процессов – генеалогический лес. Пример генеалогического леса изображен на рис. 2.4 (стрелочка означает отношение родитель–ребенок). Следует отметить, что все пользовательские процессы вместе с некоторыми процессами операционной системы принадлежат одному и тому же дереву леса. В ряде вычислительных систем лес вообще вырождается в одно такое дерево.

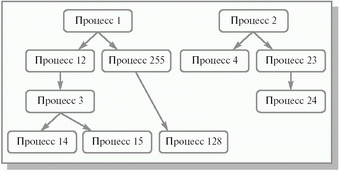


Рисунок 2.5 – Упрощенный генеалогический лес процессов.

При рождении процесса система заводит новый PCB с состоянием процесса рождение и начинает его заполнять. Новый процесс получает собственный уникальный идентификационный номер. Поскольку для хранения идентификационного номера процесса в операционной системе отводится ограниченное количество битов, для соблюдения уникальности номеров количество одновременно присутствующих в ней процессов должно быть ограничено. После завершения какого-либо процесса его освободившийся идентификационный номер может быть повторно использован для другого процесса.

Обычно для выполнения своих функций процесс-ребенок требует определенных ресурсов: памяти, файлов, устройств ввода-вывода и т. д. Существует два подхода к их выделению. Новый процесс может получить в свое распоряжение некоторую часть родительских ресурсов, возможно разделяя с процессом-родителем и другими процессами-детьми права на них, или может получить свои ресурсы непосредственно от операционной системы. Информация о выделенных ресурсах заносится в PCB.

После наделения процесса-ребенка ресурсами необходимо занести в его адресное пространство программный код, значения данных, установить программный счетчик. Здесь также возможны два решения. В первом случае процесс-ребенок становится дубликатом процесса-родителя по регистровому и пользовательскому контекстам, при этом должен существовать способ определения, кто для кого из процессов-двойников является родителем. Во втором случае процесс-ребенок загружается новой программой из какого-либо файла. Операционная система Unix разрешает порождение процесса только первым способом; для запуска новой программы необходимо сначала создать копию процесса-родителя, а затем процесс-ребенок должен заменить свой пользовательский контекст с помощью специального системного вызова. Операционная система VAX/VMS допускает только второе решение. В Windows NT возможны оба варианта (в различных API).

Порождение нового процесса как дубликата процесса-родителя приводит к возможности существования программ (т. е. исполняемых файлов), для работы которых организуется более одного процесса. Возможность замены пользовательского контекста процесса по ходу его работы (т. е. загрузки для исполнения новой программы) приводит к тому, что в рамках одного и того же процесса может последовательно выполняться несколько различных программ.

После того как процесс наделен содержанием, в PCB дописывается оставшаяся информация, и состояние нового процесса изменяется на готовность. Осталось сказать несколько слов о том, как ведут себя процессы-родители после рождения процессов-детей. Процесс-родитель может продолжать свое выполнение одновременно с выполнением процесса-ребенка, а может ожидать завершения работы некоторых или всех своих "детей".

Мы не будем подробно останавливаться на причинах, которые могут привести к завершению жизненного цикла процесса. После того как процесс завершил свою работу, операционная система переводит его в состояние закончил исполнение и освобождает все ассоциированные с ним ресурсы, делая соответствующие записи в блоке управления процессом. При этом сам PCB не уничтожается, а остается в системе еще некоторое время. Это связано с тем, что процесс-родитель после завершения процесса-ребенка может запросить операционную систему о причине "смерти" порожденного им процесса и/или статистическую информацию о его работе. Подобная информация сохраняется в PCB отработавшего процесса до запроса процесса-родителя или до конца его деятельности, после чего все следы завершившегося процесса окончательно исчезают из системы. В операционной системе Unix процессы, находящиеся в состоянии закончил исполнение, принято называть процессами-зомби.

Следует заметить, что в ряде операционных систем (например, в VAX/VMS) гибель процесса-родителя приводит к завершению работы всех его "детей". В других операционных системах (например, в Unix) процессы-дети продолжают свое существование и после окончания работы процесса-родителя. При этом возникает необходимость изменения информации в PCB процессов-детей о породившем их процессе для того, чтобы генеалогический лес процессов оставался целостным. Рассмотрим следующий пример. Пусть процесс с номером 2515 был порожден процессом с номером 2001 и после завершения его работы остается в вычислительной системе неограниченно долго. Тогда не исключено, что номер 2001 будет использован операционной системой повторно для совсем другого процесса. Если не изменить информацию о процессе-родителе для процесса 2515, то генеалогический лес процессов окажется некорректным – процесс 2515 будет считать своим родителем новый процесс 2001, а процесс 2001 будет открещиваться от нежданного потомка. Как правило, "осиротевшие" процессы "усыновляются" одним из системных процессов, который порождается при старте операционной системы и функционирует все время, пока она работает.

**Многоразовые операции.** Одноразовые операции приводят к изменению количества процессов, находящихся под управлением операционной системы, и всегда связаны с выделением или освобождением определенных ресурсов. Многоразовые операции, напротив, не приводят к изменению количества процессов в операционной системе и не обязаны быть связанными с выделением или освобождением ресурсов.

*Запуск процесса*. Из числа процессов, находящихся в состоянии готовность, операционная система выбирает один процесс для последующего исполнения. Критерии и алгоритмы такого выбора будут подробно рассмотрены в лекции 3 – "Планирование процессов ". Для избранного процесса операционная система обеспечивает наличие в оперативной памяти информации, необходимой для его дальнейшего выполнения. То, как она это делает, будет в деталях описано в лекциях 8-10. Далее состояние процесса изменяется на исполнение, восстанавливаются значения регистров для данного процесса и управление передается команде, на которую указывает счетчик команд процесса. Все данные, необходимые для восстановления контекста, извлекаются из PCB процесса, над которым совершается операция.

*Приостановка процесса*. Работа процесса, находящегося в состоянии исполнение, приостанавливается в результате какого-либо прерывания. Процессор автоматически сохраняет счетчик команд и, возможно, один или несколько регистров в стеке исполняемого процесса, а затем передает управление по специальному адресу обработки данного прерывания. На этом деятельность hardware по обработке прерывания завершается. По указанному адресу обычно располагается одна из частей операционной системы. Она сохраняет динамическую часть системного и регистрового контекстов процесса в его PCB, переводит процесс в состояние готовность и приступает к обработке прерывания, то есть к выполнению определенных действий, связанных с возникшим прерыванием.

*Блокирование процесса*. Процесс блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе. Для этого он обращается к операционной системе с помощью определенного системного вызова. Операционная система обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет процесс в очередь процессов, дожидающихся освобождения устройства или возникновения события, и т. д.) и, при необходимости сохранив нужную часть контекста процесса в его PCB, переводит процесс из состояния исполнение в состояние ожидание .

*Разблокирование процесса.* После возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло. Затем операционная система проверяет, находился ли некоторый процесс в состоянии ожидание для данного события, и если находился, переводит его в состояние готовность, выполняя необходимые действия, связанные с наступлением события (инициализация операции ввода-вывода для очередного ожидающего процесса и т. п.). Эта операция, как и операция блокирования, будет подробно описана в лекции 13.

**Переключение контекста.** До сих пор мы рассматривали операции над процессами изолированно, независимо друг от друга. В действительности же деятельность мультипрограммной операционной системы состоит из цепочек операций, выполняемых над различными процессами, и сопровождается переключением процессора с одного процесса на другой.

Давайте для примера упрощенно рассмотрим, как в реальности может протекать операция разблокирования процесса, ожидающего ввода-вывода (рис. 2.6). При исполнении процессором некоторого процесса (на рисунке – процесс 1) возникает прерывание от устройства ввода-вывода, сигнализирующее об окончании операций на устройстве. Над выполняющимся процессом производится операция приостановки. Далее операционная система разблокирует процесс, инициировавший запрос на ввод-вывод (на рисунке – процесс 2) и осуществляет запуск приостановленного или нового процесса, выбранного при выполнении планирования (на рисунке был выбран разблокированный процесс ). Как мы видим, в результате обработки информации об окончании операции ввода-вывода возможна смена процесса, находящегося в состоянии исполнение.

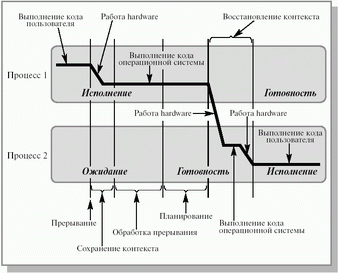


Рисунок 2.6 – Выполнение операции разблокирования процесса.

Для корректного переключения процессора с одного процесса на другой необходимо сохранить контекст исполнявшегося процесса и восстановить контекст процесса, на который будет переключен процессор. Такая процедура сохранения/восстановления работоспособности процессов называется переключением контекста. Время, затраченное на переключение контекста, не используется вычислительной системой для совершения полезной работы и представляет собой накладные расходы, снижающие производительность системы. Оно меняется от машины к машине и обычно колеблется в диапазоне от 1 до 1000 микросекунд. Существенно сократить накладные расходы в современных операционных системах позволяет расширенная модель процессов, включающая в себя понятие threadsofexecution (нити исполнения или просто нити).

Лекция 3:

## Планирование процессов

Всякий раз, когда нам приходится иметь дело с ограниченным количеством ресурсов и несколькими их потребителями, будь то фонд заработной платы в трудовом коллективе или студенческая вечеринка с несколькими ящиками сока, мы вынуждены заниматься распределением наличных ресурсов между потребителями или, другими словами, планированием использования ресурсов. Такое планирование должно иметь четко поставленные цели (чего мы хотим добиться за счет распределения ресурсов) и алгоритмы, соответствующие целям и опирающиеся на параметры потребителей. Только при правильном выборе критериев и алгоритмов можно избежать таких вопросов, как: "Почему я получаю в десять раз меньше, чем мой шеф?" или "А где мой сок?".

**Уровни планирования.** Ранее, рассматривая эволюцию компьютерных систем, мы говорили о двух видах планирования в вычислительных системах: планировании заданий и планировании использования процессора. Планирование заданий появилось в пакетных системах после того, как для хранения сформированных пакетов заданий начали использоваться магнитные диски. Магнитные диски, являясь устройствами прямого доступа, позволяют загружать задания в компьютер в произвольном порядке, а не только в том, в котором они были записаны на диск. Изменяя порядок загрузки заданий в вычислительную систему, можно повысить эффективность ее использования. Процедуру выбора очередного задания для загрузки в машину, т. е. для порождения соответствующего процесса, мы и назвали планированием заданий. Планирование использования процессора впервые возникает в мультипрограммных вычислительных системах, где в состоянии готовность могут одновременно находиться несколько процессов. Именно для процедуры выбора из них одного процесса, который получит процессор в свое распоряжение, т. е. будет переведен в состояние исполнение, мы использовали это словосочетание. Теперь, познакомившись с концепцией процессов в вычислительных системах, оба вида планирования мы будем рассматривать как различные уровни планирования процессов.

Планирование заданий используется в качестве ***долгосрочного планирования*** процессов. Оно отвечает за порождение новых процессов в системе, определяя ее степень мультипрограммирования, т. е. количество процессов, одновременно находящихся в ней. Если степень мультипрограммирования системы поддерживается постоянной, т. е. среднее количество процессов в компьютере не меняется, то новые процессы могут появляться только после завершения ранее загруженных. Поэтому долгосрочное планирование осуществляется достаточно редко, между появлением новых процессов могут проходить минуты и даже десятки минут. Решение о выборе для запуска того или иного процесса оказывает влияние на функционирование вычислительной системы на протяжении достаточно длительного времени. Отсюда и название этого уровня планирования – долгосрочное. В некоторых операционных системах долгосрочное планирование сведено к минимуму или отсутствует вовсе. Так, например, во многих интерактивных системах разделения времени порождение процесса происходит сразу после появления соответствующего запроса. Поддержание разумной степени мультипрограммирования осуществляется за счет ограничения количества пользователей, которые могут работать в системе, и особенностей 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.

Планирование использования процессора применяется в качестве ***краткосрочного планирования процессов***. Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода или просто по завершении определенного интервала времени. Поэтому краткосрочное планирование осуществляется, как правило, не реже одного раза в 100 миллисекунд. Выбор нового процесса для исполнения оказывает влияние на функционирование системы до наступления очередного аналогичного события, т. е. в течение короткого промежутка времени, чем и обусловлено название этого уровня планирования – краткосрочное.

В некоторых вычислительных системах бывает выгодно для повышения производительности временно удалить какой-либо частично выполнившийся процесс из оперативной памяти на диск, а позже вернуть его обратно для дальнейшего выполнения. Такая процедура в англоязычной литературе получила название ***swapping***, что можно перевести на русский язык как «*перекачка*», хотя в специальной литературе оно употребляется без перевода – свопинг. Когда и какой из процессов нужно перекачать на диск и вернуть обратно, решается дополнительным промежуточным уровнем планирования процессов – среднесрочным .

**Критерии планирования и требования к алгоритмам.** Для каждого уровня планирования процессов можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых мы хотим достичь, используя планирование. К числу таких целей можно отнести следующие:

* Справедливость – гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не начинал выполняться.
* Эффективность – постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90%.
* Сокращение полного времени выполнения (turnaroundtime) – обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением.
* Сокращение времени ожидания (waitingtime) – сократить время, которое проводят процессы в состоянии готовность и задания в очереди для загрузки.
* Сокращение времени отклика (responsetime) – минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Независимо от поставленных целей планирования желательно также, чтобы алгоритмы обладали следующими свойствами.

* Были предсказуемыми. Одно и то же задание должно выполняться приблизительно за одно и то же время. Применение алгоритма планирования не должно приводить, к примеру, к извлечению квадратного корня из 4 за сотые доли секунды при одном запуске и за несколько суток – при втором запуске.
* Были связаны с минимальными накладными расходами. Если на каждые 100 миллисекунд, выделенные процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, применять не стоит.
* Равномерно загружали ресурсы вычислительной системы, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы.
* Обладали масштабируемостью, т. е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

Многие из приведенных выше целей и свойств являются противоречивыми. Улучшая работу алгоритма с точки зрения одного критерия, мы ухудшаем ее с точки зрения другого. Приспосабливая алгоритм под один класс задач, мы тем самым дискриминируем задачи другого класса. "В одну телегу впрячь не можно коня и трепетную лань". Ничего не поделаешь. Такова жизнь.

**Параметры планирования.** Для осуществления поставленных целей разумные алгоритмы планирования должны опираться на какие-либо характеристики процессов в системе, заданий в очереди на загрузку, состояния самой вычислительной системы, иными словами, на параметры планирования . В этом разделе мы опишем ряд таких параметров, не претендуя на полноту изложения.

Все параметры планирования можно разбить на две большие группы: статические параметры и динамические параметры. Статические параметры не изменяются в ходе функционирования вычислительной системы, динамические же, напротив, подвержены постоянным изменениям.

К статическим параметрам вычислительной системы можно отнести предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т. п.). Динамические параметры системы описывают количество свободных ресурсов на данный момент.

К статическим параметрам процессов относятся характеристики, как правило присущие заданиям уже на этапе загрузки.

* 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 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.

Алгоритмы долгосрочного планирования используют в своей работе статические и динамические параметры вычислительной системы и статические параметры процессов (динамические параметры процессов на этапе загрузки заданий еще не известны). Алгоритмы краткосрочного и среднесрочного планирования дополнительно учитывают и динамические характеристики процессов. Для среднесрочного планирования в качестве таких характеристик может использоваться следующая информация:

* сколько времени прошло с момента выгрузки процесса на диск или его загрузки в оперативную память;
* сколько оперативной памяти занимает процесс;
* сколько процессорного времени уже предоставлено процессу.

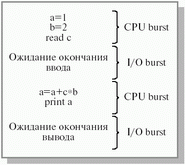


Рисунок 2.7 – Фрагмент деятельности процесса с выделением промежутков

непрерывного использования процессора и ожидания ввода-вывода

Для краткосрочного планирования нам понадобится ввести еще два динамических параметра. Деятельность любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Промежуток времени непрерывного использования процессора носит название CPU burst, а промежуток времени непрерывного ожидания ввода-вывода – I/O burst . На рис. 3.1. показан фрагмент деятельности некоторого процесса на псевдоязыке программирования с выделением указанных промежутков. Для краткости мы будем использовать термины CPU burst и I/O burst без перевода. Значения продолжительности последних и очередных CPU burst и I/O burst являются важными динамическими параметрами процесса.

**Вытесняющее и невытесняющее планирование.** Процесс планирования осуществляется частью операционной системы, называемой планировщиком. Планировщик может принимать решения о выборе для исполнения нового процесса из числа находящихся в состоянии готовность в следующих четырех случаях.

1. Когда процесс переводится из состояния исполнение в состояние закончил исполнение.
2. Когда процесс переводится из состояния исполнение в состояние ожидание.
3. Когда процесс переводится из состояния исполнение в состояние готовность (например, после прерывания от таймера).
4. Когда процесс переводится из состояния ожидание в состояние готовность (завершилась операция ввода-вывода или произошло другое событие).

В случаях 1 и 2 процесс, находившийся в состоянии исполнение, не может дальше исполняться, и операционная система вынуждена осуществлять планирование выбирая новый процесс для выполнения. В случаях 3 и 4 планирование может как проводиться, так и не проводиться, планировщик не вынужден обязательно принимать решение о выборе процесса для выполнения, процесс, находившийся в состоянии исполнение может просто продолжить свою работу. Если в операционной системе планирование осуществляется только в вынужденных ситуациях, говорят, что имеет место ***невытесняющее*** (nonpreemptive) планирование. Если планировщик принимает и вынужденные, и невынужденные решения, говорят о ***вытесняющем*** (preemptive) планировании. Термин «вытесняющее планирование» возник потому, что исполняющийся процесс помимо своей воли может быть вытеснен из состояния исполнение другим процессом.

Невытесняющее планирование используется, например, в MS Windows 3.1 и ОС AppleMacintosh. При таком режиме планирования процесс занимает столько процессорного времени, сколько ему необходимо. При этом переключение процессов возникает только при желании самого исполняющегося процесса передать управление (для ожидания завершения операции ввода-вывода или по окончании работы). Этот метод планирования относительно просто реализуем и достаточно эффективен, так как позволяет выделить большую часть процессорного времени для работы самих процессов и до минимума сократить затраты на переключение контекста. Однако при невытесняющем планировании возникает проблема возможности полного захвата процессора одним процессом, который вследствие каких-либо причин (например, из-за ошибки в программе) зацикливается и не может передать управление другому процессу. В такой ситуации спасает только перезагрузка всей вычислительной системы.

Вытесняющее планирование обычно используется в системах разделения времени. В этом режиме планирования процесс может быть приостановлен в любой момент исполнения. Операционная система устанавливает специальный таймер для генерации сигнала прерывания по истечении некоторого интервала времени – кванта. После прерывания процессор передается в распоряжение следующего процесса. Временные прерывания помогают гарантировать приемлемое время отклика процессов для пользователей, работающих в диалоговом режиме, и предотвращают "зависание" компьютерной системы из-за зацикливания какой-либо программы.

**Алгоритмы планирования**. Существует достаточно большой набор разнообразных алгоритмов планирования, которые предназначены для достижения различных целей и эффективны для разных классов задач. Многие из них могут использоваться на нескольких уровнях планирования. В этом разделе мы рассмотрим некоторые наиболее употребительные алгоритмы применительно к процессу кратковременного планирования.

***First-Come, First-Served (FCFS)***

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой FCFS по первым буквам его английского названия – First-Come, First-Served (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии готовность, выстроены в очередь. Когда процесс переходит в состояние готовность, он, а точнее, ссылка на его PCB помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки наегоPCB. Очередь подобного типа имеет в программировании специальное наименование – FIFO1, сокращение от FirstIn, FirstOut (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего CPU burst. После этого для выполнения выбирается новый процесс из начала очереди.

*Таблица 2.1*

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| **Процесс** | p0 | p1 | p2 |
| **Продолжительность очередногоCPU burst** | 13 | 4 | 1 |

Преимуществом алгоритма FCFS является легкость его реализации, но в то же время он имеет и много недостатков. Рассмотрим следующий пример. Пусть в состоянии готовность находятся три процесса p0, p1 и p2, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 2.1. в некоторых условных единицах. Для простоты будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что время переключения контекста так мало, что им можно пренебречь.

Если процессы расположены в очереди процессов, готовых к исполнению, в порядке p0, p1, p2, то картина их выполнения выглядит так, как показано на рис. 2.8. Первым для выполнения выбирается процесс p0, который получает процессор на все время своего CPU burst, т. е. на 13 единиц времени. После его окончания в состояние исполнение переводится процесс p1, он занимает процессор на 4 единицы времени. И, наконец, возможность работать получает процесс p2. Время ожидания для процесса p0 составляет 0 единиц времени, для процесса p1 – 13 единиц, для процесса p2 – 13 + 4 = 17 единиц. Таким образом, среднее время ожидания в этом случае – (0 + 13 + 17)/3 = 10 единиц времени. Полное время выполнения для процесса p0 составляет 13 единиц времени, для процесса p1 – 13 + 4 = 17 единиц, для процесса p2 – 13 + 4 + 1 = 18 единиц. Среднее полное время выполнения оказывается равным (13 + 17 + 18)/3 = 16 единицам времени.

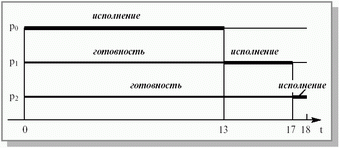


Рисунок 2.8 – Выполнение процессов при порядке p0,p1,p2

Если те же самые процессы расположены в порядке p2, p1, p0, то картина их выполнения будет соответствовать рис. 2.9. Время ожидания для процесса p0 равняется 5 единицам времени, для процесса p1 – 1 единице, для процесса p2 – 0 единиц. Среднее время ожидания составит (5 + 1 + 0)/3 = 2 единицы времени. Это в 5 (!) раз меньше, чем в предыдущем случае. Полное время выполнения для процесса p0 получается равным 18 единицам времени, для процесса p1 – 5 единицам, для процесса p2 – 1 единице. Среднее полное время выполнения составляет (18 + 5 + 1)/3 = 8 единиц времени, что почти в 2 раза меньше, чем при первой расстановке процессов.

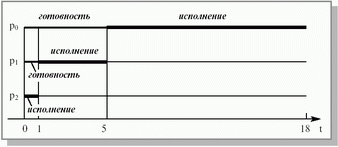


Рисунок 2.9 – Выполнение процессов при порядке p2, p1, p0

Как мы видим, среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. Если у нас есть процесс с длительным CPU burst, то короткие процессы, перешедшие в состояние готовность после длительного процесса, будут очень долго ждать начала выполнения. Поэтому алгоритм FCFS практически неприменим для систем разделения времени – слишком большим получается среднее время отклика в интерактивных процессах.

***RoundRobin (RR)***

Модификацией алгоритма FCFS является алгоритм, получивший название RoundRobin (RoundRobin – это вид детской карусели в США) или сокращенно RR. По сути дела, это тот же самый алгоритм, только реализованный в режиме вытесняющего планирования. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10 – 100 миллисекунд (рис. 2.10.). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

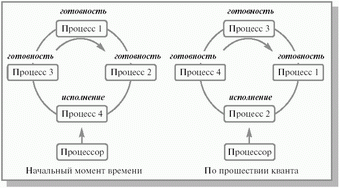


Рисунок 2.10 – Карусельная стратегия планирования

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии готовность, в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени. При выполнении процесса возможны два варианта.

* Время непрерывного использования процессора, необходимое процессу (остаток текущего CPU burst), меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет кванта заново.
* Продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

Рассмотрим предыдущий пример с порядком процессов p0, p1, p2 и величиной кванта времени равной 4. Выполнение этих процессов иллюстрируется таблицей 2.2. Обозначение "И" используется в ней для процесса, находящегося в состоянии исполнение, обозначение "Г" – для процессов в состоянии готовность, пустые ячейки соответствуют завершившимся процессам. Состояния процессов показаны на протяжении соответствующей единицы времени, т. е. колонка с номером 1 соответствует промежутку времени от 0 до 1.

*Таблица 2.2*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| **p0** | И | И | И | И | Г | Г | Г | Г | Г | И | И | И | И | И | И | И | И | И |
| **p1** | Г | Г | Г | Г | И | И | И | И |  |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | Г | Г | Г | Г | Г | Г | И |  |  |  |  |  |  |  |  |  |

Первым для исполнения выбирается процесс p0. Продолжительность его CPU burst больше, чем величина кванта времени, и поэтому процесс исполняется до истечения кванта, т. е. в течение 4 единиц времени. После этого он помещается в конец очереди готовых к исполнению процессов, которая принимает вид p1, p2, p0. Следующим начинает выполняться процесс p1. Время его исполнения совпадает с величиной выделенного кванта, поэтому процесс работает до своего завершения. Теперь очередь процессов в состоянии готовность состоит из двух процессов, p2 и p0. Процессор выделяется процессу p2. Он завершается до истечения отпущенного ему процессорного времени, и очередные кванты отмеряются процессу p0 – единственному не закончившему к этому моменту свою работу. Время ожидания для процесса p0 (количество символов "Г" в соответствующей строке) составляет 5 единиц времени, для процесса p1 – 4 единицы времени, для процесса p2 – 8 единиц времени. Таким образом, среднее время ожидания для этого алгоритма получается равным (5 + 4 + 8)/3 = 5,6(6) единицы времени. Полное время выполнения для процесса p0 (количество непустых столбцов в соответствующей строке) составляет 18 единиц времени, для процесса p1 – 8 единиц, для процесса p2 – 9 единиц. Среднее полное время выполнения оказывается равным (18 + 8 + 9)/3 = 11,6(6) единицы времени.

Легко увидеть, что среднее время ожидания и среднее полное время выполнения для обратного порядка процессов не отличаются от соответствующих времен для алгоритма FCFS и составляют 2 и 8 единиц времени соответственно.

На производительность алгоритма RR сильно влияет величина кванта времени. Рассмотрим тот же самый пример с порядком процессов p0, p1, p2 для величины кванта времени, равной 1 (табл. 2.3.). Время ожидания для процесса p0 составит 5 единиц времени, для процесса p1 – тоже 5 единиц, для процесса p2 – 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3)/3 = 10 единиц времени.

*Таблица 2.3*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| **p0** | И | Г | Г | И | Г | И | Г | И | Г | И | И | И | И | И | И | И | И | И |
| **p1** | Г | И | Г | Г | И | Г | И | Г | И |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

При очень больших величинах кванта времени, когда каждый процесс успевает завершить свой CPU burst до возникновения прерывания по времени, алгоритм RR вырождается в алгоритм FCFS. При очень малых величинах создается иллюзия того, что каждый из n процессов работает на собственном виртуальном процессоре с производительностью ~ 1/n от производительности реального процессора. Правда, это справедливо лишь при теоретическом анализе при условии пренебрежения временами переключения контекста процессов. В реальных условиях при слишком малой величине кванта времени и, соответственно, слишком частом переключении контекста накладные расходы на переключение резко снижают производительность системы.

