Подготовка к “Операционным системам”

Комментарий

Данная шпора, по сути, является электронным конспектом. Костяк тем - то что начитал Симоненко потоку ИВ-7х (2010 год) за 2 семестра. Существенное отличие от предыдущих годов - тема “Непротиворечивость в распределенных системах”. Данная тема полностью выдрана из книги Таненбаума, про распределенные системы. В шпоре приведен конспект соответствующей главы и попытки выделения основных идей.

Материал шпоры: частично написан самостоятельно, частично содран с конспекта, частично из просторов интернета или книг Таненбаума.

Содержание

Определения

Расширяемость – простота подключения новых устройств.

Масштабируемость – система может подключать дополнительное оборудование для увеличения производительности.

Маршаллинг – преобразования объекта в памяти в формат данных, пригодный для хранения или передачи. Обычно применяется, когда данные необходимо передавать между различными частями одной программы или от одной программы к другой. (c) Wiki

(альтернативное определение: проход по уровням чего-либо сверху-вниз с целью сбора данных)

Симоненко несколько раз приводил пример маршаллинга как упаковки данных в пакеты при пересылке по сети.

Демаршаллинг – обратный процесс. Как бы распаковка данных для того, чтоб их можно было поместить в память.(c) Wiki

(альтернативное определение: проход по уровням чего-либо снизу-вверх с целью сбора данных)

Кэш-промах – если при обращении к кешу, в нём не оказалось запрашиваемых данных, то происходит кеш-промах, в процессе которого необходимое значение подгружается в кеш.

Однопрограммный режим работы – если все ресурсы машины отданы 1 задаче, которая не может быть прервана (она завершается либо сама запланировано, либо аварийно)

Многопрограммный режим работы – если в системе находится несколько задач на разных стадиях исполнения, каждая из которых может быть прервана и восстановлена в любой момент времени.

Пробуксовка системы – процессор нагружен на 100%, а реальной работы не выполняется.

Спин-блокировка – возникает когда процесс требует ресурс, в то время как этот ресурс захвачен другим процессом. В таком случае этот процесс постоянно опрашивает ресурс занимая все процессорное время.

Задание – внешняя единица работы системы, на которую система ресурсы не выделяет. Работа, которую мы хотим дать ОС, чтоб она её выполнила. Задания накапливаются в буферах. В задании должна быть указана программа и данные. Как только появляются ресурсы, задание расшифровывается. Формируется task control block (TCB), он же process control block (PCB).

Задача – внутренняя единица системы, для которой система выделяет ресурсы (активизирует задание).

Процесс – любая выполняемая программа системы; это динамический объект системы, которому она выделяет ресурсы; траектория процессора в адресном пространстве машины. ОС всегда должна знать, что происходит с процессом.

Программное обеспечение – совокупность программ и инструкций по их использованию.

Математическое обеспечение – совокупность программ, инструкций по их использованию и описание математического аппарата, использованного для создания программы.

Ресурсы – то, что может понадобиться для выполнения программы (задачи, процесса). Все ресурсы можно разделить условно на две категории: *Физические* устройства в составе компа. *Внутренние* ресурсы пользовательских программ (данные, подпрограммы и т. д.).

I семестр

Этапы развития вычислительной техники

Таненбаум выделяет 4 этапа развития компьютеров/ОС:

I поколение (1945-1955) Лампы, коммутационные панели

Использовались лампы. Составные блоки машины соединялись посредством коммутационных панелей. ОС еще не было. Только прямые численные расчеты (sin, cos, ln)

II поколение (1955-1965) транзисторы, системы пакетной обработки

Компьютеры стали более надежными и быстродействие увеличилось. С перфокарт задания (силами маломощного компьютера) переносились на магнитную ленту, с которой работал основной (мощный компьютер), так как заданий на ленте было много, нужно

было управлять ими (начинать, заканчивать и тд) – появился прообраз ОС. Выходные данные так же записывались на магнитную ленту.

III поколение (1965-1980) интегральные схемы и многозадачность

IBM создала совместимую линейку машин разной мощности и одну операционную систему, которая работала на этих компьютерах (очень сложную). Были созданы средства аппаратной поддержки многозадачности и системы разделения времени (MULTICS, из которого берет свои корни Unix).