***Shortest-Job-First (SJF)***

При рассмотрении алгоритмов FCFS и RR мы видели, насколько существенным для них является порядок расположения процессов в очереди процессов, готовых к исполнению. Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает. Если бы мы знали время следующих CPU burst для процессов, находящихся в состоянии готовность, то могли бы выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название "кратчайшая работа первой" или ShortestJobFirst( SJF ).

SJF-алгоритм краткосрочного планирования может быть как вытесняющим, так и невытесняющим. При невытесняющем SJF – планировании процессор предоставляется избранному процессу на все необходимое ему время, независимо от событий, происходящих в вычислительной системе. При вытесняющем SJF – планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burstу исполняющегося, то исполняющийся процесс вытесняется новым.

Рассмотрим пример работы невытесняющего алгоритма SJF. Пусть в состоянии готовность находятся четыре процесса, p0, p1, p2 и p3, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 2.4. Как и прежде, будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что временем переключения контекста можно пренебречь.

*Таблица 2.4*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | | |
| **Процесс** | p0 | p1 | p2 | p3 |
| **Продолжительность очередногоCPU burst** | 5 | 3 | 7 | 1 |

При использовании невытесняющего алгоритма SJF первым для исполнения будет выбран процесс p3, имеющий наименьшее значение продолжительности очередного CPU burst. После его завершения для исполнения выбирается процесс p1, затем p0 и, наконец, p2. Эта картина отражена в таблице 2.5.

*Таблица 2.5*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **p0** | Г | Г | Г | Г | И | И | И | И | И |  |  |  |  |  |  |  |
| **p1** | Г | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И | И |
| **p3** | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Как мы видим, среднее время ожидания для алгоритма SJF составляет (4 + 1 + 9 + 0)/4 = 3,5 единицы времени. Легко посчитать, что для алгоритма FCFS при порядке процессов p0, p1, p2, p3 эта величина будет равняться (0 + 5 + 8 + 15)/4 = 7 единицам времени, т. е. будет в два раза больше, чем для алгоритма SJF. Можно показать, что для заданного набора процессов (если в очереди не появляются новые процессы) алгоритм SJF является оптимальным с точки зрения минимизации среднего времени ожидания среди класса невытесняющих алгоритмов.

Для рассмотрения примера вытесняющего SJFпланирования мы возьмем ряд процессов p0, p1, p2 и p3 с различными временами CPU burst и различными моментами их появления в очереди процессов, готовых к исполнению (табл. 2.6.).

*Таблица 2.6*

|  |  |  |
| --- | --- | --- |
|  | | |
| **Процесс** | **Время появления в очереди очередного CPU burst** | **Продолжительность** |
| **p0** | 0 | 6 |
| **p1** | 2 | 2 |
| **p2** | 6 | 7 |
| **p3** | 0 | 5 |

В начальный момент времени в состоянии готовность находятся только два процесса, p0 и p3. Меньшее время очередного CPU burst оказывается у процесса p3, поэтому он и выбирается для исполнения (табл. 2.7.). По прошествии2 единиц времени в систему поступает процесс p1. Время его CPU burst меньше, чем остаток CPU burst у процесса p3, который вытесняется из состояния исполнение и переводится в состояние готовность. По прошествии еще 2 единиц времени процесс p1 завершается, и для исполнения вновь выбирается процесс p3. В момент времени t = 6 в очереди процессов, готовых к исполнению, появляется процесс p2, но поскольку ему для работы нужно 7 единиц времени, а процессу p3 осталось трудиться всего 1 единицу времени, то процесс p3 остается в состоянии исполнение. После его завершения в момент времени t = 7 в очереди находятся процессы p0 и p2, из которых выбирается процесс p0. Наконец, последним получит возможность выполняться процесс p2.

*Таблица 2.7*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **p0** | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И |  |  |  |  |  |  |  |
| **p1** |  |  | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **p2** |  |  |  |  |  |  | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И | И |
| **p3** | И | И | Г | Г | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |

Основную сложность при реализации алгоритма SJF представляет невозможность точного знания продолжительности очередного CPU burst для исполняющихся процессов. В пакетных системах количество процессорного времени, необходимое заданию для выполнения, указывает пользователь при формировании задания. Мы можем брать эту величину для осуществления долгосрочного SJF – планирования. Если пользователь укажет больше времени, чем ему нужно, он будет ждать результата дольше, чем мог бы, так как задание будет загружено в систему позже. Если же он укажет меньшее количество времени, задача может не досчитаться до конца. Таким образом, в пакетных системах решение задачи оценки времени использования процессора перекладывается на плечи пользователя. При краткосрочном планировании мы можем делать только прогноз длительности следующего CPU burst, исходя из предыстории работы процесса. Пусть – величина *n* -го CPU burst, *T*(*n* + 1) – предсказываемое значение для *n* + 1 -го CPU burst, – некоторая величина в диапазоне от 0 до 1.

Определим рекуррентное соотношение

*T*(0) положим произвольной константой. Первое слагаемое учитывает последнее поведение процесса, тогда как второе слагаемое учитывает его предысторию. При мы перестаем следить за последним поведением процесса, фактически полагая

*T*(*n*)= *T*(*n*+1)=...=*T*(0)

т. е. оценивая все CPU burst одинаково, исходя из некоторого начального предположения.

Положив , мы забываем о предыстории процесса. В этом случае мы полагаем, что время очередного CPU burst будет совпадать со временем последнего CPU burst:

Обычно выбирают для равноценного учета последнего поведения и предыстории. Надо отметить, что такой выбор удобен и для быстрой организации вычисления оценки *T*(*n* + 1). Для подсчета новой оценки нужно взять старую оценку, сложить с измеренным временем CPU burst и полученную сумму разделить на 2, например, сдвинув ее на 1 бит вправо. Полученные оценки *T*(*n* + 1) применяются как продолжительности очередных промежутков времени непрерывного использования процессора для краткосрочного SJF – планирования.

***Гарантированное планирование***

При интерактивной работе *N* пользователей в вычислительной системе можно применить алгоритм планирования, который гарантирует, что каждый из пользователей будет иметь в своем распоряжении ~1/*N* часть процессорного времени. Пронумеруем всех пользователей от 1 до *N*. Для каждого пользователя с номером *i* введем две величины: *Ti* – время нахождения пользователя в системе или, другими словами, длительность сеанса его общения с машиной и – суммарное процессорное время уже выделенное всем его процессам в течение сеанса. Справедливым для пользователя было бы получение *Ti*/*N* процессорного времени. Если

то *i* -й пользователь несправедливо обделен процессорным временем. Если же

то система явно благоволит к пользователю с номером *i*. Вычислим для процессов каждого пользователя значение коэффициента справедливости

и будем предоставлять очередной квант времени готовому 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.

***Приоритетное планирование***

Алгоритмы SJF и гарантированного планирования представляют собой частные случаи приоритетного планирования. При приоритетном планировании каждому процессу присваивается определенное числовое значение – приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJF в качестве такого приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс. Для алгоритма гарантированного планирования приоритетом служит вычисленный коэффициент справедливости. Чем он меньше, тем больше у процесса приоритет .

Алгоритмы назначения приоритетов процессов могут опираться как на внутренние параметры, связанные с происходящим внутри вычислительной системы, так и на внешние по отношению к ней. К внутренним параметрам относятся различные количественные и качественные характеристики процесса такие как: ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей I/O burst к CPU burst и т. д. Алгоритмы SJF и гарантированного планирования используют внутренние параметры. В качестве внешних параметров могут выступать важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и другие политические факторы. Высокий внешний приоритет может быть присвоен задаче лектора или того, кто заплатил $100 за работу в течение одного часа.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов. Давайте рассмотрим примеры использования различных режимов приоритетногопланирования.

Пусть в очередь процессов, находящихся в состоянии готовность, поступают те же процессы, что и в примере для вытесняющего алгоритма SJF, только им дополнительно еще присвоены приоритеты (табл. 2.8.). В вычислительных системах не существует определенного соглашения, какое значение приоритета – 1 или 4 считать более приоритетным. Во избежание путаницы, во всех наших примерах мы будем предполагать, что большее значение соответствует меньшему приоритету, т. е. наиболее приоритетным в нашем примере является процесс p3, а наименее приоритетным – процесс p0.

*Таблица 2.8*

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| **Процесс** | **Время появления в очереди** | **Продолжительность очередногоCPU burst** | **Приоритет** |
| **p0** | 0 | 6 | 4 |
| **p1** | 2 | 2 | 3 |
| **p2** | 6 | 7 | 2 |
| **p3** | 0 | 5 | 1 |

Как будут вести себя процессы при использовании невытесняющего приоритетного планирования? Первым для выполнения в момент времени t = 0 выбирается процесс p3, как обладающий наивысшим приоритетом. После его завершения в момент времени t = 5 в очереди процессов, готовых к исполнению, окажутся два процесса p0 и p1. Больший приоритет из них у процесса p1, он и начнет выполняться (см. табл. 2.9.). Затем в момент времени t = 8 для исполнения будет избран процесс p2, и лишь потом – процесс p0.

*Таблица 2.9*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **p0** | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И |
| **p1** |  |  | Г | Г | Г | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **p2** |  |  |  |  |  |  | Г | И | И | И | И | И | И | И |  |  |  |  |  |  |
| **p3** | И | И | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Иным будет предоставление процессора процессам в случае вытесняющего приоритетного планирования (табл. 2.10.). Первым, как и в предыдущем случае, начнет исполняться процесс p3, а по его окончании – процесс p1. Однако в момент времени t = 6 он будет вытеснен процессом p2 и продолжит свое выполнение только в момент времени t = 13. Последним, как и раньше, будет исполняться процесс p0.

*Таблица 2.10*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **p0** | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И |
| **p1** |  |  | Г | Г | Г | И | Г | Г | Г | Г | Г | Г | Г | И |  |  |  |  |  |  |
| **p2** |  |  |  |  |  |  | И | И | И | И | И | И | И |  |  |  |  |  |  |  |
| **p3** | И | И | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

В рассмотренном выше примере приоритеты процессов с течением времени не изменялись. Такие приоритеты принято называть статическими. Механизмы статической приоритетности легко реализовать, и они сопряжены с относительно небольшими издержками на выбор наиболее приоритетного процесса. Однако статические приоритеты не реагируют на изменения ситуации в вычислительной системе, которые могут сделать желательной корректировку порядка исполнения процессов. Более гибкими являются динамические приоритеты процессов, изменяющие свои значения по ходу исполнения процессов. Начальное значение динамического приоритета, присвоенное процессу, действует в течение лишь короткого периода времени, после чего ему назначается новое, более подходящее значение. Изменение динамического приоритета процесса является единственной операцией над процессами, которую мы до сих пор не рассмотрели. Как правило, изменение приоритета процессов проводится согласованно с совершением каких-либо других операций: при рождении нового процесса, при разблокировке или блокировании процесса, по истечении определенного кванта времени или по завершении процесса. Примерами алгоритмов с динамическими приоритетами являются алгоритм SJF и алгоритм гарантированного планирования. Схемы с динамической приоритетностью гораздо сложнее в реализации и связаны с большими издержками по сравнению со статическими схемами. Однако их использование предполагает, что эти издержки оправдываются улучшением работы системы.

Главная проблема приоритетного планирования заключается в том, что при ненадлежащем выборе механизма назначения и изменения приоритетов низкоприоритетные процессы могут не запускаться неопределенно долгое время. Обычно случается одно из двух. Или они все же дожидаются своей очереди на исполнение (в девять часов утра в воскресенье, когда все приличные программисты ложатся спать). Или вычислительную систему приходится выключать, и они теряются (при остановке IBM 7094 в Массачусетском технологическом институте в 1973 году были найдены процессы, запущенные в 1967 году и ни разу с тех пор не исполнявшиеся). Решение этой проблемы может быть достигнуто с помощью увеличения со временем значения приоритета процесса, находящегося в состоянии готовность. Пусть изначально процессам присваиваются приоритеты от 128 до 255. Каждый раз по истечении определенного промежутка времени значения приоритетов готовых процессов уменьшаются на 1. Процессу, побывавшему в состоянии исполнение, присваивается первоначальное значение приоритета. Даже такая грубая схема гарантирует, что любому процессу в разумные сроки будет предоставлено право на исполнение.

***Многоуровневые очереди (MultilevelQueue)***

Для систем, в которых процессы могут быть легко рассортированы по разным группам, был разработан другой класс алгоритмов планирования. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии готовность (рис. 2.11). Этим очередям приписываются фиксированные приоритеты. Например, приоритет очереди системных процессов устанавливается выше, чем приоритет очередей пользовательских процессов. А приоритет очереди процессов, запущенных студентами, ниже, чем для очереди процессов, запущенных преподавателями. Это значит, что ни один пользовательский процесс не будет выбран для исполнения, пока есть хоть один готовый системный процесс, и ни один студенческий процесс не получит в свое распоряжение процессор, если есть процессы 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 FCFS, а для интерактивных процессов – алгоритм RR. Подобный подход, получивший название многоуровневых очередей, повышает гибкость планирования: для процессов с различными характеристиками применяется наиболее подходящий им алгоритм.



Рисунок 2.11 –Несколько очередей планирования

***Многоуровневые очереди с обратной связью (MultilevelFeedbackQueue)***

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

Для простоты рассмотрим ситуацию, когда процессы в состоянии готовность организованы в 4 очереди, как на рис. 2.12. Планирование процессов между очередями осуществляется на основе вытесняющего приоритетного механизма. Чем выше на рисунке располагается очередь, тем выше ее приоритет. Процессы в очереди 1 не могут исполняться, если в очереди 0 есть хотя бы один процесс. Процессы в очереди 2 не будут выбраны для выполнения, пока есть хоть один процесс в очередях 0 и 1. И наконец, процесс в очереди 3 может получить процессор в свое распоряжение только тогда, когда очереди 0, 1 и 2 пусты. Если при работе процесса появляется другой процесс в какой-либо более приоритетной очереди, исполняющийся процесс вытесняется новым. Планирование процессов внутри очередей 0 – 2 осуществляется с использованием алгоритма RR, планирование процессов в очереди 3 основывается на алгоритме FCFS.

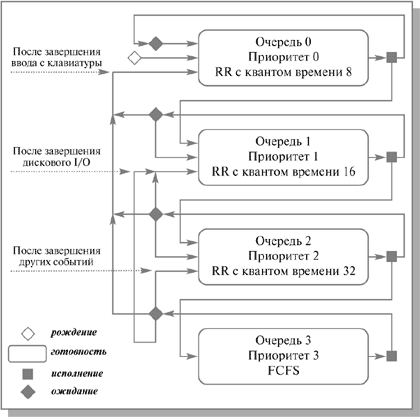


Рисунок 2.12 –Схема миграции процессов в многоуровневых очередях планирования с обратной связью.

Вытеснение процессов более приоритетными процессами и завершение процессов на схеме не показано

Родившийся процесс поступает в очередь 0. При выборе на исполнение он получает в свое распоряжение квант времени размером 8 единиц. Если продолжительность его CPU burst меньше этого кванта времени, процесс остается в очереди 0. В противном случае он переходит в очередь 1. Для процессов из очереди 1 квант времени имеет величину 16. Если процесс не укладывается в это время, он переходит в очередь 2. Если укладывается – остается в очереди 1. В очереди 2 величина кванта времени составляет 32 единицы. Если для непрерывной работы процесса и этого мало, процесс поступает в очередь 3, для которой квантование времени не применяется и, при отсутствии готовых процессов в других очередях, может исполняться до окончания своего CPU burst. Чем больше значение продолжительности CPU burst, тем в менее приоритетную очередь попадает процесс, но тем на большее процессорное время он может рассчитывать. Таким образом, через некоторое время все процессы, требующие малого времени работы процессора, окажутся размещенными в высокоприоритетных очередях, а все процессы, требующие большого счета и с низкими запросами к времени отклика, – в низкоприоритетных.

Миграция процессов в обратном направлении может осуществляться по различным принципам. Например, после завершения ожидания ввода с клавиатуры процессы из очередей 1, 2 и 3 могут помещаться в очередь 0, после завершения дисковых операций ввода-вывода процессы из очередей 2 и 3 могут помещаться в очередь 1, а после завершения ожидания всех других событий – из очереди 3 в очередь 2. Перемещение процессов из очередей с низкими приоритетами в очереди с высокими приоритетами позволяет более полно учитывать изменение поведения процессов с течением времени.

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

* Количество очередей для процессов, находящихся в состоянии готовность.
* Алгоритм планирования, действующий между очередями.
* Алгоритмы планирования, действующие внутри очередей.
* Правила помещения родившегося процесса в одну из очередей.
* Правила перевода процессов из одной очереди в другую.
* Изменяя какой-либо из перечисленных пунктов, мы можем существенно менять поведение вычислительной системы.

На этом следует остановиться, так как объем учебного пособия ограничен. С более обширным количеством алгоритмов можyj ознакомиться в [3].

Алгоритм SJF и алгоритм гарантированного планирования являются частными случаями планирования с использованием приоритетов. В более общих методах приоритетного планирования применяются многоуровневые очереди процессов, готовых к исполнению, и многоуровневые очереди с обратной связью. Будучи наиболее сложными в реализации, эти способы планирования обеспечивают гибкое поведение вычислительных систем и их адаптивность к решению задач разных классов.

## Средства коммуникации процессов

Взаимодействие процессов в вычислительной системе напоминает жизнь в коммунальной квартире. Постоянное ожидание в очереди к местам общего пользования (процессору) и ежедневная борьба за ресурсы (кто опять занял все конфорки на плите?). Для нормального функционирования процессов операционная система старается максимально обособить их друг от друга. Каждый процесс имеет собственное адресное пространство (каждая семья должна жить в отдельной комнате), нарушение которого, как правило, приводит к аварийной остановке процесса (вызов милиции). Каждому процессу по возможности предоставляются свои дополнительные ресурсы (каждая семья предпочитает иметь собственный холодильник). Тем не менее для решения некоторых задач (приготовление праздничного стола на всю квартиру) процессы могут объединять свои усилия. В настоящей лекции описываются причины взаимодействия процессов, способы их взаимодействия и возникающие при этом проблемы (попробуйте отремонтировать общую квартиру так, чтобы жильцы не перессорились друг с другом).

**Взаимодействующие процессы**

Для достижения поставленной цели различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой.

Для чего процессам нужно заниматься совместной деятельностью? Какие существуют причины для их кооперации?

* Повышение скорости работы. Пока один процесс ожидает наступления некоторого события (например, окончания операции ввода-вывода), другие могут заниматься полезной работой, направленной на решение общей задачи. В многопроцессорных вычислительных системах программа разбивается на отдельные кусочки, каждый из которых будет исполняться на своем процессоре.
* Совместное использование данных. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с разделяемым файлом, совместно изменяя их содержимое.
* Модульная конструкция какой-либо системы. Типичным примером может служить микроядерный способ построения операционной системы, когда различные ее части представляют собой отдельные процессы, взаимодействующие путем передачи сообщений через микроядро.
* Наконец, это может быть необходимо просто для удобства работы пользователя, желающего, например, редактировать и отлаживать программу одновременно. В этой ситуации процессы редактора и отладчика должны уметь взаимодействовать друг с другом.

Процессы не могут взаимодействовать, не общаясь, то есть не обмениваясь информацией. «Общение» процессов обычно приводит к изменению их поведения в зависимости от полученной информации. Если деятельность процессов остается неизменной при любой принятой ими информации, то это означает, что они на самом деле в «общении» не нуждаются. Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть ***кооперативными***или взаимодействующими процессами , в отличие от независимых процессов, не оказывающих друг на друга никакого воздействия.

Различные процессы в вычислительной системе изначально представляют собой обособленные сущности. Работа одного процесса не должна приводить к нарушению работы другого процесса. Для этого, в частности, разделены их адресные пространства и системные ресурсы, и для обеспечения корректного взаимодействия процессов требуются специальные средства и действия операционной системы. Нельзя просто поместить значение, вычисленное в одном процессе, в область памяти, соответствующую переменной в другом процессе, не предприняв каких-либо дополнительных усилий. Давайте рассмотрим основные аспекты организации совместной работы процессов.

**Категории средств обмена информацией.**

Процессы могут взаимодействовать друг с другом, только обмениваясь информацией. По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории.

* Сигнальные. Передается минимальное количество информации – один бит, "да" или "нет". Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом. Неправильная реакция на сигнал или его игнорирование могут привести к трагическим последствиям. Вспомним профессора Плейшнера из кинофильма "Семнадцать мгновений весны". Сигнал тревоги – цветочный горшок на подоконнике – был ему передан, но профессор проигнорировал его. И к чему это привело?
* Канальные. "Общение" процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.
* Разделяемая память. Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием разделяемой памяти занимается операционная система (если, конечно, ее об этом попросят). "Общение" процессов напоминает совместное проживание студентов в одной комнате общежития. Возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, но требует повышенной осторожности (если вы переложили на другое место вещи вашего соседа по комнате, а часть из них еще и выбросили). Использование разделяемой памяти для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как сигнальным и канальным средствам коммуникации для этого необходимы специальные системные вызовы. Разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

**Логическая организация механизма передачи информации**

При рассмотрении любого из средств коммуникации нас будет интересовать не их физическая реализация (общая шина данных, прерывания, аппаратно разделяемая память и т. д.), а логическая, определяющая в конечном счете механизм их использования. Некоторые важные аспекты логической реализации являются общими для всех категорий средств связи, некоторые относятся к отдельным категориям. Давайте кратко охарактеризуем основные вопросы, требующие разъяснения при изучении того или иного способа обмена информацией.

Могу ли я использовать средство связи непосредственно для обмена информацией сразу после создания процесса или первоначально необходимо предпринять определенные действия для инициализации обмена? Например, для использования общей памяти различными процессами потребуется специальное обращение к операционной системе, которая выделит необходимую область адресного пространства. Но для передачи сигнала от одного процесса к другому никакая инициализация не нужна. В то же время передача информации по линиям связи может потребовать первоначального резервирования такой линии для процессов, желающих обменяться информацией.

К этому же вопросу тесно примыкает вопрос о способе адресации при использовании средства связи. Если я передаю некоторую информацию, я должен указать, куда я ее передаю. Если я желаю получить некоторую информацию, то мне нужно знать, откуда я могу ее получить.

Различают два способа адресации: прямую и непрямую. В случае прямой адресации взаимодействующие процессы непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, указывают имена своих партнеров по взаимодействию, то такая схема адресации называется симметричной прямой адресацией. ***Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные****.* Если только один из взаимодействующих процессов, например передающий, указывает имя своего партнера по кооперации, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например ожидает получения информации от произвольного источника, то такая схема адресации называется асимметричной прямой адресацией .

При непрямой адресации данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных, имеющий свой адрес, откуда они могут быть затем изъяты каким-либо другим процессом. Примером такого объекта может служить обычная доска объявлений или рекламная газета. При этом передающий процесс не знает, как именно идентифицируется процесс, который получит информацию, а принимающий процесс не имеет представления об идентификаторе процесса, от которого он должен ее получить.

При использовании прямой адресации связь между процессами в классической операционной системе устанавливается автоматически, без дополнительных инициализирующих действий. Единственное, что нужно для использования средства связи, – это знать, как идентифицируются процессы, участвующие в обмене данными.

При использовании непрямой адресации инициализация средства связи может и не требоваться. Информация, которой должен обладать процесс для взаимодействия с другими процессами, – это некий идентификатор промежуточного объекта для хранения данных, если он, конечно, не является единственным и неповторимым в вычислительной системе для всех процессов.

**Информационная валентность процессов и средств связи**

Следующий важный вопрос – это вопрос об информационной валентности связи. Слово «валентность» здесь использовано по аналогии с химией. Сколько процессов может быть одновременно ассоциировано с конкретным средством связи? Сколько таких средств связи может быть задействовано между двумя процессами?

Понятно, что при прямой адресации только одно фиксированное средство связи может быть задействовано для обмена данными между двумя процессами, и только эти два процесса могут быть ассоциированы с ним. При непрямой адресации может существовать более двух процессов, использующих один и тот же объект для данных, и более одного объекта может быть использовано двумя процессами.

К этой же группе вопросов следует отнести и вопрос о направленности связи. Является ли связь однонаправленной или двунаправленной? Под однонаправленной связью мы будем понимать связь, при которой каждый процесс, ассоциированный с ней, может использовать средство связи либо только для приема информации, либо только для ее передачи. При двунаправленной связи каждый процесс, участвующий в общении, может использовать связь и для приема, и для передачи данных. В коммуникационных системах принято называть однонаправленную связь симплексной , двунаправленную связь с поочередной передачей информации в разных направлениях – полудуплексной , а двунаправленную связь с возможностью одновременной передачи информации в разных направлениях – дуплексной . Прямая и непрямая адресация не имеет непосредственного отношения к направленности связи.

**Буферизация**

Может ли линия связи сохранять информацию, переданную одним процессом, до ее получения другим процессом или помещения в промежуточный объект? Каков объем этой информации? Иными словами, речь идет о том, обладает ли канал связи ***буфером*** и каков объем этого буфера. Здесь можно выделить три принципиальных варианта.

1. Буфер нулевой емкости или отсутствует. Никакая информация не может сохраняться на линии связи. В этом случае процесс, посылающий информацию, должен ожидать, пока процесс, принимающий информацию, не соблаговолит ее получить, прежде чем заниматься своими дальнейшими делами (в реальности этот случай никогда не реализуется).
2. Буфер ограниченной емкости. Размер буфера равен n, то есть линия связи не может хранить до момента получения более чем n единиц информации. Если в момент передачи данных в буфере хватает места, то передающий процесс не должен ничего ожидать. Информация просто копируется в буфер. Если же в момент передачи данных буфер заполнен или места недостаточно, то необходимо задержать работу процесса отправителя до появления в буфере свободного пространства.
3. Буфер неограниченной емкости. Теоретически это возможно, но практически вряд ли реализуемо. Процесс, посылающий информацию, никогда не ждет окончания ее передачи и приема другим процессом.

При использовании канального средства связи с непрямой адресацией под емкостью буфера обычно понимается количество информации, которое может быть помещено в промежуточный объект для хранения данных.

**Поток ввода/вывода и сообщения**

Существует две модели передачи данных по каналам связи – поток ввода-вывода и сообщения. При передаче данных с помощью потоковой модели операции передачи/приема информации вообще не интересуются содержимым данных. Процесс, прочитавший 100 байт из линии связи, не знает и не может знать, были ли они переданы одновременно, т. е. одним куском или порциями по 20 байт, пришли они от одного процесса или от разных. Данные представляют собой простой поток байтов, без какой-либо их интерпретации со стороны системы. Примерами потоковых каналов связи могут служить pipe и FIFO, описанные ниже.

Одним из наиболее простых способов передачи информации между процессами по линиям связи является передача данных через ***pipe*** (канал, трубу или, как его еще называют в литературе, конвейер). Представим себе, что у нас есть некоторая труба в вычислительной системе, в один из концов которой процессы могут "сливать" информацию, а из другого конца принимать полученный поток. Такой способ реализует потоковую модель ввода/вывода. Информацией о расположении трубы в операционной системе обладает только процесс, создавший ее. Этой информацией он может поделиться исключительно со своими наследниками – процессами-детьми и их потомками. Поэтому использовать pipe для связи между собой могут только родственные процессы, имеющие общего предка, создавшего данный канал связи.