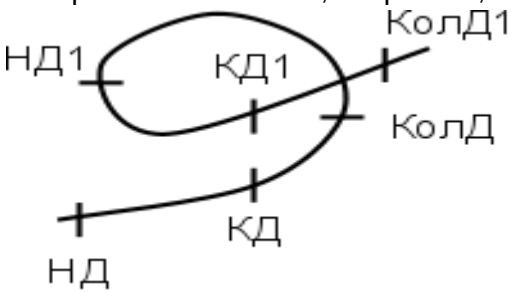
Так же в это время начали развиваться относительно дешевые компьютеры, которые на специфических заданиях почти не уступали по производительности большим мейнфреймам.

IV поколение (1980-наши дни) персональные компьютеры

В конспекте этапы другие.

Методы доступа пользователя к ресурсам

Исторический контекст, спиралька, в которой всё повторяется.



НД – Непосредственный доступ. Все ресурсы одному пользователю, машина работала медленно, поэтому человеческой реакции вполне хватало.

КД (первый прообраз ОС) – Косвенный доступ. Скорость работы машины увеличилась, человек стал тормозить. Появился так называемый однопрограммный режим работы, программы загружали операторы.

КолД – Коллективный доступ. Скорость проца так увеличилась, что, пока тупит один человек, проц может решать другую задачу, появились терминалы – многопрограмный режим работы.

НД1 – Непосредственный доступ 1. Большинство пользователей сидит за своими персоналками – персональные компьютеры.

КД1 – Косвенный доступ 1. Программы начали отправлять на сервер.

КолД1 – Коллективный доступ 1. Клиент-серверный – сервер обрабатывает множество запросов. Возможно, это cloud computing? или “тонкие клиенты”.

Организация многопрограммного режима работы

Система работает в *многопрограммном* режиме, если в ней находится несколько задач в разной стадии исполнения, и каждая из них может быть прервана другой с последующим возвратом.

Многопрограммный режим в однопроцессорной системе – имеем один набор оборудования (проц) и много процессов. ОС планирует выполнение процессов во времени, синхронизируя их работу. Важно для ОС поддерживать защиту данных процессов друг от друга. О синхронизации заботится программист.

Многопрограммный режим в многопроцессорной системе – это планирование во времени и в пространстве. ОС распределяет процессы на процессоры, синхронизирует их работу во времени (см. статическое и динамическое планирование). ОС заботится об эффективной нагрузке на ресурсы (процы).

По конспекту выделяем:

1. **классическое мультипрограммирование**

Режим работы истинного совмещения, когда разные блоки могут заниматься разными задачами одновременно. Проц, выполняющий пользовательскую задачу, может работать одновременно с УВВ.

2. **параллельная обработка**

Режим кажущегося параллелизма (ну, вы знаете, задачи делятся по квантам и выполняются на проце по очереди кванты маленькие, создаётся впечатление параллелизма).

3. **режим разделения времени**

Совмещает классическое мультипрограммирование и параллельную обработку + доступ привилегированных пользователей к ресурсам машины. Одна машина и много терминалов доступа. При этом часть задач (таких как ввод или редактирование данных оператором) могла исполняться в режиме диалога, а другие задачи (такие как массивные вычисления) – в пакетном режиме.

4. **работа в реальном времени**

Время ответа соответствует заранее заданным характеристика (определено внешними факторами). Если есть хоть один процесс, который невозможно прервать – уже не режим реального времени. Если процесс не может выполниться за отведённое ему время, должен быть зафиксирован сбой в его работе. Операционная система должна за предсказуемое время отреагировать на непредсказуемое появление внешних событий.

Функции операционной системы

1) Обеспечить пользователя средствами для решения его конкретных задач

2) Обеспечение эффективного управления оборудованием при выполнении основных функций

(Таненбаум выделяет несколько иные (но в целом похожие) цели:

а) ОС как виртуальная машина (то есть расширение функций компьютера),

б) Управление ресурсами)

Основные функции:

- Выполнение элементарных (низкоуровневых) действий, которые являются общими для большинства программ и часто встречаются почти во всех программах (ввод и вывод данных, запуск и остановка других программ, выделение и освобождение дополнительной памяти и др.).
- Управление процессами

- Управление оперативной памятью
- Организация файловой системы
- Обеспечение пользовательского интерфейса.
- Сетевые операции, поддержка стека сетевых протоколов.