Если разрешить процессу, создавшему трубу, сообщать о ее местонахождении в системе другим процессам, сделав вход и выход трубы каким-либо образом видимыми для всех остальных, например, зарегистрировав ее в операционной системе под определенным именем, мы получим объект, который принято называть FIFO или именованный pipe. Именованный pipe может использоваться для организации связи между любыми процессами в системе.

В модели сообщений процессы налагают на передаваемые данные некоторую структуру. Весь поток информации они разделяют на отдельные сообщения, вводя между данными, по крайней мере, границы сообщений. Примером границ сообщений являются точки между предложениями в сплошном тексте или границы абзаца. Кроме того, к передаваемой информации могут быть присоединены указания на то, кем конкретное сообщение было послано и для кого оно предназначено. Примером указания отправителя могут служить подписи под эпиграфами в книге. Все сообщения могут иметь одинаковый фиксированный размер или могут быть переменной длины. В вычислительных системах используются разнообразные средства связи для передачи сообщений: очереди сообщений, sockets (гнезда) и т. д. Часть из них мы рассмотрим подробнее в дальнейшем, в частности очереди сообщений будут рассмотрены в лекции 6, а гнезда (иногда их еще называют по транслитерации английского названия – сокеты) в лекции 14.

И потоковые линии связи, и каналы сообщений всегда имеют буфер конечной длины. Когда мы будем говорить о емкости буфера для потоков данных, мы будем измерять ее в байтах. Когда мы будем говорить о емкости буфера для сообщений, мы будем измерять ее в сообщениях.

**Надежность средств связи**

Одним из существенных вопросов при рассмотрении всех категорий средств связи является вопрос об их надежности. Мы все знаем, как бывает тяжело расслышать собеседника по вечно трещащему телефону или разобрать, о чем сообщается в телеграмме: "Прибдупыездом в вонедельник 33 июня в 25.34. Пама".

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

1. Не происходит потери информации.
2. Не происходит повреждения информации.
3. Не появляется лишней информации.
4. Не нарушается порядок данных в процессе обмена.

Очевидно, что передача данных через разделяемую память является надежным способом связи. То, что мы сохранили в разделяемой памяти, будет считано другими процессами в первозданном виде, если, конечно, не произойдет сбоя в питании компьютера. Для других средств коммуникации, как видно из приведенных выше примеров, это не всегда верно.

Каким образом в вычислительных системах пытаются бороться с ненадежностью коммуникаций? Давайте рассмотрим возможные варианты на примере обмена данными через линию связи с помощью сообщений. Для обнаружения повреждения информации будем снабжать каждое передаваемое сообщение некоторой контрольной суммой, вычисленной по посланной информации. При приеме сообщения контрольную сумму будем вычислять заново и проверять ее соответствие пришедшему значению. Если данные не повреждены (контрольные суммы совпадают), то подтвердим правильность их получения. Если данные повреждены (контрольные суммы не совпадают), то сделаем вид, что сообщение к нам не поступило. Вместо контрольной суммы можно использовать специальное кодирование передаваемых данных с помощью кодов, исправляющих ошибки. Такое кодирование позволяет при числе искажений информации, не превышающем некоторого значения, восстановить первоначальные неискаженные данные. Если по прошествии некоторого интервала времени подтверждение правильности полученной информации не придет на передающий конец линии связи, будем считать информацию утерянной и пошлем ее повторно. Для того чтобы избежать двойного получения одной и той же информации, на приемном конце линии связи должен осуществляться соответствующий контроль. Для гарантии правильного порядка получения сообщений будем их нумеровать. При приеме сообщения с номером, не соответствующим ожидаемому, поступаем с ним как с утерянным и ждем сообщения с правильным номером.

Подобные действия могут быть возложены:

* на операционную систему;
* на процессы, обменивающиеся данными;
* совместно на систему и процессы, разделяя их ответственность. Операционная система может обнаруживать ошибки при передаче данных и извещать об этом взаимодействующие процессы для принятия ими решения о дальнейшем поведении.

**Как завершается связь?**

Наконец, важным вопросом при изучении средств обмена данными является вопрос прекращения обмена. Здесь нужно выделить два аспекта: требуются ли от процесса какие-либо специальные действия по прекращению использования средства коммуникации и влияет ли такое прекращение на поведение других процессов. Для способов связи, которые не подразумевали никаких инициализирующих действий, обычно ничего специального для окончания взаимодействия предпринимать не надо. Если же установление связи требовало некоторой инициализации, то, как правило, при ее завершении бывает необходимо выполнить ряд операций, например сообщить операционной системе об освобождении выделенного связного ресурса.

Если кооперативные процессы прекращают взаимодействие согласованно, то такое прекращение не влияет на их дальнейшее поведение. Иная картина наблюдается при несогласованном окончании связи одним из процессов. Если какой-либо из взаимодействующих процессов, не завершивших общение, находится в этот момент в состоянии ожидания получения данных либо попадает в такое состояние позже, то операционная система обязана предпринять некоторые действия для того, чтобы исключить вечное блокирование этого процесса. Обычно это либо прекращение работы ожидающего процесса, либо его извещение о том, что связи больше нет (например, с помощью передачи заранее определенного сигнала).

**Нити исполнения**

Рассмотренные выше аспекты логической реализации относятся к средствам связи, ориентированным на организацию взаимодействия различных процессов. Однако усилия, направленные на ускорение решения задач в рамках классических операционных систем, привели к появлению совершенно иных механизмов, к изменению самого понятия "процесс".

В свое время внедрение идеи мультипрограммирования позволило повысить пропускную способность компьютерных систем, т. е. уменьшить среднее время ожидания результатов работы процессов. Но любой отдельно взятый процесс в мультипрограммной системе никогда не может быть выполнен быстрее, чем при работе в однопрограммном режиме на том же вычислительном комплексе. Тем не менее, если алгоритм решения задачи обладает определенным внутренним параллелизмом, мы могли бы ускорить его работу, организовав взаимодействие нескольких процессов. Рассмотрим следующий пример. Пусть у нас есть следующая программа на псевдоязыке программирования:

Ввести массив a

Ввести массив b

Ввести массив c

a = a + b

c = a + c

Вывести массив c

При выполнении такой программы в рамках одного процесса этот процесс четырежды будет блокироваться, ожидая окончания операций ввода-вывода. Но наш алгоритм обладает внутренним параллелизмом. Вычисление суммы массивов a + b можно было бы выполнять параллельно с ожиданием окончания операции ввода массива c.

Ввести массив a

Ожидание окончания операции ввода

Ввести массив b

Ожидание окончания операции ввода

Ввести массив с

Ожидание окончания операции ввода a = a + b

c = a + c

Вывести массив с

Ожидание окончания операции вывода

Такое совмещение операций по времени можно было бы реализовать, используя два взаимодействующих процесса. Для простоты будем полагать, что средством коммуникации между ними служит разделяемая память. Тогда наши процессы могут выглядеть следующим образом.

Процесс 1 Процесс 2

Ввести массив a Ожидание ввода

Ожидание окончания массивов a и b

операции ввода

Ввести массив b

Ожидание окончания

операции ввода

Ввести массив с

Ожидание окончания a = a + b

операции ввода

c = a + c

Вывести массив с

Ожидание окончания

операции вывода

Казалось бы, мы предложили конкретный способ ускорения решения задачи. Однако в действительности дело обстоит не так просто. Второй процесс должен быть создан, оба процесса должны сообщить операционной системе, что им необходима память, которую они могли бы разделить с другим процессом, и, наконец, нельзя забывать о переключении контекста. Поэтому реальное поведение процессов будет выглядеть примерно так.

Процесс 1 Процесс 2

Создать процесс 2

Переключение контекста

Выделение общей

памяти

Ожидание ввода

a и b

Переключение контекста

Выделение общей памяти

Ввести массив a

Ожидание окончания

операции ввода

Ввести массив b

Ожидание окончания

операции ввода

Ввести массив с

Ожидание окончания

операции ввода

Переключение контекста

a = a + b

Переключение контекста

c = a + c

Вывести массив с

Ожидание окончания

операции вывода

Очевидно, что мы можем не только не выиграть во времени при решении задачи, но даже и проиграть, так как временные потери на создание процесса, выделение общей памяти и переключение контекста могут превысить выигрыш, полученный за счет совмещения операций.

Для того чтобы реализовать нашу идею, введем новую абстракцию внутри понятия "процесс" – нить исполнения или просто нить (в англоязычной литературе используется термин thread ). Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Теперь процесс представляется как совокупность взаимодействующих нитей и выделенных ему ресурсов. Процесс, содержащий всего одну нить исполнения, идентичен процессу в том смысле, который мы употребляли ранее. Для таких процессов мы в дальнейшем будем использовать термин "традиционный процесс". Иногда нити называют облегченными процессами или мини-процессами, так как во многих отношениях они подобны традиционным процессам. Нити, как и процессы, могут порождать нити-потомки, правда, только внутри своего процесса, и переходить из одного состояния в другое. Состояния нитей аналогичны состояниям традиционных процессов. Из состояния *рождение* процесс приходит содержащим всего одну нить исполнения. Другие нити процесса будут являться потомками этой нити-прародительницы. Мы можем считать, что процесс находится в состоянии *готовность*, если хотя бы одна из его нитей находится в состоянии *готовность* и ни одна из нитей не находится в состоянии *исполнение*. Мы можем считать, что процесс находится в состоянии *исполнение*, если одна из его нитей находится в состоянии *исполнение*. Процесс будет находиться в состоянии *ожидание*, если все его нити находятся в состоянии *ожидание*. Наконец, процесс находится в состоянии *закончил исполнение*, если все его нити находятся в состоянии *закончил исполнение*. Пока одна нить процесса заблокирована, другая нить того же процесса может выполняться. Нити разделяют процессор так же, как это делали традиционные процессы, в соответствии с рассмотренными алгоритмами планирования.

Поскольку нити одного процесса разделяют существенно больше ресурсов, чем различные процессы, то операции создания новой нити и переключения контекста между нитями одного процесса занимают значительно меньше времени, чем аналогичные операции для процессов в целом. Предложенная нами схема совмещения работы в терминах нитей одного процесса получает право на существование.

Нить 1 Нить 2

Создать нить 2

Переключение контекста нитей

Ожидание ввода a и b

Переключение контекста нитей

Ввести массив a

Ожидание окончания

операции ввода

Ввести массив b

Ожидание окончания

операции ввода

Ввести массив с

Ожидание окончания

операции ввода

Переключение контекста нитей

a = a + b

Переключение контекста нитей

c = a + c

Вывести массив с

Ожидание окончания

операции вывода

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

## Синхронизация траекторий процессов

Для более легкого восприятия учебного материал временно отвлечемся от операционных систем, процессов и нитей исполнения и поговорим о некоторых «активностях». Под активностями будем понимать последовательное выполнение ряда действий, направленных на достижение определенной цели. Активности могут иметь место в программном и техническом обеспечении, в обычной деятельности людей и животных. Мы будем разбивать активности на некоторые неделимые, или атомарные, операции. Например, активность "приготовление бутерброда" можно разбить на следующие атомарные операции:

1. Отрезать ломтик хлеба.
2. Отрезать ломтик колбасы.
3. Намазать ломтик хлеба маслом.
4. Положить ломтик колбасы на подготовленный ломтик хлеба.

Неделимые операции могут иметь внутренние невидимые действия (взять батон хлеба в левую руку, взять нож в правую руку, произвести отрезание). Мы же называем их неделимыми потому, что считаем выполняемыми за раз, без прерывания деятельности.

Пусть имеется две активности

P: a b c

Q: d e f

где a, b, c, d, e, f – атомарные операции. При последовательном выполнении активностей мы получаем такую последовательность атомарных действий:

PQ: a b c d e f

Что произойдет при исполнении этих активностей псевдопараллельно, в режиме разделения времени? Активности могут расслоиться на неделимые операции с различным чередованием, то есть может произойти то, что на английском языке принято называть словом interleaving. Возможные варианты чередования:

а b c d e f

a b d c e f

a b d e c f

a b d e f c

a d b c e f

......

d e f a b c

Атомарные операцииактивностей могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри активностей. Так как псевдопараллельное выполнение двух активностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеется две активностиP и Q, состоящие из двух атомарных операций каждая:

P: x=2 Q: x=3

y=x-1 y=x+1

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для активностей общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). . Мы будем говорить, что набор активностей (например, программ) детерминирован, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он недетерминирован . Выше приведен пример недетерминированного набора программ. Понятно, что детерминированный наборактивностей можно безбоязненно выполнять в режиме разделения времени. Для недетерминированного набора такое исполнение нежелательно.

Можно ли до получения результатов определить, является ли наборактивностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна. Изложим их применительно к программам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных – это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы R(P) (R от слова read) – суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова write) – суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

P: x=u+v

y=x\*w

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменнаяx присутствует как в R(P), так и в W(P).

Теперь сформулируем условия Бернстайна.

Если для двух данных активностейP и Q:

* пересечение W(P) и W(Q) пусто,
* пересечение W(P) с R(Q) пусто,
* пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а может быть, и нет.

Случай двух активностей естественным образом обобщается на их большее количество.

У словия Бернстайна информативны, но слишком жестки. По сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы детерминированный набор образовывали активности, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований атомарных операций, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный доступ программ к некоторым данным.

Про недетерминированный набор программ (и активностей вообще) говорят, что он имеет ***racecondition*** ( состояние гонки, состояние состязания). В приведенном выше примере процессы состязаются за вычисление значений переменных x и y.

Задачу упорядоченного доступа к разделяемым данным (устранение racecondition) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется взаимоисключением (mutualexclusion) . Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними взаимоисключениями уже не обойтись, нужна взаимосинхронизация поведения программ.

**Критическая секция**

Важным понятием при изучении способов синхронизации процессов является понятие критической секции (criticalsection) программы. ***Критическая секция*** – это часть программы, исполнение которой может привести к возникновению racecondition для определенного набора программ. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом (рис.2.13). Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций программ. Реализация взаимоисключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам, участвующим во взаимодействии, критическая секция начинает выполняться как атомарная операция.

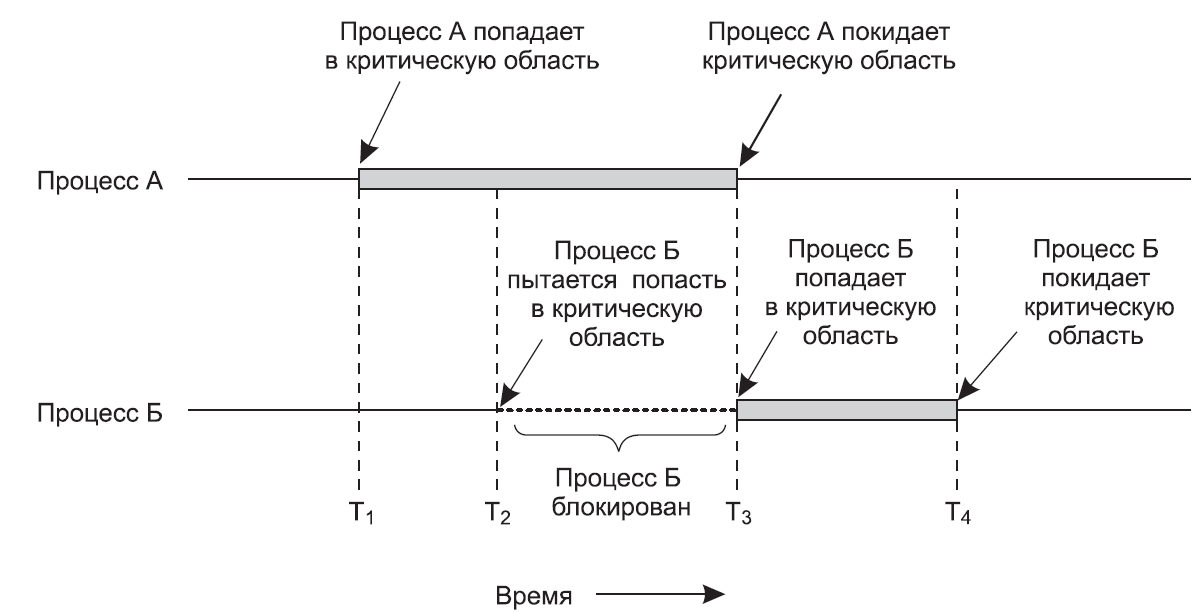


Рисунок 2.13 – Взаимное исключение использования критических областей

Рассмотрим пример, в котором псевдопараллельные взаимодействующие процессы представлены действиями различных студентов (таблица 2.11):

Здесь критический участок для каждого процесса – от операции "Обнаруживает, что хлеба нет" до операции "Возвращается в комнату" включительно. В результате отсутствия взаимоисключения мы из ситуации "Нет хлеба" попадаем в ситуацию "Слишком много хлеба". Если бы этот критический участок выполнялся как атомарная операция – "Достает два батона хлеба", то проблема образования излишков была бы снята.

*Таблица 2.11*

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| **Время** | **Студент 1** | **Студент 2** | **Студент 3** |
| 17-05 | Приходит в комнату |  |  |
| 17-07 | Обнаруживает, что хлеба нет |  |  |
| 17-09 | Уходит в магазин |  |  |
| 17-11 |  | Приходит в комнату |  |
| 17-13 |  | Обнаруживает, что хлеба нет |  |
| 17-15 |  | Уходит в магазин |  |
| 17-17 |  |  | Приходит в комнату |
| 17-19 |  |  | Обнаруживает, что хлеба нет |
| 17-21 |  |  | Уходит в магазин |
| 17-23 | Приходит в магазин |  |  |
| 17-25 | Покупает 2 батона на всех |  |  |
| 17-27 | Уходит из магазина |  |  |
| 17-29 |  | Приходит в магазин |  |
| 17-31 |  | Покупает 2 батона на всех |  |
| 17-33 |  | Уходит из магазина |  |
| 17-35 |  |  | Приходит в магазин |
| 17-37 |  |  | Покупает 2 батона на всех |
| 17-39 |  |  | Уходит из магазина |
| 17-41 | Возвращается в комнату |  |  |
| 17-43 |  |  |  |
| 17-45 |  |  |  |
| 17-47 |  | Возвращается в комнату |  |
| 17-49 |  |  |  |
| 17-51 |  |  |  |
| 17-53 |  |  | Возвращается в комнату |

Сделать процесс добывания хлеба атомарной операцией можно было бы следующим образом: перед началом этого процесса закрыть дверь изнутри на засов и уходить добывать хлеб через окно, а по окончании процесса вернуться в комнату через окно и отодвинуть засов. Тогда пока один студент добывает хлеб, все остальные находятся в состоянии ожидания под дверью (таблица 2.12).

*Таблица 2.12*

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

Итак, для решения задачи необходимо, чтобы в том случае, когда процесс находится в своем критическом участке, другие процессы не могли войти в свои критические участки. Мы видим, что критический участок должен сопровождаться прологом ( entrysection ) – «закрыть дверь изнутри на засов» – и эпилогом ( exitsection ) – "отодвинуть засов", которые не имеют отношения к активности одиночного процесса. Во время выполнения пролога процесс должен, в частности, получить разрешение на вход в критический участок, а во время выполнения эпилога – сообщить другим процессам, что он покинул критическую секцию.

В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

while (some condition) {

entry section

critical section

exitsection

remaindersection

}

Здесь под remaindersection понимаются все атомарные операции, не входящие в критическую секцию.

Оставшаяся часть этой лекции посвящена различным способам программной организации пролога и эпилога критического участка в случае, когда очередность доступа к критическому участку не имеет значения.

**Требования, предъявляемые к алгоритмам**

Организация взаимоисключения для критических участков, конечно, позволит избежать возникновения racecondition, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов. Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих критические участки, если они могут проходить их в произвольном порядке.

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test ) являются атомарными операциями.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс Pi исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях. Это условие получило название условия взаимоисключения (mutualexclusion).
4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remaindersection, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress) .
5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (boundwaiting) .

Надо заметить, что описание соответствующего алгоритма в нашем случае означает описание способа организации пролога и эпилога для критической секции.

**Запрет прерываний**

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

while (somecondition) {

запретить все прерывания

critical section

разрешитьвсепрерывания

remainder section

}

Поскольку выход процесса из состояния *исполнение* без его завершения осуществляется по прерыванию, внутри критической секции никто не может вмешаться в его работу. Однако такое решение может иметь далеко идущие последствия, поскольку позволяет процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе. Допустим, что пользователь случайно или по злому умыслу запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в такой ситуации не обойтись.

Тем не менее запрет и разрешение прерываний часто применяются как пролог и эпилог к критическим секциям внутри самой операционной системы, например при обновлении содержимого PCB.

**Переменная-замок**

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из критической секции процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе " Критическая секция ").

shared int lock = 0;

/\* shared означает, что \*/

/\* переменная является разделяемой \*/

while (somecondition) {

while(lock); lock = 1;

critical section

lock = 0;

remaindersection

}

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию взаимоисключения, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

**Строгое чередование**

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

sharedintturn = 0;

while (some condition) {

while(turn != i);

critical section

turn = 1-i;

remainder section

}

Очевидно, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение turn равно 1, и процесс P0 готов войти в критический участок, он не может сделать этого, даже если процесс P1 находится в **remaindersection**.

**Флаги готовности**

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в критический участок

sharedintready[2] = {0, 0};

Когда i-й процесс готов войти в критическую секцию, он присваивает элементу массива ready[i] значение равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

while (some condition) {

ready[i] = 1;

while(ready[1-i]);

critical section

ready[i] = 0;

remaindersection

}

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (deadlock). (Подробнее о тупиковых ситуациях рассказывается в лекции 7.)

**Алгоритм Петерсона**

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

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

shared int turn;

while (some condition) {

ready[i] = 1;

turn =1-i;

while(ready[1-i] && turn == 1-i);

critical section

ready[i] = 0;

remainder section

}

При исполнении пролога критической секции процесс Pi заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия взаимоисключения методом от противного. Пусть оба процесса одновременно оказались внутри своих критических секций. Заметим, что процесс Pi может войти в критическую секцию, только если ready[1-i] == 0 или turn == i. Заметим также, что если оба процесса выполняют свои критические секции одновременно, то значения флагов готовности для обоих процессов совпадают и равны 1. Могли ли оба процесса войти в критические секции из состояния, когда они оба одновременно находились в процессе выполнения цикла while? Нет, так как в этом случае переменная turn должна была бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных измениться не могут). Пусть процесс P0 первым вошел в критический участок, тогда процесс P1 должен был выполнить перед вхождением в цикл while по крайней мере один предваряющий оператор (turn = 0;). Однако после этого он не может выйти из цикла до окончания критического участка процесса P0, так как при входе в цикл ready[0] == 1 и turn == 0, и эти значения не могут измениться до тех пор, пока процесс P0 не покинет свой критический участок. Мы пришли к противоречию. Следовательно, имеет место взаимоисключение.

Докажем выполнение условия прогресса. Возьмем, без ограничения общности, процесс P0. Заметим, что он не может войти в свою критическую секцию только при совместном выполнении условий ready[1] == 1 и turn == 1. Если процесс P1 не готов к выполнению критического участка, то ready[1] == 0, и процесс P0 может осуществить вход. Если процесс P1 готов к выполнению критического участка, то ready[1] == 1 и переменная turn имеет значение 0 либо 1, позволяя процессу P0 либо процессу P1 начать выполнение критической секции. Если процесс P1 завершил выполнение критического участка, то он сбросит свой флаг готовности ready[1] == 0, разрешая процессу P0 приступить к выполнению критической работы. Таким образом, условие прогресса выполняется.

Отсюда же вытекает выполнение условия ограниченного ожидания. Так как в процессе ожидания разрешения на вход процесс P0 не изменяет значения переменных, он сможет начать исполнение своего критического участка после не более чем одного прохода по критической секции процесса P1.

**Алгоритм булочной (Bakeryalgorithm)**

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название алгоритм булочной, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) 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 процессов 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

sharedenum {false, true} choosing[n];

sharedintnumber[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(a0, a1, ...., an) – это число k такое, что

k >= ai для всех i = 0, ...,n

Структура процесса Pi 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;

remaindersection

}

Доказательство того, что этот алгоритм удовлетворяет условиям 1 – 5, выполните самостоятельно в качестве упражнения.

**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. Мы уже обращались к общепринятому hardware для решения задачи реализации взаимоисключений, когда говорили об использовании механизма запрета/разрешения прерываний.

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. Давайте обсудим, как концепции таких команд могут использоваться 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 о выполнении функции

intTest\_and\_Set (int \*target){

inttmp = \*target;

\*target = 1;

return tmp;

}

С использованием этой атомарной команды мы можем модифицировать наш алгоритм для переменной-замка, так чтобы он обеспечивал взаимоисключения

shared int lock = 0;

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 для алгоритмов. Подумайте, как его следует изменить для соблюдения всех условий.

**Команда Swap (обменять значения)**

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

void Swap (int \*a, int \*b){

inttmp = \*a;

\*a = \*b;

\*b = tmp;

}

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

shared int lock = 0;

int key;

while (some condition) {

key = 1;

do Swap(&lock,&key);

while (key);

critical section

lock = 0;

remaindersection

}

**Семафоры**

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 году.

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать). Классическое определение этих операций выглядит следующим образом:

P(S): пока S == 0 процесс блокируется;

S = S – 1;

V(S): S = S + 1;

Эта запись означает следующее: при выполнении операции P над семафоромS сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над семафоромS к его значению просто прибавляется 1. В момент создания семафор может быть инициализирован любым неотрицательным значением.

Подобные переменные-семафоры могут с успехом применяться для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях реализуются с помощью специальных системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

**Решение проблемы producer-consumer с помощью семафоров**

Одной из типовых задач, требующих организации взаимодействия процессов, является задача producer-consumer (производитель-потребитель). Пусть два процесса обмениваются информацией через буфер ограниченного размера. Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда. На этом уровне деятельность потребителя и производителя можно описать следующим образом.

Producer: while(1) {

produce\_item;

put\_item;

}

Consumer: while(1) {

get\_item;

consume\_item;

}

Если буфер заполнен, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации. Если буфер пуст, то потребитель должен дожидаться нового сообщения. Как можно реализовать эти условия с помощью семафоров? Возьмем три семафора: empty, full и mutex. Семафорfull будем использовать для гарантии того, что потребитель будет ждать, пока в буфере появится информация. Семафорempty будем использовать для организации ожидания производителя при заполненном буфере, а семафорmutex – для организации взаимоисключения на критических участках, которыми являются действия put\_item и get\_item (операции "положить информацию" и "взять информацию" не могут пересекаться, так как в этом случае возникнет опасность искажения информации). Тогда решение задачи на C-подобном языке выглядит так:

Semaphoremutex = 1;

Semaphoreempty = N; /\* где N – емкость буфера\*/

Semaphore full = 0;

Producer:

while(1) {

produce\_item;

P(empty);

P(mutex);

put\_item;

V(mutex);

V(full);

}

Consumer:

while(1) {

P(full);

P(mutex);

get\_item;

V(mutex);

V(empty);

consume\_item;

}

Легко убедиться, что это действительно корректное решение поставленной задачи. Попутно заметим, что семафоры использовались здесь для достижения двух целей: организации взаимоисключения на критическом участке и взаимосинхронизации скорости работы процессов.

**Мониторы**

Хотя решение задачи producer-consumer с помощью семафоров выглядит достаточно изящно, программирование с их использованием требует повышенной осторожности и внимания, чем отчасти напоминает программирование на языке Ассемблера. Допустим, что в рассмотренном примере мы случайно поменяли местами операцииP, сначала выполнив операцию для семафораmutex, а уже затем для семафоровfull и empty. Допустим теперь, что потребитель, войдя в свой критический участок ( mutex сброшен), обнаруживает, что буфер пуст. Он блокируется и начинает ждать появления сообщений. Но производитель не может войти в критический участок для передачи информации, так как тот заблокирован потребителем. Получаем тупиковую ситуацию.

В сложных программах произвести анализ правильности использования семафоров с карандашом в руках становится очень непросто. В то же время обычные способы отладки программ зачастую не дают результата, поскольку возникновение ошибок зависит от interleavingатомарных операций, и ошибки могут быть трудновоспроизводимы. Для того чтобы облегчить работу программистов, в 1974 году Хором (Hoare) был предложен механизм еще более высокого уровня, чем семафоры, получивший название мониторов. Мы с вами рассмотрим конструкцию, несколько отличающуюся от оригинальной.

Мониторы представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. Монитор обладает собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функций-методов, принадлежащих монитору. В свою очередь, эти функции-методы могут использовать в работе только данные, находящиеся внутри монитора, и свои параметры. На абстрактном уровне можно описать структуру монитора следующим образом:

monitormonitor\_name{

описание внутренних переменных ;

void m1(...){...

}

void m2(...){...

}

...

voidmn(...){...

}

{

блок инициализации

внутренних переменных;

}

}

Здесь функции m1,..., mn представляют собой функции-методы монитора, а блок инициализации внутренних переменных содержит операции, которые выполняются один и только один раз: при создании монитора или при самом первом вызове какой-либо функции-метода до ее исполнения.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т. е. находиться в состоянии *готовность* или *исполнение*, внутри данного монитора. Поскольку мониторы представляют собой особые конструкции языка программирования, компилятор может отличить вызов функции, принадлежащей монитору, от вызовов других функций и обработать его специальным образом, добавив к нему пролог и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на компилятор, а не на программиста, работа программиста при использовании мониторов существенно упрощается, а вероятность возникновения ошибок становится меньше.

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нам нужны еще и средства организации очередности процессов, подобно семафорам full и empty в предыдущем примере. Для этого в мониторах было введено понятие условных переменных (conditionvariables)1, над которыми можно совершать две операции wait и signal, отчасти похожие на операции P и V над семафорами.

Если функциямонитора не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию wait над какой-либо условной переменной. При этом процесс, выполнивший операцию wait, блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию signal над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов дожидались операцииsignal для этой переменной, то активным становится только один из них. Что можно предпринять для того, чтобы у нас не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри монитора? Хор предложил, чтобы пробужденный процесс подавлял исполнение разбудившего процесса, пока он сам не покинет монитор. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает монитор немедленно после исполнения операцииsignal. Мы будем придерживаться подхода Хансена.

*Необходимо отметить, что условные переменные, в отличие от семафоров Дейкстры, не умеют запоминать предысторию. Это означает, что операция signal всегда должна выполняться после операции wait. Если операция signal выполняется над условной переменной, с которой не связано ни одного заблокированного процесса, то информация о произошедшем событии будет утеряна. Следовательно, выполнение операцииwaitвсегда будет приводить к блокированию процесса.*

Давайте применим концепцию мониторов к решению задачи производитель-потребитель.

monitorProducerConsumer {

condition full, empty;

int count;

void put() {

if(count == N) full.wait;

put\_item;

count += 1;

if(count == 1) empty.signal;

}

void get() {

if (count == 0) empty.wait;

get\_item();

count -= 1;

if(count == N-1) full.signal;

}

{

count = 0;

}

}

Producer:

while(1) {

produce\_item;

ProducerConsumer.put();

}

Consumer:

while(1) {

ProducerConsumer.get();

consume\_item;

}

Легко убедиться, что приведенный пример действительно решает поставленную задачу.

Реализация мониторов требует разработки специальных языков программирования и компиляторов для них. Мониторы встречаются в таких языках, как параллельный Евклид, параллельный Паскаль, Java и т. д. Эмуляция мониторов с помощью системных вызовов для обычных широко используемых языков программирования не так проста, как эмуляция семафоров. Поэтому можно пользоваться еще одним механизмом со скрытыми взаимоисключениями, механизмом, о котором мы уже упоминали, – передачей сообщений.

**Сообщения**

Для прямой и непрямой адресации достаточно двух примитивов, чтобы описать передачу сообщений по линии связи – send и receive. В случае прямой адресации мы будем обозначать их так:

|  |
| --- |
| send(P, message) – послать сообщение message процессу P ; |
| receive(Q, message) – получить сообщение message от процесса Q. |

В случае непрямой адресации мы будем обозначать их так:

|  |
| --- |
| send(A, message) – послать сообщение message в почтовый ящик A ; |
| receive(A, message) – получить сообщение message из почтового ящика A. |

Примитивы send и receive уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер. Реализация решения задачи producer-consumer для таких примитивов становится неприлично тривиальной. Надо отметить, что, несмотря на простоту использования, передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами.

**Эквивалентность семафоров, мониторов и сообщений**

Мы рассмотрели три высокоуровневых механизма, использующихся для организации взаимодействия процессов. Можно показать, что в рамках одной вычислительной системы, когда процессы имеют возможность использовать разделяемую память, все они эквивалентны. Это означает, что любые два из предложенных механизмов могут быть реализованы на базе третьего, оставшегося механизма.

Рассмотрим сначала, как *реализовать мониторы с помощью семафоров*. Для этого нам нужно уметь реализовывать взаимоисключения при входе в монитор и условные переменные. Возьмем семафор mutex с начальным значением 1 для реализации взаимоисключения при входе в монитор и по одному семафору ci для каждой условной переменной. Кроме того, для каждой условной переменной заведем счетчик fi для индикации наличия ожидающих процессов. Когда процесс входит в монитор, компилятор будет генерировать вызов функции monitor\_enter, которая выполняет операцию P над семафором mutex для данного монитора. При нормальном выходе из монитора (то есть при выходе без вызова операции signal для условной переменной ) компилятор будет генерировать вызов функции monitor\_exit, которая выполняет операцию V над этим семафором.

Для выполнения операции wait над условной переменной компилятор будет генерировать вызов функции wait, которая выполняет операцию V для семафора mutex, разрешая другим процессам входить в монитор, и выполняет операцию P над соответствующим семафором ci, блокируя вызвавший процесс. Для выполнения операции signal над условной переменной компилятор будет генерировать вызов функции signal\_exit, которая выполняет операцию V над ассоциированным семафором ci (если есть процессы, ожидающие соответствующего события), и выход из монитора, минуя функцию monitor\_exit.

Semaphore mutex = 1;

void monitor\_enter(){

P(mutex);

}

void monitor\_exit(){

V(mutex);

}

Semaphore ci = 0;

int fi = 0;

void wait(i){

fi=fi + 1;

V(mutex);

P(ci);

fi=fi - 1;

}

void signal\_exit(i){

if (fi)V(ci);

else V(mutex);

}

Заметим, что при выполнении функции signal\_exit, если кто-либо ожидал этого события, процесс покидает монитор без увеличения значения семафора mutex, не разрешая тем самым всем процессам, кроме разбуженного, войти в монитор. Это увеличение совершит разбуженный процесс, когда покинет монитор обычным способом или когда выполнит новую операцию wait над какой-либо условной переменной.

Рассмотрим теперь, как *реализовать передачу сообщений, используя семафоры*. Для простоты опишем реализацию только одной очереди сообщений. Выделим в разделяемой памяти достаточно большую область под хранение сообщений, там же будем записывать, сколько пустых и заполненных ячеек находится в буфере, хранить ссылки на списки процессов, ожидающих чтения и памяти. Взаимоисключение при работе с разделяемой памятью будем обеспечивать семафором mutex. Также заведем по одному семафору ci на взаимодействующий процесс, для того чтобы обеспечивать блокирование процесса при попытке чтения из пустого буфера или при попытке записи в переполненный буфер. Посмотрим, как такой механизм будет работать. Начнем с процесса, желающего получить сообщение.

Процесс-получатель с номером i прежде всего выполняет операцию P(mutex), получая в монопольное владение разделяемую память. После чего он проверяет, есть ли в буфере сообщения. Если нет, то он заносит себя в список процессов, ожидающих сообщения, выполняет V(mutex) и P(ci). Если сообщение в буфере есть, то он читает его, изменяет счетчики буфера и проверяет, есть ли процессы в списке процессов, жаждущих записи. Если таких процессов нет, то выполняется V(mutex), и процесс-получатель выходит из критической области. Если такой процесс есть (с номером j ), то он удаляется из этого списка, выполняется V для его семафораcj, и мы выходим из критического района. Проснувшийся процесс начинает выполняться в критическом районе, так как mutex у нас имеет значение 0 и никто более не может попасть в критический район. При выходе из критического района именно разбуженный процесс произведет вызов V(mutex).

Как строится работа процесса-отправителя с номером i? Процесс, посылающий сообщение, тоже ждет, пока он не сможет иметь монополию на использование разделяемой памяти, выполнив операцию P(mutex). Далее он проверяет, есть ли место в буфере, и если да, то помещает сообщение в буфер, изменяет счетчики и смотрит, есть ли процессы, ожидающие сообщения. Если нет, выполняет V(mutex) и выходит из критической области, если есть, «будит» один из них (с номером j ), вызывая V(cj), с одновременным удалением этого процесса из списка процессов, ожидающих сообщений, и выходит из критического региона без вызова V(mutex), предоставляя тем самым возможность разбуженному процессу прочитать сообщение. Если места в буфере нет, то процесс-отправитель заносит себя в очередь процессов, ожидающих возможности записи, и вызывает V(mutex) и P(ci).

Нам достаточно показать, что *с помощью мониторов можно реализовать семафоры*, так как получать из семафоров сообщения мы уже умеем.

Самый простой способ такой реализации выглядит следующим образом. Заведем внутри монитора переменную-счетчик, связанный с эмулируемым семафором список блокируемых процессов и по одной условной переменной на каждый процесс. При выполнении операции P над семафором вызывающий процесс проверяет значение счетчика. Если оно больше нуля, уменьшает его на 1 и выходит из монитора. Если оно равно 0, процесс добавляет себя в очередь процессов, ожидающих события, и выполняет операцию wait над своей условной переменной. При выполнении операции V над семафором процесс увеличивает значение счетчика, проверяет, есть ли процессы, ожидающие этого события, и если есть, удаляет один из них из списка и выполняет операцию signal для условной переменной, соответствующей процессу.

Покажем, наконец, как *реализовать семафоры с помощью очередей сообщений*. Для этого воспользуемся более хитрой конструкцией, введя новый синхронизирующий процесс. Этот процесс имеет счетчик и очередь для процессов, ожидающих включения семафора. Для того чтобы выполнить операции P и V, процессы посылают синхронизирующему процессу сообщения, в которых указывают свои потребности, после чего ожидают получения подтверждения от синхронизирующего процесса.

После получения сообщения синхронизирующий процесс проверяет значение счетчика, чтобы выяснить, можно ли совершить требуемую операцию. Операция V всегда может быть выполнена, в то время как операция P может потребовать блокирования процесса. Если операция может быть совершена, то она выполняется, и синхронизирующий процесс посылает подтверждающее сообщение. Если процесс должен быть блокирован, то его идентификатор заносится в очередь блокированных процессов, и подтверждение не посылается. Позднее, когда какой-либо из других процессов выполнит операцию V, один из блокированных процессов удаляется из очереди ожидания и получает соответствующее подтверждение.

Поскольку мы показали ранее, как из семафоров построить мониторы, мы доказали эквивалентность мониторов, семафоров и сообщений.

# Управление памятью

Главная задача компьютерной системы – выполнять программы. Программы вместе с данными, к которым они имеют доступ, в процессе выполнения должны (по крайней мере частично) находиться в оперативной памяти. Операционной системе приходится решать задачу распределения памяти между пользовательскими процессами и компонентами ОС. Эта деятельность называется управлением памятью. Таким образом, память (storage, memory) является важнейшим ресурсом, требующим тщательного управления. В недавнем прошлом память была самым дорогим ресурсом.

Часть ОС, которая отвечает за управление памятью, называется ***менеджером памяти***.

## Организация памяти компьютера

**Физическая организация памяти компьютера**

Запоминающие устройства компьютера разделяют, как минимум, на два уровня: основную (главную, оперативную, физическую) и вторичную (внешнюю) память.

Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно основная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. В отличие от оперативной памяти, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

Эту схему можно дополнить еще несколькими промежуточными уровнями, Разновидности памяти могут быть объединены в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости (рис 3.1).



Рисунок 3.1 – Иерархия памяти

Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную информацию на i-м уровне, он начинает искать ее на следующих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

**Локальность**

Оказывается, при таком способе организации по мере снижения скорости доступа к уровню памяти снижается также и частота обращений к нему.

Ключевую роль здесь играет свойство реальных программ, в течение ограниченного отрезка времени способных работать с небольшим набором адресов памяти. Это эмпирически наблюдаемое свойство известно как принцип локальности или локализации обращений.

Свойство локальности (соседние в пространстве и времени объекты характеризуются похожими свойствами) присуще не только функционированию ОС, но и природе вообще. В случае ОС свойство локальности объяснимо, если учесть, как пишутся программы и как хранятся данные, то есть обычно в течение какого-то отрезка времени ограниченный фрагмент кода работает с ограниченным набором данных. Эту часть кода и данных удается разместить в памяти с быстрым доступом. В результате реальное время доступа к памяти определяется временем доступа к верхним уровням, что и обусловливает эффективность использования иерархической схемы. Надо сказать, что описываемая организация вычислительной системы во многом имитирует деятельность человеческого мозга при переработке информации. Действительно, решая конкретную проблему, человек работает с небольшим объемом информации, храня не относящиеся к делу сведения в своей памяти или во внешней памяти (например, в книгах).

Кэш процессора обычно является частью аппаратуры, поэтому менеджер памяти ОС занимается распределением информации главным образом в основной и внешней памяти компьютера. В некоторых схемах потоки между оперативной и внешней памятью регулируются программистом (см. например, далее оверлейные структуры ), однако это связано с затратами времени программиста, так что подобную деятельность стараются возложить на ОС.

Адреса в основной памяти, характеризующие реальное расположение данных в физической памяти, называются физическими адресами. Набор физических адресов, с которым работает программа, называют физическим адресным пространством.

**Логическая память**

Аппаратная организация памяти в виде линейного набора ячеек не соответствует представлениям программиста о том, как организовано хранение программ и данных. Большинство программ представляет собой набор модулей, созданных независимо друг от друга. Иногда все модули, входящие в состав процесса, располагаются в памяти один за другим, образуя линейное пространство адресов. Однако чаще модули помещаются в разные области памяти и используются по-разному.

Схема управления памятью, поддерживающая этот взгляд пользователя на то, как хранятся программы и данные, называется сегментацией. Сегмент – область памяти определенного назначения, внутри которой поддерживается линейная адресация. Сегменты содержат процедуры, массивы, стек или скалярные величины, но обычно не содержат информацию смешанного типа.

По-видимому, вначале сегменты памяти появились в связи с необходимостью обобществления процессами фрагментов программного кода (текстовый редактор, тригонометрические библиотеки и т. д.), без чего каждый процесс должен был хранить в своем адресном пространстве дублирующую информацию. Эти отдельные участки памяти, хранящие информацию, которую система отображает в память нескольких процессов, получили название сегментов. Память, таким образом, перестала быть линейной и превратилась в двумерную. Адрес состоит из двух компонентов: номер сегмента, смещение внутри сегмента. Далее оказалось удобным размещать в разных сегментах различные компоненты процесса (код программы, данные, стек и т. д.). Попутно выяснилось, что можно контролировать характер работы с конкретным сегментом, приписав ему атрибуты, например права доступа или типы операций, которые разрешается производить с данными, хранящимися в сегменте.

Большинство современных ОС поддерживают сегментную организацию памяти. В некоторых архитектурах (Intel, например) сегментация поддерживается оборудованием.

Адреса, к которым обращается процесс, таким образом, отличаются от адресов, реально существующих в оперативной памяти. В каждом конкретном случае 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). Подобный адрес, сгенерированный программой, обычно называют логическим (в системах с виртуальной памятью он часто называется виртуальным) адресом. Совокупность всех логических адресов называется логическим (виртуальным) адресным пространством.

**Связывание адресов**

Итак, логические и физические адресные пространства ни по организации, ни по размеру не соответствуют друг другу. Максимальный размер логического адресного пространства обычно определяется разрядностью процессора (например, 232) и в современных системах значительно превышает размер физического адресного пространства. Следовательно, процессор и ОС должны быть способны отобразить ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти. Такое отображение адресов называют трансляцией (привязкой) адреса или связыванием адресов (см. рис. 3.2).

Связывание логического адреса, порожденного оператором программы, сфизическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения. Таким образом, привязка инструкций и данных к памяти в принципе может быть сделана на следующих шагах

* Этап компиляции (Compiletime). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда непосредственно генерируются физические адреса. При изменении стартового адреса программы необходимо перекомпилировать ее код. В качестве примера можно привести .com программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.
* Этап загрузки (Loadtime). Если информация о размещении программы на стадии компиляции отсутствует, 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 (Executiontime). Если процесс может быть перемещен во время выполнения из одной области памяти в другую, связывание откладывается до стадии выполнения. Здесь желательно наличие специализированного оборудования, например регистров перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Большинство современных ОС осуществляет трансляцию адресов на этапе выполнения, используя для этого специальный аппаратный механизм.

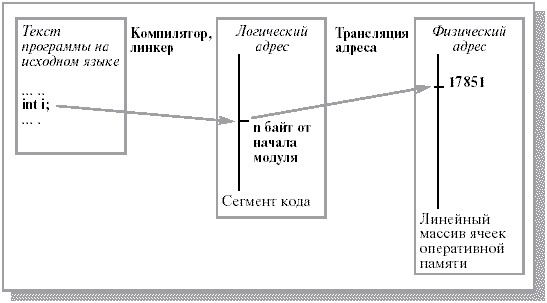


Рисунок 3.2– Формирование логического адреса и связывание логического адреса с физическим

**Функции системы управления памятью**

Чтобы обеспечить эффективный контроль использования памяти, ОС должна выполнять следующие функции:

* отображение адресного пространства процесса на конкретные области физической памяти;
* распределение памяти между конкурирующими процессами;
* контроль доступа к адресным пространствам процессов;
* выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;
* учет свободной и занятой памяти.

**Простейшие схемы управления памятью**

Первые ОС применяли очень простые методы управления памятью. Вначале каждый процесс пользователя должен был полностью поместиться в основной памяти, занимать непрерывную область памяти, а система принимала к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещались в основной памяти. Затем появился "простой свопинг" (система по-прежнему размещает каждый процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю и заменяет его в основной памяти образом другого процесса). Такого рода схемы имеют не только историческую ценность. В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для встроенных (embedded) компьютеров.

***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. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического адресного пространства. Связывание логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда – на этапе компиляции.

Каждый раздел может иметь свою очередь процессов, а может существовать и глобальная очередь для всех разделов(см. [рис. 8.4](http://www.intuit.ru/studies/courses/2192/31/lecture/982?page=2#image.8.4)).

Эта схема была реализована в IBM OS/360 (MFT), DEC RSX-11 и ряде других систем.

Подсистема управления памятью оценивает размер поступившего процесса, выбирает подходящий для него раздел, осуществляет загрузку процесса в этот раздел и настройку адресов.

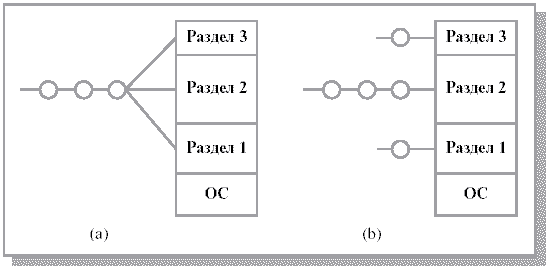


Рисунок 3.3 – Схема с фиксированными разделами: (a) – с общей очередью процессов, (b) – с отдельными очередями процессов

Очевидный недостаток этой схемы – число одновременно выполняемых процессов ограничено числом разделов.

Другим существенным недостатком является то, что предлагаемая схема сильно страдает от внутренней фрагментации – потери части памяти, выделенной процессу, но не используемой им. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ.

***Оверлейная структура***

Так как размер логического 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.

Потребность в таком способе загрузки появляется, если логическое адресное пространство системы мало, например 1 Мбайт (MS-DOS) или даже всего 64 Кбайта (PDP-11), а программа относительно велика. На 32-разрядных системах, где виртуальное адресное пространство измеряется гигабайтами, проблемы с нехваткой памяти решаются другими способами (см. раздел «Виртуальная память»).

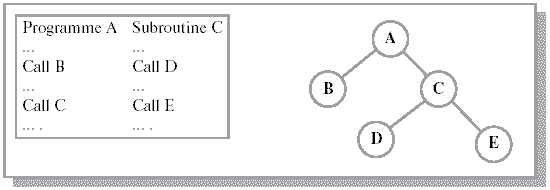


Рисунок 3.4 – Организация структуры с перекрытием. Можно поочередно загружать в память ветви A-B, A-C-D и A-C-E программы

Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для описания оверлейной структуры обычно используется специальный несложный язык (overlaydescriptionlanguage). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением .odl), описывающим дерево вызовов внутри программы. Для примера, приведенного на рис. 3.4, текст этого файла может выглядеть так:

A-(B,C)

C-(D,E)

Синтаксис подобного файла может распознаваться загрузчиком. Привязка к физической памяти происходит в момент очередной загрузки одной из ветвей программы.

Оверлеи могут быть полностью реализованы на пользовательском уровне в системах с простой файловой структурой. ОС при этом лишь делает несколько больше операций ввода-вывода. Типовое решение – порождение линкером специальных команд, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

Тщательное проектирование оверлейной структуры отнимает много времени и требует знания устройства программы, ее кода, данных и языка описания оверлейной структуры. По этой причине применение оверлеев ограничено компьютерами с небольшим логическим адресным пространством. Как мы увидим в дальнейшем, проблема оверлейных сегментов, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Заметим, что возможность организации структур с перекрытиями во многом обусловлена свойством локальности, которое позволяет хранить в памяти только ту информацию, которая необходима в конкретный момент вычислений.

***Динамическое распределение. Свопинг***

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. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (paging) и будет рассмотрена ниже.

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. Очевидно, что свопинг увеличивает время переключения контекста. Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, то есть быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

***Схема с переменными разделами***

В принципе, система свопинга может базироваться на фиксированных разделах. Более эффективной, однако, представляется схема динамического распределения или схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком помещаются в памяти, то есть в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера (рис. 3.5). Смежные свободные участки могут быть объединены.

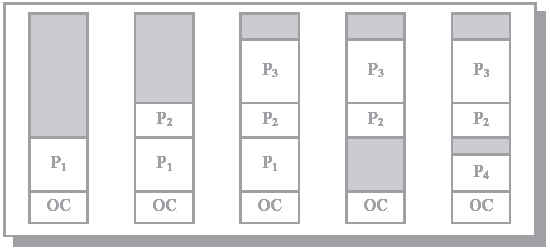


Рисунок 3.5 – Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память)

В какой раздел помещать процесс? Наиболее распространены три стратегии.

* Стратегия первого подходящего (Firstfit). Процесс помещается в первый подходящий по размеру раздел.
* Стратегия наиболее подходящего (Bestfit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места.
* Стратегия наименее подходящего (Worstfit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например для размещения файлов на диске.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может осуществляться на этапах загрузки и выполнения.

Этот метод более гибок по сравнению с методом фиксированных разделов, однако ему присуща внешняя фрагментация – наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор стратегии размещения процесса между первым подходящим и наиболее подходящим слабо влияет на величину фрагментации. Любопытно, что метод наиболее подходящего может оказаться наихудшим, так как он оставляет множество мелких незанятых блоков.

Статистический анализ показывает, что пропадает в среднем 1/3 памяти! Это известное правило 50% (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены).

Одно из решений проблемы внешней фрагментации – организовать 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.

***Страничная память***

Описанные выше схемы недостаточно эффективно используют память, поэтому в современных схемах управления памятью не принято размещать процесс в оперативной памяти одним непрерывным блоком.

В самом простом и наиболее распространенном случае страничной организации памяти (или paging) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы (page), а соответствующие единицы в физической памяти называют физическими страницами или страничными кадрами (pageframes). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое числостраниц, ограничены частью последней страницы процесса.

Логический адрес в страничной системе – упорядоченная пара (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиениеадресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с точки зрения программиста адресное пространство процесса остается линейным.

Описываемая схема позволяет загрузить процесс, даже если нет непрерывной области кадров, достаточной для размещения процесса целиком. Но одного базового регистра для осуществления трансляции адреса в данной схеме недостаточно. Система отображения логических адресов в физические сводится к системе отображения логических страниц в физические и представляет собой таблицу страниц, которая хранится в оперативной памяти. Иногда говорят, что таблицастраниц – это кусочно-линейная функция отображения, заданная в табличном виде.

Интерпретация логического адреса показана на рис. 3.6. Если выполняемый процесс обращается к логическому адресу v = (p,d), механизм отображения ищет номер страницы p в таблице страниц и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d.

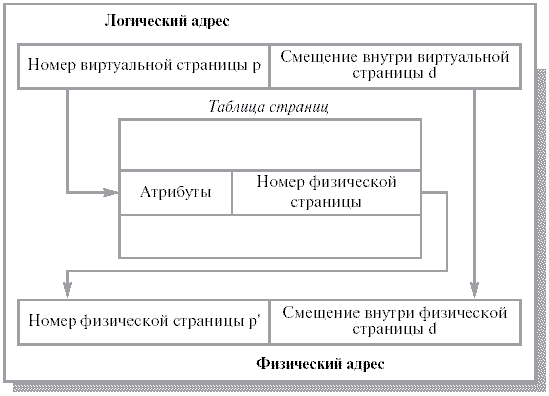


Рисунок 3.6 – Связь логического и физического адресов при страничной организации памяти

Таблица страниц (pagetable) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

Отметим еще раз различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя, его память – единое непрерывное пространство, содержащее только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. Заметим, что процессу пользователя чужая память недоступна. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение адресов должно быть осуществлено корректно даже в сложных случаях и обычно реализуется аппаратно. Для ссылки на таблицу процессов используется специальный регистр. При переключении процессов необходимо найти таблицу страниц нового процесса, указатель на которую входит в контекст процесса.