Ссылочки на вики про ОС и про Пуллинг

Классификация операционных систем

1. Однопрограммные (в однопрограмных системах обычно ОС нет вообще)
2. Многопрограммные
3. Сетевые (операционная система со встроенными возможностями для работы в компьютерных сетях) (c) Wiki
4. Распределённые (должна поддерживаться функция прозрачности (по времени, месторасположению, etc.), масштабируемости (если не хватает производительности - подключаются новые ресурсы), расширяемости)
5. Метавычисления
6. Кластерные вычисления (объединение составных систем по определенном признаку)
7. GRID система

Распределённые системы

Распределённые вычислительные системы

Среда, в которой компоненты системы или ресурсы: процессоры, память, принтеры, графические станции, программы, данные и т.д., связанны вместе посредством сети, которая позволяет пользователям представлять ВС как единую вычислительную среду и иметь доступ к ее ресурсам.

Особенности:

- **Ограниченность** связана с тем, что количество узлов в сети ограничено и они являются независимыми компонентами сети.
- **Идентификация.** Каждый из ресурсов сети должен однозначно идентифицироваться (именоваться, например).
- **Распределенное управление.** Каждый узел должен иметь возможность и средства (hardware, software) для управления сетью. Но с т.з. надёжности нежелательно давать каким-то узлам больше привилегий в управлении.
- **Гетерогенность.** Система должна работать на разнородных узлах с различной длиной слов и байтовой организацией. Это касается программного, прикладного и системного обеспечения узлов.

Пользователь не должен знать особенности аппаратного обеспечения сети.

Распределённые операционные системы

РОС строятся на РВС.

Свойства распределенных операционных систем:

- Надо поддерживать когерентность файлов.
- Распределенная система распределяет выполняемые работы в узлах системы, исходя из соображений повышения пропускной способности всей системы;
- Распределенные системы имеют высокий уровень организации параллельных вычислений

Принципы построения распределенных ОС

1. Прозрачность (для пользователя и программы)

Прозрачность сети требует, чтобы детали сети были скрыты от конечных пользователей.

- расположения – пользователь не должен знать, где расположены ресурсы
- миграции – ресурсы могут перемещаться без изменения их имен
- размножения – пользователь не должен знать, сколько копий существует
- конкуренции – множество пользователей разделяет ресурсы автоматически
- параллелизма – работа может выполняться параллельно без участия пользователя
- именование – имя должно быть уникальным в глобальном смысле, и не имеет значения в каком месте системы оно будет использовано

2. Гибкость (не все еще ясно - потребуется менять решения)

Использование монолитного ядра ОС или микроядра.

3. Надежность

Доступность, устойчивость к ошибкам (fault tolerance).

Секретность.

4. Производительность

Грануллированность. Мелкозернистый и крупнозернистый параллелизм (fine-grained parallelism, coarse-grained parallelism). Устойчивость к ошибкам требует дополнительных накладных расходов.

5. Масштабируемость – система может подключать дополнительное оборудование для увеличения производительности.

Плохие решения:

- централизованные компоненты (один почтовый сервер);
- централизованные таблицы (один телефонный справочник);
- централизованные алгоритмы (маршрутизатор на основе полной информации).

Только децентрализованные алгоритмы со следующими чертами:

- ни одна машина не имеет полной информации о состоянии системы;
- машины принимают решения на основе только локальной информации;
- выход из строя одной машины не должен приводить к отказу алгоритма;
- не должно быть неявного предположения о существовании глобальных часов.