**Сегментная и сегментно-страничная организация памяти**

Существуют две другие схемы организации управления памятью: сегментная и сегментно-страничная. Сегменты, в отличие от страниц, могут иметь переменный размер. Идея сегментации изложена во введении. При сегментной организации виртуальный адрес является двумерным как для программиста, так и для операционной системы, и состоит из двух полей – номера сегмента и смещения внутри сегмента. Подчеркнем, *что в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набор сегментов переменного размера (данные, код, стек...).*

Программисты, пишущие на языках низкого уровня, должны иметь представление о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере). Логическое адресное пространство – набор сегментов. Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия). В отличие от страничной схемы, где пользователь задает только один адрес, который разбивается на номер страницы и смещение прозрачным для программиста образом, в сегментной схеме пользователь специфицирует каждый адрес двумя величинами: именем сегмента и смещением.

Каждый сегмент – линейная последовательность адресов, начинающаяся с 0. Максимальный размер сегмента определяется разрядностью процессора (при 32-разрядной адресации это 232байт или 4 Гбайт). Размер сегмента может меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента обычно содержится и длинасегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает исключительная ситуация.

Логический адрес – упорядоченная пара v=(s,d), номер сегмента и смещение внутри сегмента.

В системах, где сегменты поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов сегментов, а программа обращается к этим дескрипторам по номерам-селекторам. При этом в контекст каждого процесса входит набор сегментных регистров, содержащих селекторы текущих сегментов кода, стека, данных и т. д. и определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа. Аппаратная поддержка сегментов распространена мало (главным образом на процессорах Intel). В большинстве ОС сегментация реализуется на уровне, не зависящем от аппаратуры.

Хранить в памяти сегменты большого размера целиком так же неудобно, как и хранить процесс непрерывным блоком. Напрашивается идея разбиения сегментов на страницы. При сегментно-страничной организации памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае логический адрес состоит из трех полей: номера сегмента   логической памяти, номера страницы внутри сегмента и смещения внутри страницы (рис.3.7). Соответственно, используются две таблицы отображения – таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

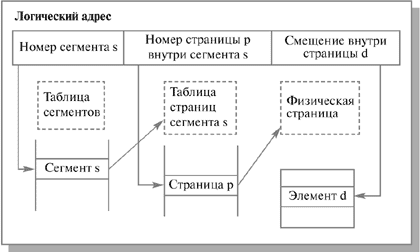


Рисунок 3.7 – Упрощенная схема формирования физического адреса при сегментно-страничной организации памяти

Сегментно-страничная и сегментная организация памяти позволяет легко организовать совместное использование одних и тех же данных и программного кода разными задачами. Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок физической памяти, где размещается разделяемый фрагмент кода или данных.

## Виртуальная память. Средства поддержки виртуальной памяти.

Разработчикам программного обеспечения часто приходится решать проблему размещения в памяти больших программ, размер которых превышает объем доступной оперативной памяти. Один из вариантов решения данной проблемы – организация структур с перекрытием – рассмотрен в предыдущей лекции. При этом предполагалось активное участие программиста в процессе формирования перекрывающихся частей программы. Развитие архитектуры компьютеров и расширение возможностей операционной системы по управлению памятью позволило переложить решение этой задачи на компьютер. Одним из главных достижений стало появление виртуальной памяти (virtualmemory). Впервые она была реализована в 1959 г. на компьютере «Атлас», разработанном в Манчестерском университете.

Суть концепции виртуальной памяти заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах виртуальной памяти у процесса создается иллюзия того, что вся необходимая ему информация имеется в основной памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей, например страниц. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И, наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный бит присутствия, входящий в состав атрибутов страницы в таблице страниц .

Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти. Принцип локальности обеспечивает этой схеме нужную эффективность.

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

* Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о 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 (характерный размер для 32-разрядных архитектур 232 = 4 Гбайт) адресуемой пользовательской памяти (логическое адресное пространство) при наличии основной памяти существенно меньших размеров (физическое адресное пространство) – очень важный аспект.

Но введение виртуальной памяти позволяет решать другую, не менее важную задачу – обеспечение контроля доступа к отдельным сегментам памяти и, в частности, защиту пользовательских программ друг от друга и защиту ОС от пользовательских программ. Каждый процесс работает со своими виртуальными адресами, трансляцию которых в физические выполняет аппаратура компьютера. Таким образом, пользовательский процесс лишен возможности напрямую обратиться к страницам основной памяти, занятым информацией, относящейся к другим процессам.

Например, 16-разрядный компьютер PDP-11/70 с 64 Кбайт логической памяти мог иметь до 2 Мбайт оперативной памяти. Операционная система этого компьютера тем не менее поддерживала виртуальную память, которая обеспечивала защиту и перераспределение основной памяти между пользовательскими процессами.

Напомним, что в системах с виртуальной памятью те адреса, которые генерирует программа (логические адреса), называются виртуальными, и они формируют виртуальное адресное пространство. Термин " виртуальная память " означает, что программист имеет дело с памятью, отличной от реальной, размер которой потенциально больше, чем размер оперативной памяти.

Хотя известны и чисто программные реализации виртуальной памяти, это направление получило наиболее широкое развитие после соответствующей аппаратной поддержки.

Следует отметить, что оборудование компьютера принимает участие в трансляции адреса практически во всех схемах управления памятью. Но в случае виртуальной памяти это становится более сложным вследствие разрывности отображения и многомерности логического адресного пространства. Может быть, наиболее существенным вкладом аппаратуры в реализацию описываемой схемы является автоматическая генерация исключительных ситуаций при отсутствии в памяти нужных страниц (pagefault).

Любая из трех ранее рассмотренных схем управления памятью – страничной, сегментной и сегментно-страничной – пригодна для организации виртуальной памяти. Чаще всего используется cегментно-страничная модель, которая является синтезом страничной модели и идеи сегментации. Причем для тех архитектур, в которых сегменты не поддерживаются аппаратно, их реализация – задача архитектурно-независимого компонента менеджера памяти.

Сегментная организация в чистом виде встречается редко.

**Архитектурные средства поддержки виртуальной памяти**

Очевидно, что невозможно создать полностью машинно-независимый компонент управления виртуальной памятью. С другой стороны, имеются существенные части программного обеспечения, связанного с управлением виртуальной памятью, для которых детали аппаратной реализации совершенно не важны. Одним из достижений современных ОС является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части. Коротко рассмотрим, что и каким образом входит в аппаратно-зависимую часть подсистемы управления виртуальной памятью. Компоненты аппаратно-независимой подсистемы будут рассмотрены в следующей лекции.

В самом распространенном случае необходимо отобразить большое виртуальное адресное пространство в физическое адресное пространство существенно меньшего размера. Пользовательский процесс или ОС должны иметь возможность осуществить запись по виртуальному адресу, а задача ОС – сделать так, чтобы записанная информация оказалась в физической памяти (впоследствии при нехватке оперативной памяти она может быть вытеснена во внешнюю память). В случае виртуальной памяти система отображения адресных пространств помимо трансляции адресов должна предусматривать ведение таблиц, показывающих, какие области виртуальной памяти в данный момент находятся в физической памяти и где именно размещаются.

***Страничная виртуальная память***

Как и в случае простой страничной организации, страничная виртуальная память и физическая память представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальные адреса делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (pageframes), а в целом система поддержки страничной виртуальной памяти называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы виртуальный адрес преобразуется в упорядоченную пару (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

При отсутствии страницы в памяти в процессе выполнения команды возникает исключительная ситуация, называемая страничное нарушение (pagefault) или страничный отказ. Обработка страничного нарушения заключается в том, что выполнение команды прерывается, затребованная страница подкачивается из конкретного места вторичной памяти в свободный страничный кадр физической памяти и попытка выполнения команды повторяется. При отсутствии свободных страничных кадров на диск выгружается редко используемая страница. Проблемы замещения страниц и обработки страничных нарушений рассматриваются в следующей лекции.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающую его состояние.

В большинстве современных компьютеров со страничной организацией в основной памяти хранится лишь часть таблицы страниц, а быстрота доступа к элементам таблицы текущей виртуальной памяти достигается, как будет показано ниже, за счет использования сверхбыстродействующей памяти, размещенной в кэше процессора.

***Сегментно-страничная организации виртуальной памяти***

Как и в случае простой сегментации, в схемах виртуальной памяти сегмент – это линейная последовательность адресов, начинающаяся с 0. При организации виртуальной памяти размер сегмента может быть велик, например, может превышать размер оперативной памяти. Повторяя все ранее приведенные рассуждения о размещении в памяти больших программ, приходим к разбиению сегментов на страницы и необходимости поддержки своей таблицы страниц для каждого сегмента.

На практике, однако, появления в системе большого количества таблиц страниц стараются избежать, организуя неперекрывающиеся сегменты в одном виртуальном пространстве, для описания которого хватает одной таблицы страниц. Таким образом, одна таблица страниц отводится для всего процесса. Например, в популярных ОС Linux и Windows 2000 все сегменты процесса, а также область памяти ядра ограничены виртуальным адресным пространством объемом 4 Гбайт. При этом ядро ОС располагается по фиксированным виртуальным адресам вне зависимости от выполняемого процесса.

***Структура таблицы страниц***

Организация таблицы страниц – один из ключевых элементов отображения адресов в страничной и сегментно-страничной схемах. Рассмотрим структуру таблицы страниц для случая страничной организации более подробно.

Итак, виртуальный адрес состоит из виртуального номера страницы и смещения. Номер записи в таблице страниц соответствует номеру виртуальной страницы. Размер записи колеблется от системы к системе, но чаще всего он составляет 32 бита. Из этой записи в таблице страниц находится номер кадра для данной виртуальной страницы, затем прибавляется смещение и формируется физический адрес. Помимо этого запись в таблице страниц содержит информацию об атрибутах страницы. Это биты присутствия и защиты (например, 0 – read/write, 1 – readonly...). Также могут быть указаны: бит модификации, который устанавливается, если содержимое страницы модифицировано, и позволяет контролировать необходимость перезаписи страницы на диск; бит ссылки, который помогает выделить малоиспользуемые страницы; бит, разрешающий кэширование, и другие управляющие биты. Заметим, что адреса страниц на диске не являются частью таблицы страниц.

Основную проблему для эффективной реализации таблицы страниц создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора. Самыми распространенными на сегодня являются 32-разрядные процессоры, позволяющие создавать виртуальные адресные пространства размером 4 Гбайт (для 64-разрядных компьютеров эта величина равна 264 байт). Кроме того, существует проблема скорости отображения, которая решается за счет использования так называемой ассоциативной памяти (см. следующий раздел).

Подсчитаем примерный размер таблицы страниц. В 32-битном адресном пространстве при размере страницы 4 Кбайт (Intel) получаем 232/212=220, то есть приблизительно миллион страниц, а в 64-битном и того более. Таким образом, таблица должна иметь примерно миллион строк (entry), причем запись в строке состоит из нескольких байтов. Заметим, что каждый процесс нуждается в своей таблице страниц (а в случае сегментно-страничной схемы желательно иметь по одной таблице страниц на каждый сегмент).

Понятно, что количество памяти, отводимое таблицам страниц, не может быть так велико. Для того чтобы избежать размещения в памяти огромной таблицы, ее разбивают на ряд фрагментов. В оперативной памяти хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты таблицы страниц. В силу свойства локальности число таких фрагментов относительно невелико. Выполнить разбиение таблицы страниц на части можно по-разному. Наиболее распространенный способ разбиения – организация так называемой многоуровневой таблицы страниц. Для примера рассмотрим двухуровневую таблицу с размером страниц 4 Кбайт, реализованную в 32-разрядной архитектуре Intel.

Таблица, состоящая из 220 строк, разбивается на 210таблиц второго уровня по 210 строк. Эти таблицы второго уровня объединены в общую структуру при помощи одной таблицы первого уровня, состоящей из 210 строк. 32-разрядный адрес делится на 10-разрядное поле p1, 10-разрядное поле p2 и 12-разрядное смещение d. Поле p1 указывает на нужную строку в таблице первого уровня, поле p2 – второго, а поле d локализует нужный байт внутри указанного страничного кадра (см. [рис. 9.1](http://www.intuit.ru/studies/courses/2192/31/lecture/984?page=2#image.9.1)).

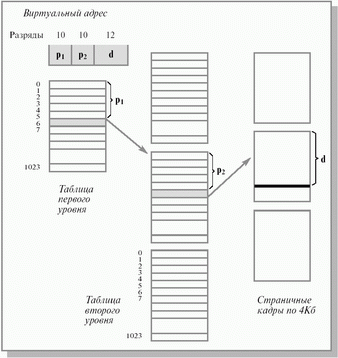


Рисунок 3.8 – Пример двухуровневой таблицы страниц

При помощи всего лишь одной таблицы второго уровня можно охватить 4 Мбайт (4 Кбайт x 1024) оперативной памяти. Таким образом, для размещения процесса с большим объемом занимаемой памяти достаточно иметь в оперативной памяти одну таблицу первого уровня и несколько таблиц второго уровня. Очевидно, что суммарное количество строк в этих таблицах много меньше 220. Такой подход естественным образом обобщается на три и более уровней таблицы.

Наличие нескольких уровней, естественно, снижает производительность менеджера памяти. Несмотря на то что размеры таблиц на каждом уровне подобраны так, чтобы таблица помещалась целиком внутри одной страницы, обращение к каждому уровню – это отдельное обращение к памяти. Таким образом, трансляция адреса может потребовать нескольких обращений к памяти.

***Ассоциативная память***

Поиск номера кадра, соответствующего нужной странице, в многоуровневой таблице страниц требует нескольких обращений к основной памяти, поэтому занимает много времени. В некоторых случаях такая задержка недопустима. Проблема ускорения поиска решается на уровне архитектуры компьютера.

В соответствии со свойством локальности большинство программ в течение некоторого промежутка времени обращаются к небольшому количеству страниц, поэтому активно используется только небольшая часть таблицы страниц.

Естественное решение проблемы ускорения – снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц, то есть иметь небольшую, быструю кэш-память, хранящую необходимую на данный момент часть таблицы страниц. Это устройство называется ассоциативной памятью, иногда также употребляют термин буфер поиска трансляции (translation lookaside buffer – TLB).

Одна запись таблицы в ассоциативной памяти (один вход) содержит информацию об одной виртуальной странице: ее атрибуты и кадр, в котором она находится. Эти поля в точности соответствуют полям в таблице страниц.

Так как ассоциативная память содержит только некоторые из записей таблицы страниц, каждая запись в TLB должна включать поле с номером виртуальной страницы. Память называется ассоциативной, потому что в ней происходит одновременное сравнение номера отображаемой виртуальной страницы с соответствующим полем во всех строках этой небольшой таблицы. Поэтому данный вид памяти достаточно дорого стоит. В строке, поле виртуальной страницы которой совпало с искомым значением, находится номер страничного кадра. Обычное число записей в TLB от 8 до 4096. Рост количества записей в ассоциативной памяти должен осуществляться с учетом таких факторов, как размер кэша основной памяти и количества обращений к памяти при выполнении одной команды.

Рассмотрим функционирование менеджера памяти при наличии ассоциативной памяти.

В начале информация об отображении виртуальной страницы в физическую отыскивается в ассоциативной памяти. Если нужная запись найдена – все нормально, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется.

Если нужная запись в ассоциативной памяти отсутствует, отображение осуществляется через таблицу страниц. Происходит замена одной из записей в ассоциативной памяти найденной записью из таблицы страниц. Здесь мы сталкиваемся с традиционной для любого кэша проблемой замещения (а именно какую из записей в кэше необходимо изменить). Конструкция ассоциативной памяти должна организовывать записи таким образом, чтобы можно было принять решение о том, какая из старых записей должна быть удалена при внесении новых.

Число удачных поисков номера страницы в ассоциативной памяти по отношению к общему числу поисков называется hit (совпадение) ratio (пропорция, отношение). Иногда также используется термин "процент попаданий в кэш". Таким образом, hitratio – часть ссылок, которая может быть сделана с использованием ассоциативной памяти. Обращение к одним и тем же страницам повышает hitratio. Чем больше hitratio, тем меньше среднее время доступа к данным, находящимся в оперативной памяти.

Предположим, например, что для определения адреса в случае кэш-промаха через таблицу страниц необходимо 100 нс, а для определения адреса в случае кэш-попадания через ассоциативную память – 20 нс. С 90% hitratio среднее время определения адреса – 0,9x20+0,1x100 = 28 нс.

Вполне приемлемая производительность современных ОС доказывает эффективность использования ассоциативной памяти. Высокое значение вероятности нахождения данных в ассоциативной памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

Необходимо обратить внимание на следующий факт. При переключении контекста процессов нужно добиться того, чтобы новый процесс "не видел" в ассоциативной памяти информацию, относящуюся к предыдущему процессу, например очищать ее. Таким образом, использование ассоциативной памяти увеличивает время переключения контекста.

Рассмотренная двухуровневая ( ассоциативная память + таблица страниц ) схема преобразования адреса является ярким примером иерархии памяти, основанной на использовании принципа локальности, о чем говорилось во введении к предыдущей лекции.

***Инвертированная таблица страниц***

Несмотря на многоуровневую организацию, хранение нескольких таблиц страниц большого размера по-прежнему представляют собой проблему. Ее значение особенно актуально для 64-разрядных архитектур, где число виртуальных страниц очень велико. Вариантом решения является применение инвертированной таблицы страниц (invertedpagetable). Этот подход применяется на машинах PowerPC, некоторых рабочих станциях Hewlett-Packard, IBM RT, IBM AS/400 и ряде других.

В этой таблице содержится по одной записи на каждый страничный кадр физической памяти. Существенно, что достаточно одной таблицы для всех процессов. Таким образом, для хранения функции отображения требуется фиксированная часть основной памяти, независимо от разрядности архитектуры, размера и количества процессов.

Несмотря на экономию оперативной памяти, применение инвертированной таблицы имеет существенный минус – записи в ней (как и в ассоциативной памяти ) не отсортированы по возрастанию номеров виртуальных страниц, что усложняет трансляцию адреса. Один из способов решения данной проблемы – использование хеш-таблицы виртуальных адресов. При этом часть виртуального адреса, представляющая собой номер страницы, отображается в хеш-таблицу с использованием функции хеширования. Каждой странице физической памяти здесь соответствует одна запись в хеш-таблице и инвертированной таблице страниц. Виртуальные адреса, имеющие одно значение хеш-функции, сцепляются друг с другом. Обычно длина цепочки не превышает двух записей.

**Размер страницы**

Разработчики ОС для существующих машин редко имеют возможность влиять на размер страницы. Однако для вновь создаваемых компьютеров решение относительно оптимального размера страницы является актуальным. Как и следовало ожидать, нет одного наилучшего размера. Скорее есть набор факторов, влияющих на размер. Обычно размер страницы – это степень двойки от 29 до 214 байт.

Чем больше размер страницы, тем меньше будет размер структур данных, обслуживающих преобразование адресов, но тем больше будут потери, связанные с тем, что память можно выделять только постранично.

Как следует выбирать размер страницы? Во-первых, нужно учитывать размер таблицы страниц, здесь желателен большой размер страницы (страниц меньше, соответственно и таблица страниц меньше). С другой стороны, память лучше утилизируется с маленьким размером страницы. В среднем половина последней страницы процесса пропадает. Необходимо также учитывать объем ввода-вывода для взаимодействия с внешней памятью и другие факторы. Проблема не имеет идеального решения. Историческая тенденция состоит в увеличении размера страницы.

Как правило, размер страниц задается аппаратно, например в DEC PDP-11 – 8 Кбайт, в DEC VAX – 512 байт, в других архитектурах, таких как Motorola 68030, размер страниц может быть задан программно. Учитывая все обстоятельства, в ряде архитектур возникают множественные размеры страниц, например в Pentium размер страницы колеблется от 4 Кбайт до 8 Кбайт. Тем не менее большинство коммерческих ОС ввиду сложности перехода на множественный размер страниц поддерживают только один размер страниц.

## Стратегии управления страничной памятью

Программное обеспечение подсистемы управления памятью связано с реализацией следующих стратегий:

*Стратегия выборки (fetchpolicy)* – в какой момент следует переписать страницу из вторичной памяти в первичную. Существует два основных варианта выборки – по запросу и с упреждением. Алгоритм выборки по запросу вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой находится на диске. Его реализация заключается в загрузке страницы с диска в свободную физическую страницу и коррекции соответствующей записи таблицы страниц.

*Алгоритм выборки с упреждением* осуществляет опережающее чтение, то есть кроме страницы, вызвавшей исключительную ситуацию, в память также загружается несколько страниц, окружающих ее (обычно соседние страницы располагаются во внешней памяти последовательно и могут быть считаны за одно обращение к диску). Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникающих при работе со значительными объемами данных или кода; кроме того, оптимизируется работа с диском.

*Стратегия размещения (placementpolicy)* – в какой участок первичной памяти поместить поступающую страницу. В системах со страничной организацией все просто – в любой свободный страничный кадр. В случае систем с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением.

*Стратегия замещения (replacementpolicy)* – какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место в оперативной памяти. Разумная стратегия замещения, реализованная в соответствующем алгоритме замещения страниц, позволяет хранить в памяти самую необходимую информацию и тем самым снизить частоту страничных нарушений . Замещение должно происходить с учетом выделенного каждому процессу количества кадров. Кроме того, нужно решить, должна ли замещаемая страница принадлежать процессу, который инициировал замещение, или она должна быть выбрана среди всех кадров основной памяти.

**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 кода текстового процессора при редактировании файла. Заметим, что при замещении активных страниц все работает корректно, но pagefault происходит немедленно.

*Аномалия Билэди (Belady)*

На первый взгляд кажется очевидным, что чем больше в памяти страничных кадров, тем реже будут иметь место pagefaults. Удивительно, но это не всегда так. Как установил Билэди с коллегами, определенные последовательности обращений к страницам в действительности приводят к увеличению числа страничных нарушений при увеличении кадров, выделенных процессу. Это явление носит название "аномалии Билэди" или "аномалии FIFO ".

Система с тремя кадрами (9 faults) оказывается более производительной, чем с четырьмя кадрами (10 faults), для строки обращений к памяти 012301401234 при выборе стратегии FIFO (рис.3.9).

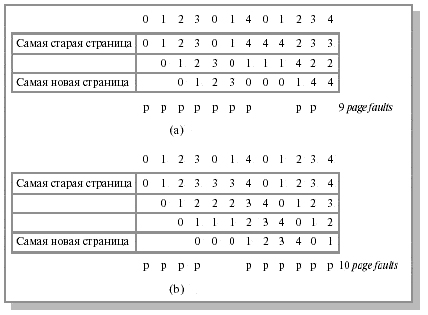


Рисунок 3.9 – Аномалия Билэди: (a) – FIFO с тремя страничными кадрами; (b) – FIFO с четырьмя страничными кадрами

Аномалию Билэди следует считать скорее курьезом, чем фактором, требующим серьезного отношения, который иллюстрирует сложность ОС, где интуитивный подход не всегда приемлем.

**Оптимальный алгоритм (OPT)**

Одним из последствий открытия аномалии Билэди стал поиск оптимального алгоритма, который при заданной строке обращений имел бы минимальную частоту pagefaults среди всех других алгоритмов. Такой алгоритм был найден. Он прост: замещай страницу, которая не будет использоваться в течение самого длительного периода времени.

Каждая страница должна быть помечена числом инструкций, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка. Выталкиваться должна страница, для которой это число наибольшее.

Этот алгоритм легко описать, но реализовать невозможно. ОС не знает, к какой странице будет следующее обращение. (Ранее такие проблемы возникали при планировании процессов – алгоритм SJF).

Зато мы можем сделать вывод, что для того, чтобы алгоритм замещения был максимально близок к идеальному алгоритму, система должна как можно точнее предсказывать обращения процессов к памяти. Данный алгоритм применяется для оценки качества реализуемых алгоритмов.

**Выталкивание дольше всего не использовавшейся страницы. Алгоритм LRU**

Одним из приближений к алгоритму OPT является алгоритм, исходящий из эвристического правила, что недавнее прошлое – хороший ориентир для прогнозирования ближайшего будущего.

Ключевое отличие между FIFO и оптимальным алгоритмом заключается в том, что один смотрит назад, а другой вперед. Если использовать прошлое для аппроксимации будущего, имеет смысл замещать страницу, которая не использовалась в течение самого долгого времени. Такой подход называется leastrecentlyused алгоритм ( LRU ). Работа алгоритма проиллюстрирована на рис.3.10. Сравнивая рис. 3.9 b и 3.10, можно увидеть, что использование LRU алгоритма позволяет сократить количество страничных нарушений.

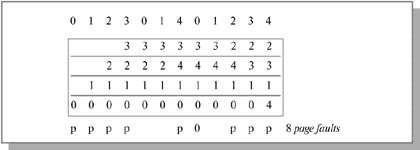


Рисунок 3.10 – Пример работы алгоритма LRU

LRU – хороший, но труднореализуемый алгоритм. Необходимо иметь связанный список всех страниц в памяти, в начале которого будут хранится недавно использованные страницы. Причем этот список должен обновляться при каждом обращении к памяти. Много времени нужно и на поиск страниц в таком списке.

В [3] рассмотрен вариант реализации алгоритма LRU со специальным 64-битным указателем, который автоматически увеличивается на единицу после выполнения каждой инструкции, а в таблице страниц имеется соответствующее поле, в которое заносится значение указателя при каждой ссылке на страницу. При возникновении pagefault выгружается страница с наименьшим значением этого поля.

Как оптимальный алгоритм, так и LRU не страдают от аномалии Билэди. Существует класс алгоритмов, для которых при одной и той же строке обращений множество страниц в памяти для n кадров всегда является подмножеством страниц для n+1 кадра. Эти алгоритмы не проявляют аномалии Билэди и называются стековыми (stack) алгоритмами.

**Выталкивание редко используемой страницы. Алгоритм NFU**

Поскольку большинство современных процессоров не предоставляют соответствующей аппаратной поддержки для реализации алгоритма LRU, хотелось бы иметь алгоритм, достаточно близкий к LRU, но не требующий специальной поддержки.

Программная реализация алгоритма, близкого к LRU, – алгоритм NFU (NotFrequentlyUsed). 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. Главный недостаток алгоритма NFU состоит в том, что он ничего не забывает. Например, страница, к которой очень часто обращались в течение некоторого времени, а потом обращаться перестали, все равно не будет удалена из памяти, потому что ее счетчик содержит большую величину. Например, в многопроходных компиляторах страницы, которые активно использовались во время первого прохода, могут надолго сохранить большие значения счетчика, мешая загрузке полезных в дальнейшем страниц.

К счастью, возможна небольшая модификация алгоритма, которая позволяет ему «забывать». Достаточно, чтобы при каждом прерывании по времени содержимое счетчика сдвигалось вправо на 1 бит, а уже затем производилось бы его увеличение для страниц с установленным флагом обращения.

Другим, уже более устойчивым недостатком алгоритма является длительность процесса сканирования таблиц страниц.

**Другие алгоритмы**

Для полноты картины можно упомянуть еще несколько алгоритмов.

Например, алгоритм Second-Chance – модификация алгоритма FIFO, которая позволяет избежать потери часто используемых страниц с помощью анализа флага обращений (бита ссылки) для самой старой страницы. Если флаг установлен, то страница, в отличие от алгоритма FIFO, не выталкивается, а ее флаг сбрасывается, и страница переносится в конец очереди. Если первоначально флаги обращений были установлены для всех страниц (на все страницы ссылались), алгоритм Second-Chance превращается в алгоритм FIFO. Данный алгоритм использовался в Multics и BSD Unix.

В компьютере Macintosh использован алгоритм NRU (NotRecently-Used), где страница-"жертва" выбирается на основе анализа битов модификации и ссылки. Интересные стратегии, основанные на буферизации страниц, реализованы в VAX/VMS и Mach.

Имеется также и много других алгоритмов замещения. Объем этого курса не позволяет рассмотреть их подробно.

**Управление количеством страниц, выделенных процессу. Модель рабочего множества**

В стратегиях замещения, рассмотренных выше, прослеживается предположение о том, что количество кадров, принадлежащих процессу, нельзя увеличить. Это приводит к необходимости выталкивания страницы. Рассмотрим более общий подход, базирующийся на концепции рабочего множества, сформулированной Деннингом.

Итак, что делать, если в распоряжении процесса имеется недостаточное число кадров? Нужно ли его приостановить с освобождением всех кадров? Что следует понимать под достаточным количеством кадров?

***Трешинг (Thrashing)***

Хотя теоретически возможно уменьшить число кадров процесса до минимума, существует какое-то число активно используемых страниц, без которого процесс часто генерирует pagefaults. Высокая частота страничных нарушений называется трешинг (thrashing, иногда употребляется русский термин "пробуксовка", рис. 3.11). Процесс находится в состоянии трешинга, если при его работе больше времени уходит на подкачку страниц, нежели на выполнение команд. Такого рода критическая ситуация возникает вне зависимости от конкретных алгоритмов замещения.

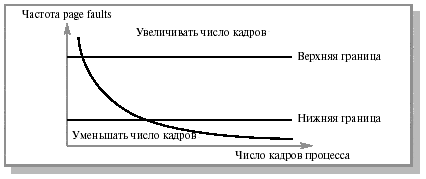


Рисунок 3.11 – Частота pagefaults в зависимости от количества кадров, выделенных процессу

Часто результатом трешинга является снижение производительности вычислительной системы. Один из нежелательных сценариев развития событий может выглядеть следующим образом. При глобальном алгоритме замещения процесс, которому не хватает кадров, начинает отбирать кадры у других процессов, которые в свою очередь начинают заниматься тем же. В результате все процессы попадают в очередь запросов к устройству вторичной памяти (находятся в состоянии ожидания), а очередь процессов в состоянии готовности пустеет. Загрузка процессора снижается. Операционная система реагирует на это увеличением степени мультипрограммирования, что приводит к еще большему трешингу и дальнейшему снижению загрузки процессора. Таким образом, пропускная способность системы падает из-за трешинга.

Эффект трешинга, возникающий при использовании глобальных алгоритмов, может быть ограничен за счет применения локальных алгоритмов замещения. При локальных алгоритмах замещения если даже один из процессов попал в трешинг, это не сказывается на других процессах. Однако он много времени проводит в очереди к устройству выгрузки, затрудняя подкачку страниц остальных процессов.

Критическая ситуация типа трешинга возникает вне зависимости от конкретных алгоритмов замещения. Единственным алгоритмом, теоретически гарантирующим отсутствие трешинга, является рассмотренный выше не реализуемый на практике оптимальный алгоритм.

Итак, трешинг – это высокая частота страничных нарушений. Hеобходимо ее контролировать. Когда она высока, процесс нуждается в кадрах. Можно, устанавливая желаемую частоту pagefaults, регулировать размер процесса, добавляя или отнимая у него кадры. Может оказаться целесообразным выгрузить процесс целиком. Освободившиеся кадры выделяются другим процессам с высокой частотой pagefaults.

Для предотвращения трешинга требуется выделять процессу столько кадров, сколько ему нужно. Hо как узнать, сколько ему нужно? Необходимо попытаться выяснить, как много кадров процесс реально использует. Для решения этой задачи Деннинг использовал модель рабочего множества, которая основана на применении принципа локальности.

***Модель рабочего множества***

Рассмотрим поведение реальных процессов.

Процессы начинают работать, не имея в памяти необходимых страниц. В результате при выполнении первой же машинной инструкции возникает pagefault, требующий подкачки порции кода. Следующий pagefault происходит при локализации глобальных переменных и еще один – при выделении памяти для стека. После того как процесс собрал большую часть необходимых ему страниц, pagefaults возникают редко.

Таким образом, существует набор страниц (P1, P2, ...Pn), активно использующихся вместе, который позволяет процессу в момент времени t в течение некоторого периода T производительно работать, избегая большого количества pagefaults. Этот набор страниц называется **рабочим множеством**W(t,T)( **workingset** ) процесса. Число страниц в рабочем множестве определяется параметромТ, является неубывающей функцией T и относительно невелико. Иногда T называют размером окна рабочего множества, через которое ведется наблюдение за процессом (рис. 3.12).

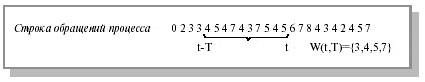


Рисунок 3.12 – Пример рабочего множества процесса

Легко написать тестовую программу, которая систематически работает с большим диапазоном адресов, но, к счастью, большинство реальных процессов не ведут себя подобным образом, а проявляют свойство локальности. В течение любой фазы вычислений процесс работает с небольшим количеством страниц.

Когда процесс выполняется, он двигается от одного рабочего множества к другому. Программа обычно состоит из нескольких рабочих множеств, которые могут перекрываться. Hапример, когда вызвана процедура, она определяет новое рабочее множество, состоящее из страниц, содержащих инструкции процедуры, ее локальные и глобальные переменные. После ее завершения процесс покидает это рабочее множество, но может вернуться к нему при новом вызове процедуры. Таким образом, рабочее множество определяется кодом и данными программы. Если процессу выделять меньше кадров, чем ему требуется для поддержки рабочего множества, он будет находиться в состоянии трешинга.

Принцип локальности ссылок препятствует частым изменениям рабочих наборов процессов. Формально это можно выразить следующим образом. Если в период времени (t-T, t) программа обращалась к страницам W(t,T), то при надлежащем выборе T с большой вероятностью эта программа будет обращаться к тем же страницам в период времени (t, t+T). Другими словами, принцип локальности утверждает, что если не слишком далеко заглядывать в будущее, то можно достаточно точно его прогнозировать исходя из прошлого. Понятно, что с течением времени рабочий набор процесса может изменяться (как по составу страниц, так и по их числу).

Наиболее важное свойство рабочего множества – его размер. ОС должна выделить каждому процессу достаточное число кадров, чтобы поместилось его рабочее множество. Если кадры еще остались, то может быть инициирован другой процесс. Если рабочие множества процессов не помещаются в память и начинается трешинг, то один из процессов можно выгрузить на диск.

Решение о размещении процессов в памяти должно, следовательно, базироваться на размере его рабочего множества. Для впервые инициируемых процессов это решение может быть принято эвристически. Во время работы процесса система должна уметь определять: расширяет процесс свое рабочее множество или перемещается на новое рабочее множество. Если в состав атрибутов страницы включить время последнего использования ti (для страницы с номером i ), то принадлежность i-й страницы к рабочему набору, определяемому параметром T в момент времени t будет выражаться неравенством: t-T <ti< t. Алгоритм выталкивания страниц WSClock, использующий информацию о рабочем наборе процесса, описан в [3].

Другой способ реализации данного подхода может быть основан на отслеживании количества страничных нарушений, вызываемых процессом. Если процесс часто генерирует pagefaults и память не слишком заполнена, то система может увеличить число выделенных ему кадров. Если же процесс не вызывает исключительных ситуаций в течение некоторого времени и уровень генерации ниже какого-то порога, то число кадров процесса может быть урезано. Этот способ регулирует лишь размер множества страниц, принадлежащих процессу, и должен быть дополнен какой-либо стратегией замещения страниц. Несмотря на то что система при этом может пробуксовывать в моменты перехода от одного рабочего множества к другому, предлагаемое решение в состоянии обеспечить наилучшую производительность для каждого процесса, не требуя никакой дополнительной настройки системы.

# Файловая система.

**Файловая система** – 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 такими данными. Организовать хранение информации на магнитном диске непросто. Это требует, например, хорошего знания устройства контроллера диска, особенностей работы с его регистрами. Непосредственное взаимодействие с диском – прерогатива компонента системы ввода-вывода ОС, называемого драйвером диска. Для того чтобы избавить пользователя компьютера от сложностей взаимодействия с аппаратурой, была придумана ясная абстрактная модель файловой системы. Операции записи или чтения файла концептуально проще, чем низкоуровневые операции работы с устройствами.

## Организация файлов

Перечислим ***основные функции*** файловой системы.

1. Идентификация файлов. Связывание имени файла с выделенным ему пространством внешней памяти.
2. Распределение внешней памяти между файлами. Для работы с конкретным файлом пользователю не требуется иметь информацию о местоположении этого файла на внешнем носителе информации. Например, для того чтобы загрузить документ в редактор с жесткого диска, нам не нужно знать, на какой стороне какого магнитного диска, на каком цилиндре и в каком секторе находится данный документ.
3. Обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера.
4. Обеспечение защиты от несанкционированного доступа.
5. Обеспечение совместного доступа к файлам, так чтобы пользователю не приходилось прилагать специальных усилий по обеспечению синхронизации доступа.
6. Обеспечение высокой производительности.

Программист воспринимает файл в виде набора однородных записей. Запись – это наименьший элемент данных, который может быть обработан как единое целое прикладной программой при обмене с внешним устройством. Причем в большинстве ОС размер записи равен одному байту. В то время как приложения оперируют записями, физический обмен с устройством осуществляется большими единицами (обычно блоками). Поэтому записи объединяются в блоки для вывода и разблокируются – для ввода. Файлы могут быть структурированы несколькими различными способами.

Три наиболее вероятные структуры показаны на рис. 4.1. Файл на рис. 4.1, *а* представляет собой бессистемную последовательность байтов. В сущности, операционной системе все равно, что содержится в этом файле, – она видит только байты. Какое-либо значение этим байтам придают программы на уровне пользователя. Такой подход используется как в UNIX, так и в Windows.



Рисунок 4.1 – Три типа файлов: *а –* последовательность байтов;

*б –* последовательность записей; *в –* дерево

**Типы файлов**

Многие операционные системы поддерживают несколько типов файлов. К примеру, в системах UNIX (опять же включая OS X) и Windows имеются обычные файлы и каталоги. В системе UNIX имеются также символьные и блочные специальные файлы.

*Обычными* считаются файлы, содержащие информацию пользователя. Все файлы на рис. 4.1 являются обычными.

*Каталоги –* это системные файлы, предназначенные для

поддержки структуры файловой системы.

*Символьные специальные файлы*имеют отношение к вводу-выводу и используются для моделирования последовательных устройств ввода-вывода, к которым относятся терминалы,принтеры и сети.

*Блочные специальные файлы*используются для моделирования

дисков.

Как правило, к обычным файлам относятся либо файлы ASCII, либо двоичные файлы. ASCII-файлы состоят из текстовых строк. В некоторых системах каждая строка завершается символом возврата каретки. В других системах используется символ перевода строки. Некоторые системы (например, Windows) используют оба символа. Строки необязательно должны иметь одинаковую длину.

ОС поддерживают несколько вариантов структуризации файлов.

**Последовательный файл**

Простейший вариант – так называемый последовательный файл. То есть файл является последовательностью записей. Поскольку записи, как правило, однобайтовые, файл представляет собой *неструктурированную последовательность байтов.*

Обработка подобных файлов предполагает последовательное чтение записей от начала файла, причем конкретная запись определяется ее положением в файле. Такой способ доступа называется последовательным (модель ленты). Если в качестве носителя файла используется магнитная лента, то так и делается. Текущая позиция считывания может быть возвращена к началу файла (rewind).

**Файл прямого доступа**

В реальной практике файлы хранятся на устройствах прямого (random) доступа, например на дисках, поэтому содержимое файла может быть разбросано по разным блокам диска, которые можно считывать в произвольном порядке. Причем номер блока однозначно определяется позицией внутри файла.

Здесь имеется в виду относительный номер, специфицирующий данный блок среди блоков диска, принадлежащих файлу. О связи относительного номера блока с абсолютным его номером на диске рассказывается в следующей лекции.

Естественно, что в этом случае для доступа к середине файла просмотр всего файла с самого начала не обязателен. Для специфицирования места, с которого надо начинать чтение, используются два способа: с начала или с текущей позиции, которую дает операция seek. Файл, байты которого могут быть считаны в произвольном порядке, называется файлом прямого доступа .

Таким образом, файл, состоящий из однобайтовых записей на устройстве прямого доступа, – наиболее распространенный способ организации файла. Базовыми операциями для такого рода файлов являются считывание или запись символа в текущую позицию. В большинстве языков высокого уровня предусмотрены операторы посимвольной пересылки данных в файл или из него.

Подобную логическую структуру имеют файлы во многих файловых системах, например в файловых системах ОС Unix и MS-DOS. ОС не осуществляет никакой интерпретации содержимого файла. Эта схема обеспечивает максимальную гибкость и универсальность. С помощью базовых системных вызовов (или функций библиотеки ввода/вывода) пользователи могут как угодно структурировать файлы. В частности, многие СУБД хранят свои базы данных в обычных файлах.

**Другие формы организации файлов**

Известны как другие формы организации файла, так и другие способы доступа к ним, которые использовались в ранних ОС, а также применяются сегодня в больших мэйнфреймах (mainframe), ориентированных на коммерческую обработку данных.

Первый шаг в структурировании – хранение файла в виде *последовательности записей фиксированной длины,* каждая из которых имеет внутреннюю структуру. Операция чтения производится над записью, а операция записи переписывает или добавляет запись целиком. Ранее использовались записи по 80 байт (это соответствовало числу позиций в перфокарте) или по 132 символа (ширина принтера). В ОС CP/M файлы были последовательностями 128-символьных записей. С введением CRT-терминалов данная идея утратила популярность.

Другой способ представления файлов – *последовательность записей переменной длины*, каждая из которых содержит ключевое поле в фиксированной позиции внутри записи (рис. 4.2). Базисная операция в данном случае – считать запись с каким-либо значением ключа. Записи могут располагаться в файле последовательно (например, отсортированные по значению ключевого поля) или в более сложном порядке. Метод доступа по значению ключевого поля к записям последовательного файла называется *индексно-последовательным.*

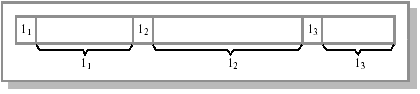


Рисунок 4.2 – Файл как последовательность записей переменной длины

В некоторых системах ускорение доступа к файлу обеспечивается конструированием *индекса*файла. Индекс обычно хранится на том же устройстве, что и сам файл, и состоит из списка элементов, каждый из которых содержит идентификатор записи, за которым следует указание о местоположении данной записи. Для поиска записи вначале происходит обращение к индексу, где находится указатель на нужную запись. Такие файлы называются *индексированными*, а метод доступа к ним – *доступ с использованием индекса*.

Предположим, у нас имеется большой несортированный файл, содержащий разнообразные сведения о студентах, состоящие из записей с несколькими полями, и возникает задача организации быстрого поиска по одному из полей, например по фамилии студента. Рис. 4.3 иллюстрирует решение данной проблемы – организацию метода доступа к файлу с использованием индекса.

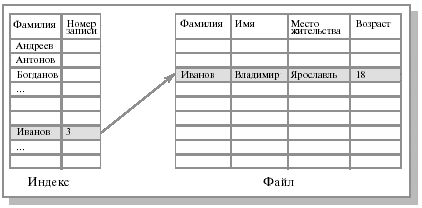


Рисунок 4.3 – Пример организации индекса для последовательного файла

Следует отметить, что почти всегда главным фактором увеличения скорости доступа является *избыточность* данных.

Способ выделения дискового пространства при помощи индексных узлов, применяемый в ряде ОС (Unix и некоторых других, см. следующую лекцию), может служить другим примером организации индекса.

В этом случае ОС использует древовидную организацию блоков, при которой блоки, составляющие файл, являются листьями дерева, а каждый внутренний узел содержит указатели на множество блоков файла. Для больших файлов индекс может быть слишком велик. В этом случае создают индекс для индексного файла (блоки промежуточного уровня или блоки косвенной адресации).

## Операции над файлами

Операционная система должна предоставить в распоряжение пользователя набор операций для работы с файлами, реализованных через системные вызовы. Чаще всего при работе с файлом пользователь выполняет не одну, а несколько операций. Во-первых, нужно найти данные файла и его атрибуты по символьному имени, во-вторых, считать необходимые атрибуты файла в отведенную область оперативной памяти и проанализировать права пользователя на выполнение требуемой операции. Затем следует выполнить операцию, после чего освободить занимаемую данными файла область памяти. Рассмотрим в качестве примера основные файловые операции ОС Unix [3].

* Создание файла, не содержащего данных. Смысл данного вызова – объявить, что файл существует, и присвоить ему ряд атрибутов. При этом выделяется место для файла на диске и вносится запись в каталог.
* Удаление файла и освобождение занимаемого им дискового пространства.
* Открытие файла. Перед использованием файла процесс должен его открыть. Цель данного системного вызова – разрешить системе проанализировать атрибуты файла и проверить права доступа к нему, а также считать в оперативную память список адресов блоков файла для быстрого доступа к его данным. Открытие файла является процедурой создания ***дескриптора*** или управляющего блока файла. Дескриптор (описатель) файла хранит всю информацию о нем. Иногда, в соответствии с парадигмой, принятой в языках программирования, под дескриптором понимается альтернативное имяфайла или указатель на описание файла в таблице открытых файлов, используемый при последующей работе с файлом. Например, на языке Cи операция открытия файла fd=open (pathname,flags,modes); возвращает дескриптор fd, который может быть задействован при выполнении операций чтения ( read (fd,buffer,count); ) или записи.
* Закрытие файла. 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.
* Позиционирование. Дает возможность специфицировать место внутри файла, откуда будет производиться считывание (или запись) данных, то есть задать **текущую** позицию.
* Чтение данных из файла. Обычно это делается с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в оперативной памяти.
* 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.

Есть и другие операции, например переименование файла, получение атрибутов файла и т. д.

**Директории. Логическая структура файлового архива**

Количество файлов на компьютере может быть большим. Отдельные системы хранят тысячи файлов, занимающие сотни гигабайт дискового пространства. Эффективное управление этими данными подразумевает наличие в них четкой логической структуры. Все современные файловые системы поддерживают многоуровневое именование файлов за счет наличия во внешней памяти дополнительных файлов со специальной структурой – каталогов (или директорий ).

Каждый каталог содержит список каталогов и/или файлов, содержащихся в данном каталоге. Каталоги имеют один и тот же внутренний формат, где каждому файлу соответствует одна запись в файле директории (например, рис.4.4).

Число директорий зависит от системы. В ранних ОС имелась только одна корневая директория, затем появились директории для пользователей (по одной директории на пользователя). В современных ОС используется произвольная структура дерева директорий.

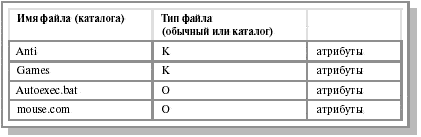


Рисунок 4.4 – Директории

Таким образом, файлы на диске образуют иерархическую древовидную структуру (рис.4.5).

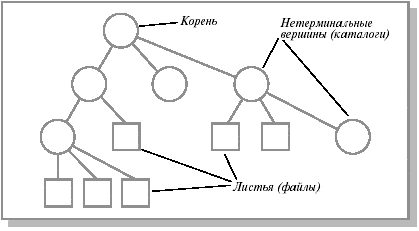


Рисунок 4.5 – Древовидная структура файловой системы

Существует несколько эквивалентных способов изображения дерева. Структура перевернутого дерева наиболее распространена. Верхнюю вершину называют корнем. Если элемент дерева не может иметь потомков, он называется терминальной вершиной или листом (в данном случае является файлом ). Нелистовые вершины – справочники или каталоги содержат списки листовых и нелистовых вершин. Путь от корня к файлу *однозначно* определяет файл.

Подобные древовидные структуры являются графами, не имеющими циклов. Можно считать, что ребра графа направлены вниз, а корень – вершина, не имеющая входящих ребер. Как мы увидим в следующей лекции, связывание файлов, которое практикуется в ряде операционных систем, приводит к образованию циклов в графе.

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 так называемым *абсолютным или полным именем (pathname)*, состоящим из списка имен вложенных каталогов, по которому можно найти путь от корня к файлу плюс имя файла в каталоге, непосредственно содержащем данный файл. То есть полное имя включает цепочку имен – путь к файлу, например /usr/games/doom. Такие имена уникальны. Компоненты пути разделяют различными символами: "/" (слэш) в Unix или обратными слэшем в MS-DOS (в Multics – ">"). Таким образом, использование древовидных каталогов минимизирует сложность назначения уникальных имен.

Указывать полное имя не всегда удобно, поэтому применяют другой способ задания имени – *относительный*путь к файлу. Он использует концепцию рабочей или текущей директории, которая обычно входит в состав атрибутов процесса, работающего с данным файлом. Тогда на файлы в такой директории можно ссылаться только по имени, при этом поискфайла будет осуществляться в рабочем каталоге. Это удобнее, но, по существу, то же самое, что и абсолютная форма.

Для получения доступа к файлу и локализации его блоков система должна выполнить навигацию по каталогам. Рассмотрим для примера путь /usr/linux/progr.c. Алгоритм одинаков для всех иерархических систем. Сначала в фиксированном месте на диске находится корневая директория. Затем находится компонент пути usr, т. е. в корневой директории ищется файл/usr. Исследуя этот файл, система понимает, что данный файл является каталогом, и блоки его данных рассматривает как список файлов и ищет следующий компонент linux в нем. Из строки для linux находится файл, соответствующий компоненту usr/linux/. Затем находится компонент progr.c, который открывается, заносится в таблицу открытых файлов и сохраняется в ней до закрытия файла.

Отклонение от типовой обработки компонентов pathname может возникнуть в том случае, когда этот компонент является не обычным каталогом с соответствующим ему индексным узлом и списком файлов, а служит точкой связывания (принято говорить "точкой монтирования") двух файловых архивов. Этот случай рассмотрен в следующей лекции.

Многие прикладные программы работают с файлами, находящимися в текущей директории, не указывая явным образом ее имени. Это дает пользователю возможность произвольным образом именовать каталоги, содержащие различные программные пакеты. Для реализации этой возможности в большинстве ОС, поддерживающих иерархическую структуру директорий, используется обозначение " ." – для текущей директории и " .." – для родительской.

Разделы диска. Организация доступа к архиву файлов.

Задание пути к файлу в файловых системах некоторых ОС отличается тем, с чего начинается эта цепочка имен.

В современных ОС принято разбивать диски на *логические диски* (это низкоуровневая операция), иногда называемые *разделами* (partitions). Бывает, что, наоборот, объединяют несколько физических дисков в один логический диск (например, это можно сделать в ОС Windows NT). Поэтому в дальнейшем изложении мы будем игнорировать проблему физического выделения пространства для файлов и считать, что каждый раздел представляет собой отдельный (виртуальный) диск. Диск содержит иерархическую древовидную структуру, состоящую из набора файлов, каждый из которых является хранилищем данных пользователя, и каталогов или директорий (то есть файлов, которые содержат перечень других файлов, входящих в состав каталога ), необходимых для хранения информации о файлах системы.

В некоторых системах управления файлами требуется, чтобы каждый архив файлов целиком располагался на одном диске (разделе диска). В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск (буквы диска). Например, c:\util\nu\ndd.exe. Такой способ именования используется в файловых системах DEC и Microsoft.

В других системах (Multics) вся совокупность файлов и каталогов представляет собой единое дерево. Сама система, выполняя поиск файлов по имени, начиная с корня, требовала установки необходимых дисков.

В ОС Unix предполагается наличие нескольких архивов файлов, каждый на своем разделе, один из которых считается корневым. После запуска системы можно **«***смонтировать***»** корневую файловую систему и ряд изолированных файловых систем в одну общую файловую систему.

Технически это осуществляется с помощью создания в корневой файловой системе специальных пустых каталогов (см. также следующую лекцию). Специальный системный вызов mount ОС Unix позволяет подключить к одному из этих пустых каталогов корневой каталог указанного архива файлов. После монтирования общей файловой системы именование файлов производится так же, как если бы она с самого начала была централизованной. Задачей ОС является беспрепятственный проход точки монтирования при получении доступа к файлу по цепочке имен. Если учесть, что обычно монтирование файловой системы производится при загрузке системы, пользователи ОС Unix обычно и не задумываются о происхождении общей файловой системы.

**Операции над директориями**

Как и в случае с файлами, система обязана обеспечить пользователя набором операций, необходимых для работы с директориями, реализованных через системные вызовы. Несмотря на то что директории – это файлы, логика работы с ними отличается от логики работы с обычными файлами и определяется природой этих объектов, предназначенных для поддержки структуры файлового архива. Совокупность системных вызовов для управления директориями зависит от особенностей конкретной ОС. Напомним, что операции над каталогами являются прерогативой ОС, то есть пользователь не может, например, выполнить запись в каталог начиная с текущей позиции. Рассмотрим в качестве примера некоторые системные вызовы, необходимые для работы с каталогами [3].

* Создание директории. Вновь созданная директория включает записи с именами ' .' и ' ..', однако считается пустой.
* Удаление директории. Удалена может быть только пустая директория.
* Открытие директории для последующего чтения. Например, чтобы перечислить файлы, входящие в директорию, процесс должен открыть директорию и считать имена всех файлов, которые она включает.
* Закрытие директории после ее чтения для освобождения места во внутренних системных таблицах.
* Поиск. Данный системный вызов возвращает содержимое текущей записи в открытой директории. Вообще говоря, для этих целей может использоваться системный вызов Read, но в этом случае от программиста потребуется знание внутренней структуры директории.
* Получение списка файлов в каталоге.
* Переименование. Имена директорий можно менять, как и имена файлов.
* Создание файла. При создании нового файла необходимо добавить в каталог соответствующий элемент.
* Удаление файла. Удаление из каталога соответствующего элемента. Если удаляемый файл присутствует только в одной директории, то он вообще удаляется из файловой системы, в противном случае система ограничивается только удалением специфицируемой записи.

Очевидно, что создание и удаление файлов предполагает также выполнение соответствующих файловых операций. Имеется еще ряд других системных вызовов, например связанных с защитой информации.

**Защита файлов**

Общие проблемы безопасности ОС рассмотрены в лекциях 15-16. Информация в компьютерной системе должна быть защищена как от физического *разрушения* (reliability), так и от несанкционированного *доступа* (protection).

**Контроль доступа к файлам**

Наличие в системе многих пользователей предполагает организацию контролируемого доступа к файлам. Выполнение любой операции над файлом должно быть разрешено только в случае наличия у пользователя соответствующих привилегий. Обычно контролируются следующие операции: чтение, запись и выполнение. Другие операции, например копирование файлов или их переименование, также могут контролироваться. Однако они чаще реализуются через перечисленные. Так, операцию копирования файлов можно представить как операцию чтения и последующую операцию записи.

**Списки прав доступа**

Наиболее общий подход к защите файлов от несанкционированного использования – сделать доступ зависящим от идентификатора пользователя, то есть связать с каждым файлом или директорией *список прав доступа* (accesscontrollist), где перечислены имена пользователей и типы разрешенных для них способов доступа к файлу. Любой запрос на выполнение операции сверяется с таким списком. Основная проблема реализации данного способа – список может быть длинным. Чтобы разрешить всем пользователям читать файл, необходимо всех их внести в список. У такой техники есть два нежелательных следствия.

* Конструирование подобного списка может оказаться сложной задачей, особенно если мы не знаем заранее пользователей системы.
* Запись в директории должна иметь переменный размер (включать список потенциальных пользователей).

Для решения этих проблем создают классификации пользователей, например, в ОС Unix все пользователи разделены на три группы.

* *Владелец* (Owner).
* *Группа* (Group). Набор пользователей, разделяющих файл и нуждающихся в типовом способе доступа к нему.
* *Остальные* (Univers).

Это позволяет реализовать конденсированную версию списка прав доступа. В рамках такой ограниченной классификации задаются только три поля (по одному для каждой группы) для каждой контролируемой операции. В итоге в Unix операции чтения, записи и исполнения контролируются при помощи 9 бит (rwxrwxrwx).

# Управление устройствами

Функционирование любой вычислительной системы обычно сводится к выполнению двух видов работы: обработке информации и операций по осуществлению ее ввода-вывода. Поскольку в рамках модели, принятой в данном курсе, все, что выполняется в вычислительной системе, организовано как набор процессов, эти два вида работы выполняются процессами. Процессы занимаются обработкой информации и выполнением операций ввода-вывода.

Содержание понятий «обработка информации» и «операции ввода-вывода» зависит от того, с какой точки зрения мы смотрим на них. С точки зрения программиста, под «обработкой информации» понимается выполнение команд процессора над данными, лежащими в памяти независимо от уровня иерархии – в регистрах, кэше, оперативной или вторичной памяти. Под «операциями ввода-вывода» программист понимает обмен данными между памятью и устройствами, внешними по отношению к памяти и процессору, такими как флешки, диски, монитор, клавиатура, таймер. С точки зрения операционной системы «обработкой информации» являются только операции, совершаемые процессором над данными, находящимися в памяти на уровне иерархии не ниже, чем оперативная память. Все остальное относится к "операциям ввода-вывода". Чтобы выполнять операции над данными, временно расположенными во вторичной памяти, операционная система, сначала производит их подкачку в оперативную память, и лишь затем процессор совершает необходимые действия.

**Физические принципы организации ввода-вывода**

Существует много разнообразных устройств, которые могут взаимодействовать с процессором и памятью: таймер, жесткие диски, клавиатура, дисплеи, мышь, модемы и т. д., вплоть до устройств отображения и ввода информации в авиационно-космических тренажерах. Часть этих устройств может быть встроена внутрь корпуса компьютера, часть – вынесена за его пределы и общаться с компьютером через различные линии связи: кабельные, оптоволоконные, радиорелейные, спутниковые и т. д. Конкретный набор устройств и способы их подключения определяются целями функционирования вычислительной системы, желаниями и финансовыми возможностями пользователя. Несмотря на все многообразие устройств, управление их работой и обмен информацией с ними строятся на относительно небольшом наборе принципов, которые мы постараемся разобрать в этом разделе.

**Общие сведения об архитектуре компьютера**

В простейшем случае процессор, память и многочисленные внешние устройства связаны большим количеством электрических соединений – *линий*, которые в совокупности принято называть локальной магистралью компьютера. Внутри локальной магистрали линии, служащие для передачи сходных сигналов и предназначенные для выполнения сходных функций, принято группировать в **шины**. При этом понятие шины включает в себя не только набор проводников, но и набор жестко заданных протоколов, определяющий перечень сообщений, который может быть передан с помощью электрических сигналов по этим проводникам. В современных компьютерах выделяют как минимум три шины:

* шину данных, состоящую из линий данных и служащую для передачи информации между процессором и памятью, процессором и устройствами ввода-вывода, памятью и внешними устройствами;
* адресную шину, состоящую из линий адреса и служащую для задания адреса ячейки памяти или указания устройства ввода-вывода, участвующих в обмене информацией;
* шину управления, состоящую из линий управления локальной магистралью и линий ее состояния, определяющих поведение локальной магистрали . В некоторых архитектурных решениях линии состояния выносятся из этой шины в отдельную шину состояния.

Количество линий, входящих в состав шины, принято называть *разрядностью( шириной* ) этой шины. Ширина адресной шины, например, определяет максимальный размер оперативной памяти, которая может быть установлена в вычислительной системе. Ширина шины данных определяет максимальный объем информации, которая за один раз может быть получена или передана по этой шине.

Операции обмена информацией осуществляются при одновременном участии всех шин. Рассмотрим, к примеру, действия, которые должны быть выполнены для передачи информации из процессора в память. В простейшем случае необходимо выполнить три действия.

1. На адресной шине процессор должен выставить сигналы, соответствующие адресу ячейки памяти, в которую будет осуществляться передача информации.
2. На шину данных процессор должен выставить сигналы, соответствующие информации, которая должна быть записана в память.
3. После выполнения действий 1 и 2 на шину управления выставляются сигналы, соответствующие операции записи и работе с памятью, что приведет к занесению необходимой информации по нужному адресу.

Естественно, что приведенные выше действия являются необходимыми, но недостаточными при рассмотрении работы конкретных процессоров и микросхем памяти. Конкретные архитектурные решения могут требовать дополнительных действий: например, выставления на шину управления сигналов частичного использования шины данных (для передачи меньшего количества информации, чем позволяет ширина этой шины); выставления сигнала готовности магистрали после завершения записи в память, разрешающего приступить к новой операции, и т. д. Однако общие принципы выполнения операции записи в память остаются неизменными.

В то время как память легко можно представить себе в виде последовательности пронумерованных адресами ячеек, локализованных внутри одной микросхемы или набора микросхем, к устройствам ввода-вывода подобный подход неприменим. Внешние устройства разнесены пространственно и могут подключаться к локальной магистрали в одной точке или множестве точек, получивших название портов ввода-вывода . Тем не менее, точно так же, как ячейки памяти взаимно однозначно отображались в адресное пространство памяти, порты ввода-вывода можно взаимно однозначно отобразить в другое адресное пространство – адресное пространство ввода-вывода. При этом каждый порт ввода-вывода получает свой номер или адрес в этом пространстве. В некоторых случаях, когда адресное пространство памяти (размер которого определяется шириной адресной шины ) задействовано не полностью (остались адреса, которым не соответствуют физические ячейки памяти) и протоколы работы с внешним устройством совместимы с протоколами работы с памятью, часть портов ввода -вывода может быть отображена непосредственно в адресное пространство памяти (так, например, поступают с видеопамятью дисплеев), правда, тогда эти порты уже не принято называть портами. Надо отметить, что при отображении портов в адресное пространство памяти для организации доступа к ним в полной мере могут быть задействованы существующие механизмы защиты памяти без организации специальных защитных устройств.

В ситуации прямого отображения портов ввода-вывода в адресное пространство памяти действия, необходимые для записи информации и управляющих команд в эти порты или для чтения данных из них и их состояний, ничем не отличаются от действий, производимых для передачи информации между оперативной памятью и процессором, и для их выполнения применяются те же самые команды. Если же порт отображен в адресное пространство ввода-вывода, то процесс обмена информацией инициируется специальными командами ввода-вывода и включает в себя несколько другие действия. Например, для передачи данных в порт необходимо выполнить следующее.

* На адресной шине процессор должен выставить сигналы, соответствующие адресу порта, в который будет осуществляться передача информации, в адресном пространстве ввода-вывода.
* На шину данных процессор должен выставить сигналы, соответствующие информации, которая должна быть передана в порт.
* После выполнения действий 1 и 2 на шину управления выставляются сигналы, соответствующие операции записи и работе с устройствами ввода-вывода (переключение адресных пространств!), что приведет к передаче необходимой информации в нужный порт.

Существенное отличие памяти от устройств ввода-вывода заключается в том, что занесение информации в память является окончанием операции записи, в то время как занесение информации в порт зачастую представляет собой инициализацию реального совершения операции ввода-вывода. Что именно должны делать устройства, приняв информацию через свой порт, и каким именно образом они должны поставлять информацию для чтения из порта, определяется электронными схемами устройств, получившими название контроллеров .Контроллер может непосредственно управлять отдельным устройством (например, контроллер диска), а может управлять несколькими устройствами, связываясь с их контроллерами посредством специальных шин ввода-вывода (шина IDE, шина SCSI и т. д.).

Современные вычислительные системы могут иметь разнообразную архитектуру, множество шин и магистралей, мосты для перехода информации от одной шины к другой и т. п. Для нас сейчас важными являются только следующие моменты.

* Устройства ввода-вывода подключаются к системе через порты.
* Могут существовать два адресных пространства: пространство памяти и пространство ввода-вывода.
* Порты, как правило, отображаются в адресное пространство ввода-вывода и иногда – непосредственно в адресное пространство памяти.
* Использование того или иного адресного пространства определяется типом команды, выполняемой процессором, или типом ее операндов.
* Физическим управлением устройством ввода-вывода, передачей информации через порт и выставлением некоторых сигналов на магистрали занимается контроллер устройства.

Именно единообразие подключения внешних устройств к вычислительной системе является одной из составляющих идеологии, позволяющих добавлять новые устройства без перепроектирования всей системы.

**Структура контроллера устройства**

Контроллеры устройств ввода-вывода весьма различны как по своему внутреннему строению, так и по исполнению (от одной микросхемы до специализированной вычислительной системы со своим процессором, памятью и т. д.), поскольку им приходится управлять совершенно разными приборами. Не вдаваясь в детали этих различий, мы выделим некоторые общие черты контроллеров, необходимые им для взаимодействия с вычислительной системой. Обычно каждый контроллер имеет по крайней мере четыре внутренних регистра, называемых регистрами состояния: управления, входных данных и выходных данных. Для доступа к содержимому этих регистров вычислительная система может использовать один или несколько портов, что для нас не существенно. Для простоты изложения будем считать, что каждому регистру соответствует свой порт.

Регистр состояния содержит биты, значение которых определяется состоянием устройства ввода-вывода и которые доступны только для чтения вычислительной системой. Эти биты индицируют завершение выполнения текущей команды на устройстве (бит занятости), наличие очередного данного в регистре выходных данных (бит готовности данных), возникновение ошибки при выполнении команды ( бит ошибки ) и т. д.

Регистр управления получает данные, которые записываются вычислительной системой для инициализации устройства ввода-вывода или выполнения очередной команды, а также изменения режима работы устройства. Часть битов в этом регистре может быть отведена под код выполняемой команды, часть битов будет кодировать режим работы устройства, бит **готовности команды** свидетельствует о том, что можно приступить к ее выполнению.

Регистр выходных данных служит для помещения в него данных для чтения вычислительной системой, а регистр входных данных предназначен для помещения в него информации, которая должна быть выведена на устройство. Обычно емкость этих регистров не превышает ширину линии данных (а чаще всего меньше ее), хотя некоторые контроллеры могут использовать в качестве регистров очередь FIFO для буферизации поступающей информации.

Разумеется, набор регистров и составляющих их битов приблизителен, он призван послужить нам моделью для описания процесса передачи информации от вычислительной системы к внешнему устройству и обратно, но в том или ином виде он обычно присутствует во всех контроллерах устройств.

**Опрос устройств и прерывания. Исключительные ситуации и системные вызовы**

Построив модель контроллера и представляя себе, что скрывается за словами "прочитать информацию из порта" и "записать информацию в порт", мы готовы к рассмотрению процесса взаимодействия устройства и процессора. Как и в предыдущих случаях, примером нам послужит команда записи, теперь уже записи или вывода данных на внешнее устройство. В нашей модели для вывода информации, помещающейся в регистр входных данных, без проверки успешности вывода процессор и контроллер должны связываться следующим образом.

1. Процессор в цикле читает информацию из порта регистра состояний и проверяет значение бита занятости. Если бит занятости установлен, то это означает, что устройство еще не завершило предыдущую операцию, и процессор уходит на новую итерацию цикла. Если бит занятости сброшен, то устройство готово к выполнению новой операции, и процессор переходит на следующий шаг.
2. Процессор записывает код команды вывода в порт регистра управления.
3. Процессор записывает данные в порт регистра входных данных.
4. Процессор устанавливает бит готовности команды. В следующих шагах процессор не задействован.
5. Когда контроллер замечает, что бит готовности команды установлен, он устанавливает бит занятости.
6. Контроллер анализирует код команды в регистре управления и обнаруживает, что это команда вывода. Он берет данные из регистра входных данных и инициирует выполнение команды.
7. После завершения операции контроллер обнуляет бит готовности команды.
8. При успешном завершении операции контроллер обнуляет бит ошибки в регистре состояния, при неудачном завершении команды – устанавливает его.
9. Контроллер сбрасывает бит занятости.

При необходимости вывода новой порции информации все эти шаги повторяются. Если процессор интересует, корректно или некорректно была выведена информация, то после шага 4 он должен в цикле считывать информацию из порта регистра состояний до тех пор, пока не будет сброшен бит занятости устройства, после чего проанализировать состояние бита ошибки.

Как видим, на первом шаге (и, возможно, после шага 4) процессор ожидает освобождения устройства, непрерывно опрашивая значение бита занятости. Такой способ взаимодействия процессора и контроллера получил название polling или, в русском переводе, способа опроса устройств. Если скорости работы процессора и устройства ввода-вывода примерно равны, то это не приводит к существенному уменьшению полезной работы, совершаемой процессором. Если же скорость работы устройства существенно меньше скорости процессора, то указанная техника резко снижает производительность системы и необходимо применять другой архитектурный подход.

Для того чтобы процессор не дожидался состояния готовности устройства ввода-вывода в цикле, а мог выполнять в это время другую работу, необходимо, чтобы устройство само умело сигнализировать процессору о своей готовности. Технический механизм, который позволяет внешним устройствам оповещать процессор о завершении команды вывода или команды ввода, получил название механизма прерываний.

В простейшем случае для реализации механизма прерываний необходимо к имеющимся у нас шинам локальной магистрали добавить еще одну линию, соединяющую процессор и устройства ввода-вывода – линию прерываний. По завершении выполнения операции внешнее устройство выставляет на эту линию специальный сигнал, по которому процессор после выполнения очередной команды (или после завершения очередной итерации при выполнении цепочечных команд, т. е. команд, повторяющихся циклически со сдвигом по памяти) изменяет свое поведение. Вместо выполнения очередной команды из потока команд он частично сохраняет содержимое своих регистров и переходит на выполнение программы обработки прерывания, расположенной по заранее оговоренному адресу. При наличии только одной линии прерываний процессор при выполнении этой программы должен опросить состояние всех устройств ввода-вывода, чтобы определить, от какого именно устройства пришло прерывание( polling   прерываний!), выполнить необходимые действия (например, вывести в это устройство очередную порцию информации или перевести соответствующий процесс из состояния ожидание в состояние готовность ) и сообщить устройству, что прерывание обработано (снять прерывание ).

В большинстве современных компьютеров процессор стараются полностью освободить от необходимости опроса внешних устройств, в том числе и от определения с помощью опроса устройства, сгенерировавшего сигнал прерывания. Устройства сообщают о своей готовности процессору не напрямую, а через специальный контроллер прерываний, при этом для общения с процессором он может использовать не одну линию, а целую шину прерываний. Каждому устройству присваивается свой номер прерывания, который при возникновении прерывания   контроллер прерывания заносит в свой регистр состояния и, возможно, после распознавания процессором сигнала прерывания и получения от него специального запроса выставляет на шину прерываний или шину данных для чтения процессором. Номер прерывания обычно служит индексом в специальной таблице прерываний, хранящейся по адресу, задаваемому при инициализации вычислительной системы, и содержащей адреса программ обработки прерываний – векторы   прерываний. Для распределения устройств по номерам прерываний необходимо, чтобы от каждого устройства к контроллеру прерываний шла специальная линия, соответствующая одному номеру прерывания. При наличии множества устройств такое подключение становится невозможным, и на один проводник (один номер прерывания ) подключается несколько устройств. В этом случае процессор при обработке прерывания все равно вынужден заниматься опросом устройств для определения устройства, выдавшего прерывание, но в существенно меньшем объеме. Обычно при установке в систему нового устройства ввода-вывода требуется аппаратно или программно определить, каким будет номер прерывания, вырабатываемый этим устройством.

Рассматривая кооперацию процессов и взаимоисключения, мы говорили о существовании критических секций внутри ядра операционной системы, при выполнении которых необходимо исключить всякие прерывания от внешних устройств. Для запрещения прерываний, а точнее, для невосприимчивости процессора к внешним прерываниям обычно существуют специальные команды, которые могут маскировать (запрещать) все или некоторые из прерываний устройств ввода-вывода. В то же время определенные кризисные ситуации в вычислительной системе (например, неустранимый сбой в работе оперативной памяти) должны требовать ее немедленной реакции. Такие ситуации вызывают прерывания, которые невозможно замаскировать или запретить и которые поступают в процессор по специальной линии шины прерываний, называемой линией немаскируемых прерываний (NMI – Non-MaskableInterrupt).

Не все внешние устройства являются одинаково важными с точки зрения вычислительной системы ("все животные равны, но некоторые равнее других"). Соответственно, некоторые прерывания являются более существенными, чем другие. Контроллер прерываний обычно позволяет устанавливать приоритеты для прерываний от внешних устройств. При почти одновременном возникновении прерываний от нескольких устройств (во время выполнения одной и той же команды процессора) процессору сообщается номер наиболее приоритетного прерывания для его обслуживания в первую очередь. Менее приоритетное прерывание при этом не пропадает, о нем процессору будет доложено после обработки более приоритетного прерывания. Более того, при обработке возникшего прерывания процессор может получить сообщение о возникновении прерывания с более высоким приоритетом и переключиться на его обработку.

Механизм обработки прерываний, по которому процессор прекращает выполнение команд в обычном режиме и, частично сохранив свое состояние, отвлекается на выполнение других действий, оказался настолько удобен, что зачастую разработчики процессоров используют его и для других целей. Хотя эти случаи и не относятся к операциям ввода-вывода, мы вынуждены упомянуть их здесь, для того чтобы их не путали с прерываниями. Похожим образом процессор обрабатывает исключительные ситуации и программные прерывания.

Для внешних прерываний характерны следующие особенности.

* Внешнее прерывание обнаруживается процессором между выполнением команд (или между итерациями в случае выполнения цепочечных команд).
* Процессор при переходе на обработку прерывания сохраняет часть своего состояния перед выполнением следующей команды.
* Прерывания происходят асинхронно с работой процессора и непредсказуемо, программист никоим образом не может предугадать, в каком именно месте работы программы произойдет прерывание.
* Исключительные ситуации возникают во время выполнения процессором команды. К их числу относятся ситуации переполнения, деления на ноль, обращения к отсутствующей странице памяти и т. д. Для исключительных ситуаций характерно следующее.
* Исключительные ситуации обнаруживаются процессором во время выполнения команд.
* Процессор при переходе на выполнение обработки исключительной ситуации сохраняет часть своего состояния перед выполнением текущей команды.
* Исключительные ситуации возникают синхронно с работой процессора, но непредсказуемо для программиста, если только тот специально не заставил процессор делить некоторое число на ноль.
* Программные прерывания возникают после выполнения специальных команд, как правило, для выполнения привилегированных действий внутри системных вызовов. Программные прерывания имеют следующие свойства.
* Программное прерывание происходит в результате выполнения специальной команды.
* Процессор при выполнении программного прерывания сохраняет свое состояние перед выполнением **следующей** команды.
* Программные прерывания, естественно, возникают синхронно с работой процессора и **абсолютно предсказуемы** программистом.

Надо сказать, что реализация похожих механизмов обработки внешних прерываний, исключительных ситуаций и программных прерываний лежит целиком на совести разработчиков процессоров. Существуют вычислительные системы, где все три ситуации обрабатываются по-разному.

**Прямой доступ к памяти (Direct Memory Access – DMA)**

Использование механизма прерываний позволяет разумно загружать процессор в то время, когда устройство ввода-вывода занимается своей работой. Однако запись или чтение большого количества информации из адресного пространства ввода-вывода (например, с диска) приводят к большому количеству операций ввода-вывода, которые должен выполнять процессор. Для освобождения процессора от операций последовательного вывода данных из оперативной памяти или последовательного ввода в нее был предложен механизм прямого доступа внешних устройств к памяти – ПДП или DirectMemoryAccess – DMA. Давайте кратко рассмотрим, как работает этот механизм.

Для того чтобы какое-либо устройство, кроме процессора, могло записать информацию в память или прочитать ее из памяти, необходимо чтобы это устройство могло забрать у процессора управление локальной магистралью для выставления соответствующих сигналов на шины адреса, данных и управления. Для централизации эти обязанности обычно возлагаются не на каждое устройство в отдельности, а на специальный контроллер – контроллер прямого доступа к памяти. Контроллер прямого доступа к памяти имеет несколько спаренных линий – каналов DMA, которые могут подключаться к различным устройствам. Перед началом использования прямого доступа к памяти этот контроллер необходимо запрограммировать, записав в его порты информацию о том, какой канал или каналы предполагается задействовать, какие операции они будут совершать, какой адрес памяти является начальным для передачи информации и какое количество информации должно быть передано. Получив по одной из линий – каналов DMA, сигнал запроса на передачу данных от внешнего устройства, контроллер по шине управления сообщает процессору о желании взять на себя управление локальной магистралью. Процессор, возможно, через некоторое время, необходимое для завершения его действий с магистралью, передает управление ею контроллеру DMA, известив его специальным сигналом. Контроллер DMA выставляет на адресную шину адрес памяти для передачи очередной порции информации и по второй линии канала прямого доступа к памяти сообщает устройству о готовности магистрали к передаче данных. После этого, используя шину данных и шину управления, контроллер DMA, устройство ввода-вывода и память осуществляют процесс обмена информацией. Затем контроллер прямого доступа к памяти извещает процессор о своем отказе от управления магистралью, и тот берет руководящие функции на себя. При передаче большого количества данных весь процесс повторяется циклически.

При прямом доступе к памяти процессор и контроллер DMA по очереди управляют локальной магистралью. Это, конечно, несколько снижает производительность процессора, так как при выполнении некоторых команд или при чтении очередной порции команд во внутренний кэш он должен поджидать освобождения магистрали, но в целом производительность вычислительной системы существенно возрастает.

При подключении к системе нового устройства, которое умеет использовать прямой доступ к памяти, обычно необходимо программно или аппаратно задать номер канала DMA, к которому будет приписано устройство. В отличие от прерываний, где один номер прерывания мог соответствовать нескольким устройствам, каналы DMA всегда находятся в монопольном владении устройств.

**Логические принципы организации ввода-вывода**

Рассмотренные в предыдущем разделе физические механизмы взаимодействия устройств ввода-вывода с вычислительной системой позволяют понять, почему разнообразные внешние устройства легко могут быть добавлены в существующие компьютеры. Все, что необходимо сделать пользователю при подключении нового устройства, – это отобразить порты устройства в соответствующее адресное пространство, определить, какой номер будет соответствовать прерыванию, генерируемому устройством, и, если нужно, закрепить за устройством некоторый канал DMA. Для нормального функционирования hardware этого будет достаточно. Однако мы до сих пор ничего не сказали о том, как должна быть построена подсистема управления вводом-выводом в операционной системе для легкого и безболезненного добавления новых устройств и какие функции вообще обычно на нее возлагаются.

***Структура системы ввода-вывода***

Если поручить неподготовленному пользователю сконструировать систему ввода-вывода, способную работать со всем множеством внешних устройств, то, скорее всего, он окажется в ситуации, в которой находились биологи и зоологи до появления трудов Линнея. Все устройства разные, отличаются по выполняемым функциям и своим характеристикам, и кажется, что принципиально невозможно создать систему, которая без больших постоянных переделок позволяла бы охватывать все многообразие видов. Вот перечень лишь нескольких направлений (далеко не полный), по которым различаются устройства.

* Скорость обмена информацией может варьироваться в диапазоне от нескольких байтов в секунду (клавиатура) до нескольких гигабайтов в секунду (сетевые карты).
* Одни устройства могут использоваться несколькими процессами параллельно (являются разделяемыми), в то время как другие требуют монопольного захвата процессом.
* Устройства могут запоминать выведенную информацию для ее последующего ввода или не обладать этой функцией. Устройства, запоминающие информацию, в свою очередь, могут дифференцироваться по формам доступа к сохраненной информации: обеспечивать к ней последовательный доступ в жестко заданном порядке или уметь находить и передавать только необходимую порцию данных.
* Часть устройств умеет передавать данные только по одному байту последовательно ( символьные устройства ), а часть устройств умеет передавать блок байтов как единое целое ( блочные устройства ).
* Существуют устройства, предназначенные только для ввода информации, устройства, предназначенные только для вывода информации, и устройства, которые могут выполнять и ввод, и вывод.

В области технического обеспечения удалось выделить несколько основных принципов взаимодействия внешних устройств с вычислительной системой, т. е. создать единый интерфейс для их подключения, возложив все специфические действия на контроллеры самих устройств. Тем самым конструкторы вычислительных систем переложили все хлопоты, связанные с подключением внешней аппаратуры, на разработчиков самой аппаратуры, заставляя их придерживаться определенного стандарта.

Похожий подход оказался продуктивным и в области программного подключения устройств ввода-вывода. Подобно тому как Линнею удалось заложить основы систематизации знаний о растительном и животном мире, разделив все живое в природе на относительно небольшое число классов и отрядов, мы можем разделить устройства на относительно небольшое число типов, отличающихся по набору операций, которые могут быть ими выполнены, считая все остальные различия несущественными. Мы можем затем специфицировать интерфейсы между ядром операционной системы, осуществляющим некоторую общую политику ввода-вывода, и программными частями, непосредственно управляющими устройствами, для каждого из таких типов. Более того, разработчики операционных систем получают возможность освободиться от написания и тестирования этих специфических программных частей, получивших название драйверов, передав эту деятельность производителям самих внешних устройств. Фактически мы приходим к использованию принципа уровневого или слоеного построения системы управления вводом-выводом для операционной системы (рис. 5.1).

Два нижних уровня этой слоеной системы составляет hardware: сами устройства, непосредственно выполняющие операции, и их контроллеры, служащие для организации совместной работы устройств и остальной вычислительной системы. Следующий уровень составляют драйверы устройств ввода-вывода, скрывающие от разработчиков операционных систем особенности функционирования конкретных приборов и обеспечивающие четко определенный интерфейс между hardware и вышележащим уровнем – уровнем базовой подсистемы ввода-вывода, которая, в свою очередь, предоставляет механизм взаимодействия между драйверами и программной частью вычислительной системы в целом.

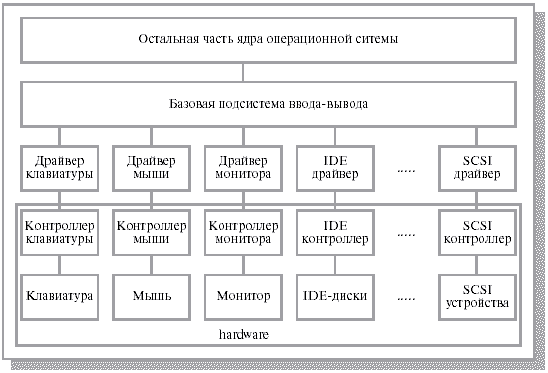


Рисунок 5.1 – Структура системы ввода-вывода

**Систематизация внешних устройств и интерфейс между базовой подсистемой ввода-вывода и драйверами**

Как и система видов Линнея, система типов устройств является далеко не полной и не строго выдержанной. Устройства обычно принято разделять по преобладающему типу интерфейса на следующие виды:

* символьные (клавиатура, модем, терминал и т. п.);
* блочные (магнитные и оптические диски и ленты, и т. д.);
* сетевые (сетевые карты);
* все остальные (таймеры, графические дисплеи, телевизионные устройства, видеокамеры и т. п.);

Такое деление является весьма условным. В одних операционных системах сетевые устройства могут не выделяться в отдельную группу, в некоторых других – отдельные группы составляют звуковые устройства и видеоустройства и т. д. Некоторые группы в свою очередь могут разбиваться на подгруппы: подгруппа жестких дисков, подгруппа мышек, подгруппа принтеров. Нас такие детали не интересуют. Мы не ставим перед собой цель осуществить систематизацию всех возможных устройств, которые могут быть подключены к вычислительной системе. Единственное, для чего нам понадобится эта классификация, так это для иллюстрации того положения, что устройства могут быть разделены на группы по выполняемым ими функциям, и для понимания функций драйверов, и интерфейса между ними и базовой подсистемой ввода-вывода.

Для этого мы рассмотрим только две группы устройств: символьные и блочные. Как уже упоминалось в предыдущем разделе, символьные устройства – это устройства, которые умеют передавать данные только последовательно, байт за байтом, а блочные устройства – это устройства, которые могут передавать блок байтов как единое целое.

К символьным устройствам обычно относятся устройства ввода информации, которые спонтанно генерируют входные данные: клавиатура, мышь, модем, джойстик. К ним же относятся и устройства вывода информации, для которых характерно представление данных в виде линейного потока: принтеры, звуковые карты и т. д. По своей природе символьные устройства обычно умеют совершать две общие операции: ввести символ (байт) и вывести символ (байт) – get и put.

Для блочных устройств, таких как магнитные и оптические диски, ленты и т. п. естественными являются операции чтения и записи блока информации – read и write, а также, для устройств прямого доступа, операция поиска требуемого блока информации – seek.

Драйверы символьных и блочных устройств должны предоставлять базовой подсистеме ввода-вывода функции для осуществления описанных общих операций. Помимо общих операций, некоторые устройства могут выполнять операции специфические, свойственные только им – например, звуковые карты умеют увеличивать или уменьшать среднюю громкость звучания, дисплеи умеют изменять свою разрешающую способность. Для выполнения таких специфических действий в интерфейс между драйвером и базовой подсистемой ввода-вывода обычно входит еще одна функция, позволяющая непосредственно передавать драйверу устройства произвольную команду с произвольными параметрами, что позволяет задействовать любую возможность драйвера без изменения интерфейса. В операционной системе Unix такая функция получила название ioctl (от input-outputcontrol).

Помимо функций read, write, seek (для блочных устройств ), get, put (для символьных устройств ) и ioctl, в состав интерфейса обычно включают еще следующие функции.

* Функцию инициализации или повторной инициализации работы драйвера и устройства – open.
* Функцию временного завершения работы с устройством (может, например, вызывать отключение устройства) – close.
* Функцию опроса состояния устройства (если по каким-либо причинам работа с устройством производится методом опроса его состояния, например, в операционных системах Windows NT и Windows 9x так построена работа с принтерами через параллельный порт) – poll.
* Функцию остановки драйвера, которая вызывается при остановке операционной системы или выгрузке драйвера из памяти, halt.

Существует еще ряд действий, выполнение которых может быть возложено на драйвер, но поскольку, как правило, это действия базовой подсистемы ввода-вывода, мы поговорим о них в следующем разделе. Приведенные выше названия функций, конечно, являются условными и могут меняться от одной операционной системы к другой, но действия, выполняемые драйверами, характерны для большинства операционных систем, и соответствующие функции присутствуют в интерфейсах к ним.

**Функции базовой подсистемы ввода-вывода**

Базовая подсистема ввода-вывода служит посредником между процессами вычислительной системы и набором драйверов. Системные вызовы для выполнения операций ввода-вывода трансформируются ею в вызовы функций необходимого драйвера устройства. Однако обязанности базовой подсистемы не сводятся к выполнению только действий трансляции общего системного вызова в обращение к частной функции драйвера. Базовая подсистема предоставляет вычислительной системе такие услуги, как поддержка блокирующихся, неблокирующихся и асинхронных системных вызовов, буферизация и кэширование входных и выходных данных, осуществление spooling'a и монопольного захвата внешних устройств, обработка ошибок и прерываний, возникающих при операциях ввода-вывода, планирование последовательности запросов на выполнение этих операций. Давайте остановимся на этих услугах подробнее.

***Блокирующиеся, неблокирующиеся и асинхронные системные вызовы***

Все системные вызовы, связанные с осуществлением операций ввода-вывода, можно разбить на три группы по способам реализации взаимодействия процесса и устройства ввода-вывода.

* К первой, наиболее привычной для большинства программистов группе относятся блокирующиеся системные вызовы. Как следует из самого названия, применение такого вызова приводит к блокировке инициировавшего его процесса, т. е. процесс переводится операционной системой из состояния исполнение в состояние ожидание. Завершив выполнение всех операций ввода-вывода, предписанных системным вызовом, операционная система переводит процесс из состояния ожидание в состояние готовность. После того как процесс будет снова выбран для исполнения, в нем произойдет окончательный возврат из системного вызова. Типичным для применения такого системного вызова является случай, когда процессу необходимо получить от устройства строго определенное количество данных, без которых он не может выполнять работу далее.
* Ко второй группе относятся неблокирующиеся системные вызовы. Их название не совсем точно отражает суть дела. В простейшем случае процесс, применивший неблокирующийся вызов, не переводится в состояние ожидание вообще. Системный вызов возвращается немедленно, выполнив предписанные ему операции ввода-вывода полностью, частично или не выполнив совсем, в зависимости от текущей ситуации (состояния устройства, наличия данных и т. д.). В более сложных ситуациях процесс может блокироваться, но условием его разблокирования является завершение всех необходимых операций или окончание некоторого промежутка времени. Типичным случаем применения неблокирующегося системного вызова может являться периодическая проверка на поступление информации с клавиатуры при выполнении трудоемких расчетов.
* К третьей группе относятся асинхронные системные вызовы. Процесс, использовавший асинхронный системный вызов, никогда в нем не блокируется. Системный вызов инициирует выполнение необходимых операций ввода-вывода и немедленно возвращается, после чего процесс продолжает свою регулярную деятельность. Об окончании завершения операции ввода-вывода операционная система впоследствии информирует процесс изменением значений некоторых переменных, передачей ему сигнала или сообщения или каким-либо иным способом. Необходимо четко понимать разницу между неблокирующимися и асинхронными вызовами. Неблокирующийся системный вызов для выполнения операции read вернется немедленно, но может прочитать запрошенное количество байтов, меньшее количество или вообще ничего. Асинхронный системный вызов для этой операции также вернется немедленно, но требуемое количество байтов рано или поздно будет прочитано в полном объеме.

***Буферизация и кэширование***

Под буфером обычно понимается некоторая область памяти для запоминания информации при обмене данных между двумя устройствами, двумя процессами или процессом и устройством. Обмен информацией между двумя процессами относится к области кооперации процессов, и мы подробно рассмотрели его организацию в соответствующей лекции. Здесь нас будет интересовать использование буферов в том случае, когда одним из участников обмена является внешнее устройство. Существует три причины, приводящие к использованию буферов в базовой подсистеме ввода-вывода.

* Первая причина буферизации – это разные скорости приема и передачи информации, которыми обладают участники обмена. Рассмотрим, например, случай передачи потока данных от клавиатуры к модему. Скорость, с которой поставляет информацию клавиатура, определяется скоростью набора текста человеком и обычно существенно меньше скорости передачи данных модемом. Для того чтобы не занимать модем на все время набора текста, делая его недоступным для других процессов и устройств, целесообразно накапливать введенную информацию в буфере или нескольких буферах достаточного размера и отсылать ее через модем после заполнения буферов.
* Вторая причина буферизации – это разные объемы данных, которые могут быть приняты или получены участниками обмена единовременно. Возьмем другой пример. Пусть информация поставляется модемом и записывается на жесткий диск. Помимо обладания разными скоростями совершения операций, модем и жесткий диск представляют собой устройства разного типа. Модем является символьным устройством и выдает данные байт за байтом, в то время как диск является блочным устройством и для проведения операции записи для него требуется накопить необходимый блок данных в буфере. Здесь также можно применять более одного буфера. После заполнения первого буфера модем начинает заполнять второй, одновременно с записью первого на жесткий диск. Поскольку скорость работы жесткого диска в тысячи раз больше, чем скорость работы модема, к моменту заполнения второго буфера операция записи первого будет завершена, и модем снова сможет заполнять первый буфер одновременно с записью второго на диск.
* Третья причина буферизации связана с необходимостью копирования информации из приложений, осуществляющих ввод-вывод, в буфер ядра операционной системы и обратно. Допустим, что некоторый пользовательский процесс пожелал вывести информацию из своего адресного пространства на внешнее устройство. Для этого он должен выполнить системный вызов с обобщенным названием write, передав в качестве параметров адрес области памяти, где расположены данные, и их объем. Если внешнее устройство временно занято, то возможна ситуация, когда к моменту его освобождения содержимое нужной области окажется испорченным (например, при использовании асинхронной формы системного вызова). Чтобы избежать возникновения подобных ситуаций, проще всего в начале работы системного вызова скопировать необходимые данные в буфер ядра операционной системы, постоянно находящийся в оперативной памяти, и выводить их на устройство из этого буфера.

Под словом ***кэш*** (cache – "тайник, запас"), этимологию которого мы не будем здесь рассматривать, обычно понимают область быстрой памяти, содержащую копию данных, расположенных где-либо в более медленной памяти, предназначенную для ускорения работы вычислительной системы. Мы с вами сталкивались с этим понятием при рассмотрении иерархии памяти. В базовой подсистеме ввода-вывода не следует смешивать два понятия, буферизацию и кэширование, хотя зачастую для выполнения этих функций отводится одна и та же область памяти. Буфер часто содержит единственный набор данных, существующий в системе, в то время как кэш по определению содержит копию данных, существующих где-нибудь еще. Например, буфер, используемый базовой подсистемой для копирования данных из пользовательского пространства процесса при выводе на диск, может в свою очередь применяться как кэш для этих данных, если операции модификации и повторного чтения данного блока выполняются достаточно часто.

Функции буферизации и кэширования не обязательно должны быть локализованы в базовой подсистеме ввода-вывода. Они могут быть частично реализованы в драйверах и даже в контроллерах устройств, скрытно по отношению к базовой подсистеме.

***Spooling и захват устройств***

О понятии **spooling** мы говорили в начале пособия, как о механизме, впервые позволившем совместить реальные операции ввода-вывода одного задания с выполнением другого задания. Теперь мы можем определить это понятие более точно. Под словом spool мы подразумеваем буфер, содержащий входные или выходные данные для устройства, на котором следует избегать чередования его использования (возникновения interleaving – см. раздел "Interleaving, racecondition и взаимоисключения" лекции 5) различными процессами. Правда, в современных вычислительных системах spool для ввода данных практически не используется, а в основном предназначен для накопления выходной информации.

Рассмотрим в качестве внешнего устройства принтер. Хотя принтер не может печатать информацию, поступающую одновременно от нескольких процессов, может оказаться желательным разрешить процессам совершать вывод на принтер параллельно. Для этого операционная система вместо передачи информации напрямую на принтер накапливает выводимые данные в буферах на диске, организованных в виде отдельного spool-файла для каждого процесса. После завершения некоторого процесса соответствующий ему spool-файл ставится в очередь для реальной печати. Механизм, обеспечивающий подобные действия, и получил название spooling.

В некоторых операционных системах вместо использования spooling для устранения racecondition применяется механизм монопольного захвата устройств процессами. Если устройство свободно, то один из процессов может получить его в монопольное распоряжение. При этом все другие процессы при попытке осуществления операций над этим устройством будут либо блокированы (переведены в состояние ожидание ), либо получат информацию о невозможности выполнения операции до тех пор, пока процесс, захвативший устройство, не завершится или явно не сообщит операционной системе о своем отказе от его использования.

Обеспечение spooling и механизма захвата устройств является прерогативой базовой подсистемы ввода-вывода.

***Обработка прерываний и ошибок***

Если при работе с внешним устройством вычислительная система не пользуется методом опроса его состояния, а задействует механизм прерываний, то при возникновении прерывания, как мы уже говорили раньше, процессор, частично сохранив свое состояние, передает управление специальной программе обработки прерывания. Мы уже рассматривали действия операционной системы над процессами, происходящими при возникновении прерывания, в разделе "Переключение контекста" лекции 2, где после возникновения прерывания осуществлялись следующие действия: сохранение контекста, обработка прерывания, планирование использования процессора, восстановление контекста. Тогда мы обращали больше внимания на действия, связанные с сохранением и восстановлением контекста и планированием использования процессора. Теперь давайте подробнее остановимся на том, что скрывается за словами "обработка прерывания ".

Одна и та же процедура обработки прерывания может применяться для нескольких устройств ввода-вывода (например, если эти устройства используют одну линию прерываний, идущую от них к контроллеру прерываний ), поэтому первое действие собственно программы обработки состоит в определении того, какое именно устройство выдало прерывание. Зная устройство, мы можем выявить процесс, который инициировал выполнение соответствующей операции. Поскольку прерывание возникает как при удачном, так и при неудачном ее выполнении, следующее, что мы должны сделать, – это определить успешность завершения операции, проверив значение бита ошибки в регистре состояния устройства. В некоторых случаях операционная система может предпринять определенные действия, направленные на компенсацию возникшей ошибки. Например, в случае возникновения ошибки чтения с гибкого диска можно попробовать несколько раз повторить выполнение команды. Если компенсация ошибки невозможна, то операционная система впоследствии известит об этом процесс, запросивший выполнение операции, (например, специальным кодом возврата из системного вызова). Если этот процесс был заблокирован до выполнения завершившейся операции, то операционная система переводит его в состояние готовность. При наличии других неудовлетворенных запросов к освободившемуся устройству операционная система может инициировать выполнение следующего запроса, одновременно известив устройство, что прерывание обработано. На этом, собственно, обработка прерывания заканчивается, и система может приступать к планированию использования процессора.

Действия по обработке прерывания и компенсации возникающих ошибок могут быть частично переложены на плечи соответствующего драйвера. Для этого в состав интерфейса между драйвером и базовой подсистемой ввода-вывода добавляют еще одну функцию – функцию обработки прерывания   intr.

***Планирование запросов***

При использовании неблокирующегося системного вызова может оказаться, что нужное устройство уже занято выполнением некоторых операций. В этом случае неблокирующийся вызов может немедленно вернуться, не выполнив запрошенных команд. При организации запроса на совершение операций ввода-вывода с помощью блокирующегося или асинхронного вызова занятость устройства приводит к необходимости постановки запроса в очередь к данному устройству. В результате с каждым устройством оказывается связан список неудовлетворенных запросов процессов, находящихся в состоянии ожидания, и запросов, выполняющихся в асинхронном режиме. Состояние ожидание расщепляется на набор очередей процессов, дожидающихся различных устройств ввода-вывода.

После завершения выполнения текущего запроса операционная система (по ходу обработки возникшего прерывания ) должна решить, какой из запросов в списке должен быть удовлетворен следующим, и инициировать его исполнение. Точно так же, как для выбора очередного процесса на исполнение из списка готовых нам приходилось осуществлять краткосрочное планирование процессов, здесь нам необходимо осуществлять планирование применения устройств, пользуясь каким-либо алгоритмом этого планирования. Критерии и цели такого планирования мало отличаются от критериев и целей планирования процессов.

Задача планирования использования устройства обычно возлагается на базовую подсистему ввода-вывода, однако для некоторых устройств лучшие алгоритмы планирования могут быть тесно связаны с деталями их внутреннего функционирования. В таких случаях операция планирования переносится внутрь драйвера соответствующего устройства, так как эти детали скрыты от базовой подсистемы. Для этого в интерфейс драйвера добавляется еще одна специальная функция, которая осуществляет выбор очередного запроса, – функция strategy.

В следующем разделе мы рассмотрим некоторые алгоритмы планирования, связанные с удовлетворением запросов, на примере жесткого диска.

**Алгоритмы планирования запросов к жесткому диску**

Прежде чем приступить к непосредственному изложению самих алгоритмов, давайте вспомним внутреннее устройство жесткого диска и определим, какие параметры запросов мы можем использовать для планирования.

***Строение жесткого диска и параметры планирования***

Современный жесткий магнитный диск представляет собой набор круглых пластин, находящихся на одной оси и покрытых с одной или двух сторон специальным магнитным слоем (рис. 5.2). Около каждой рабочей поверхности каждой пластины расположены магнитные головки для чтения и записи информации. Эти головки присоединены к специальному рычагу, который может перемещать весь блок головок над поверхностями пластин как единое целое. Поверхности пластин разделены на концентрические кольца, внутри которых, собственно, и может храниться информация. Набор концентрических колец на всех пластинах для одного положения головок (т. е. все кольца, равноудаленные от оси) образует цилиндр. Каждое кольцо внутри цилиндра получило название дорожки (по одной или две дорожки на каждую пластину). Все дорожки делятся на равное число секторов. Количество дорожек, цилиндров и секторов может варьироваться от одного жесткого диска к другому в достаточно широких пределах. Как правило, сектор является минимальным объемом информации, которая может быть прочитана с диска за один раз.

При работе диска набор пластин вращается вокруг своей оси с высокой скоростью, подставляя по очереди под головки соответствующих дорожек все их сектора. Номер сектора, номер дорожки и номер цилиндра однозначно определяют положение данных на жестком диске и, наряду с типом совершаемой операции – чтение или запись, полностью характеризуют часть запроса, связанную с устройством, при обмене информацией в объеме одного сектора.

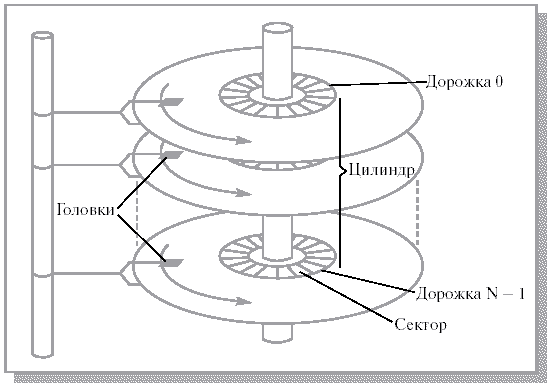


Рисунок 5.2 – Схема жесткого диска

При планировании использования жесткого диска естественным параметром планирования является время, которое потребуется для выполнения очередного запроса. Время, необходимое для чтения или записи определенного сектора на определенной дорожке определенного цилиндра, можно разделить на две составляющие: время обмена информацией между магнитной головкой и компьютером, которое обычно не зависит от положения данных и определяется скоростью их передачи (transferspeed), и время, необходимое для позиционирования головки над заданным сектором, – время позиционирования (positioningtime). Время позиционирования, в свою очередь, состоит из времени, необходимого для перемещения головок на нужный цилиндр, – времени поиска (seektime) и времени, которое требуется для того, чтобы нужный сектор довернулся под головку, – задержки на вращение (rotationallatency). Времена поиска пропорциональны разнице между номерами цилиндров предыдущего и планируемого запросов, и их легко сравнивать. Задержка на вращение определяется довольно сложными соотношениями между номерами цилиндров и секторов предыдущего и планируемого запросов и скоростями вращения диска и перемещения головок. Без знания соотношения этих скоростей сравнение становится невозможным. Поэтому естественно, что набор параметров планирования сокращается до времени поиска различных запросов, определяемого текущим положением головки и номерами требуемых цилиндров, а разницей в задержках на вращение пренебрегают.

***Алгоритм First Come First Served (FCFS)***

Простейшим алгоритмом, к которому мы уже должны были привыкнуть, является алгоритм FirstComeFirstServed (FCFS) – первым пришел, первым обслужен. Все запросы организуются в очередь FIFO и обслуживаются в порядке поступления. Алгоритм прост в реализации, но может приводить к достаточно длительному общему времени обслуживания запросов. Рассмотрим пример. Пусть у нас на диске из 100 цилиндров (от 0 до 99) есть следующая очередь запросов: 23, 67, 55, 14, 31, 7, 84, 10 и головки в начальный момент находятся на 63-м цилиндре. Тогда положение головок будет меняться следующим образом:

63->23->67->55->14->31->7->84->10

и всего головки переместятся на 329 цилиндров. Неэффективность алгоритма хорошо иллюстрируется двумя последними перемещениями с 7 цилиндра через весь диск на 84 цилиндр и затем опять через весь диск на цилиндр 10. Простая замена порядка двух последних перемещений (7 \to10 \to84) позволила бы существенно сократить общее время обслуживания запросов. Поэтому давайте перейдем к рассмотрению другого алгоритма.

***Алгоритм ShortSeekTimeFirst (SSTF)***

Как мы убедились, достаточно разумным является первоочередное обслуживание запросов, данные для которых лежат рядом с текущей позицией головок, а уж затем далеко отстоящих. Алгоритм ShortSeekTimeFirst (SSTF) – короткое время поиска первым – как раз и исходит из этой позиции. Для очередного обслуживания будем выбирать запрос, данные для которого лежат наиболее близко к текущему положению магнитных головок. Естественно, что при наличии равноудаленных запросов решение о выборе между ними может приниматься исходя из различных соображений, например по алгоритму FCFS. Для предыдущего примера алгоритм даст такую последовательность положений головок:

63->67->55->31->23->14->10->7->84

и всего головки переместятся на 141 цилиндр. Заметим, что наш алгоритм похож на алгоритм SJF планирования процессов, если за аналог оценки времени очередного CPU burst процесса выбирать расстояние между текущим положением головки и положением, необходимым для удовлетворения запроса. И точно так же, как алгоритм SJF, он может приводить к длительному откладыванию выполнения какого-либо запроса. Необходимо вспомнить, что запросы в очереди могут появляться в любой момент времени. Если у нас все запросы, кроме одного, постоянно группируются в области с большими номерами цилиндров, то этот один запрос может находиться в очереди неопределенно долго.

Точный алгоритм SJF являлся оптимальным для заданного набора процессов с заданными временами CPU burst. Очевидно, что алгоритм SSTF не является оптимальным. Если мы перенесем обслуживание запроса 67-го цилиндра в промежуток между запросами 7-го и 84-го цилиндров, мы уменьшим общее время обслуживания. Это наблюдение приводит нас к идее целого семейства других алгоритмов – алгоритмов сканирования.

***Алгоритмы сканирования (SCAN, C-SCAN, LOOK, C-LOOK)***

В простейшем из алгоритмов сканирования – SCAN – головки постоянно перемещаются от одного края диска до другого, по ходу дела обслуживая все встречающиеся запросы. По достижении другого края направление движения меняется, и все повторяется снова. Пусть в предыдущем примере в начальный момент времени головки двигаются в направлении уменьшения номеров цилиндров. Тогда мы и получим порядок обслуживания запросов, подсмотренный в конце предыдущего раздела. Последовательность перемещения головок выглядит следующим образом:

63->55->31->23->14->10->7->0->67->84

и всего головки переместятся на 147 цилиндров.

Если мы знаем, что обслужили последний попутный запрос в направлении движения головок, то мы можем не доходить до края диска, а сразу изменить направление движения на обратное:

63->55->31->23->14->10->7->67->84

и всего головки переместятся на 133 цилиндра. Полученная модификация алгоритма SCAN получила название LOOK.

Допустим, что к моменту изменения направления движения головки в алгоритме SCAN, т. е. когда головка достигла одного из краев диска, у этого края накопилось большое количество новых запросов, на обслуживание которых будет потрачено достаточно много времени (не забываем, что надо не только перемещать головку, но еще и передавать прочитанные данные!). Тогда запросы, относящиеся к другому краю диска и поступившие раньше, будут ждать обслуживания несправедливо долго. Для сокращения времени ожидания запросов применяется другая модификация алгоритма SCAN – циклическое сканирование. Когда головка достигает одного из краев диска, она без чтения попутных запросов (иногда существенно быстрее, чем при выполнении обычного поиска цилиндра) перемещается на другой край, откуда вновь начинает движение в прежнем направлении. Для этого алгоритма, получившего название C-SCAN, последовательность перемещений будет выглядеть так:

63->55->31->23->14->10->7->0->99->84->67

По аналогии с алгоритмом LOOK для алгоритма SCAN можно предложить и алгоритм C-LOOK для алгоритма C-SCAN:

63->55->31->23->14->10->7->84->67

Существуют и другие разновидности алгоритмов сканирования, и совсем другие алгоритмы, но мы на этом закончим, ибо было сказано: "И еще раз говорю: никто не обнимет необъятного".

# Библиографический список

1. Операционная система [Электронный ресурс] : Материал из Википедии – свободной энциклопедии : Версия 85762200, сохранённая в 08:00 UTC 3 июня 2017 / Авторы Википедии // Википедия, свободная энциклопедия– Электрон. дан. – Сан-Франциско: Фонд Викимедиа, 2017. – Режим доступа: http://ru.wikipedia.org/?oldid=85762200 Проверенно 7.07.2017
2. **Карпов, В.Е.** Учебное пособие / Электронное издание / В.Е. Карпов, К.А. Коньков – М.: ИНТУИТ. РУ "Интернет университет информационных технологий", 2005 – 536 с.
3. **Таненбаум,Э.** Современные операционные системы. 4-е изд. / Э. Таненбаум, Х.Бос – СПб.: Питер, 2015. – 1120 с.

*Учебное издание*

**Москат** Наталья Александровна

**ОПЕРАЦИОННЫЕ СИСТЕМЫ**

**ОПЕРАЦИОННЫЕ СИСТЕМЫ и кроссплатформенное програмирование**

Редактор Л.И. Сергейчик

Корректор Л.И. Сергейчик

Подписано в печать 12.05.17. Формат 60×84/16.

Бумага газетная. Ризография. Усл. печ. л. 3,3.

Тираж экз. Изд. № 17. Заказ .

Редакционно-издательский центр ФГБОУ ВО РГУПС.

«Электронный университет» ФГБОУ ВО РГУПС

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Адрес университета: 344038, г. Ростов н/Д, пл. Ростовского Стрелкового Полка

Народного Ополчения, 2.

|  |  |
| --- | --- |
|  | © Москат Н.А., 2017 |
|  | © ФГБОУ ВО РГУПС, 2017 |
|  | © «Электронный университет» ФГБОУ ВО РГУПС,2017 |