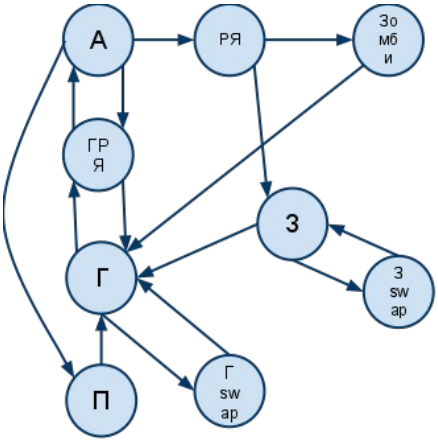
Отличие распределённой ОС от сетевой

В сетевой операционной системе пользователи знают о существовании многочисленных компьютеров, могут регистрироваться на удаленных машинах и

копировать файлы с одной машины на другую. Каждый компьютер работает под управлением локальной операционной системы и имеет своего собственного локального пользователя. Сетевые операционные системы несущественно отличаются от однопроцессорных операционных систем. Ясно, что они нуждаются в сетевом интерфейсном контроллере и специальном низкоуровневом программном обеспечении, поддерживающем работу контроллера, а также в программах, разрешающих пользователям удаленную регистрацию в системе и доступ к удаленным файлам.

Распределенная операционная система, напротив, представляется пользователям традиционной однопроцессорной системой, хотя она составлена из множества процессоров. При этом пользователи не должны беспокоиться о том, где работают их программы или расположены файлы; все это должно автоматически и эффективно обрабатываться самой ОС.

Состояние процесса



Есть два рассказа о состоянии процессов. Простой и сложный. Для сложного используется картинка выше (9 состояний). Для простого - ниже. Симоненко больше любит сложный (а шо поделать).

Итак. Сложный вариант.

П – подготовка. На этом этапе лежат задания. Т.е. программа (что делать) и данные (над чем делать). Заданию ещё не выделили ресурсы. Когда ему планировщик выделит ресурсы, задание станет процессов и перейдёт в готовое состояние.

Г – готовность. Процесс размещен в ОП, ему выделены ресурсы, сформирован РСВ. Так может случиться, что процесс своппируют на диск вместе с его ресурсами (ну, разве что ОП из ресурсов вычеркнут).

Гswap – процесс готов, но своппирован на диске.

ГРя – готовность в режиме ядра. Это некое абстрактное состояние, когда процесс уже имеет ресурсы, но ещё не допущен к процессору. С этого состояния процесс может или получить долгожданный доступ к процессору, или вернуться в очередь готовых.

А – активность. Процесс занял процессор и, собственно, выполняется. Если у процесса окончились выделенные ему кванты времени, он возвращается в ГРя. Если внезапно окончились его ресурсы – в очередь подготовленных. Если произошёл системный вызов – в Ря.

Ря – режим ядра. Сюда попадает процесс, пока выполняется системный вызов (например, ядро открывает файл). Процесс может поступить к заблокированным, или стать зомби.

Зомби – процесса в системе нет, но его PCB ещё есть. Такая ситуация возникает, когда процесс завершился, а породивший его процесс ещё не знает об этом.

Заблокированные процессы находятся в состоянии ожидания. Кому не повезёт – освободят ОП и будут своппированны на диск.

Зswap – процесс с диска можно опять вернуть в ОП в очередь заблокированных.

Если система определила необходимость активизации процесса и выделяет нужные ему ресурсы, кроме времени процессора, то она переводит его в готовое состояние.

Если процессор освободился, то первый (наиболее приоритетный) процесс из очереди готовых процессов получает время процессора и переходит в активное состояние. Выделение времени процессора процессу осуществляет диспетчер. 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 состояние ожидание;
- в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении предусмотренного времени выполнения) его возвращают в состояние готовность
- процесс не выполнился за выделенный ему квант времени, тогда он переходит снова в готовое состояние.

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

Если процесс требует действия или ресурса, которых в данный момент операционная система не может выполнить, он переводится в подготовленное состояние и считается приостановленным.

В общем случае, система должна следить за процессами в очередях готовых и заблокированных процессов, чтобы не было бесконечно ожидающих процессов или процессов, монопольно удерживающих выделенные им ресурсы.

Для любознательных: линк на интуит

Управление процессами, или Всё о PCB

Выполнение функций ОС, связанных с управлением процессами, осуществляется с помощью специальных структур данных, образующих окружение процесса, среду исполнения или образ процесса. Образ процесса состоит из двух частей: данных режима задачи и режима ядра. Образ процесса в режиме задачи состоит из сегмента кода программы, которая подчинена процессу, данных, стека, библиотек и других структур данных, к которым он может получить непосредственный доступ. Образ процесса в режиме ядра состоит из структур данных, недоступных процессу в режиме задачи, которые используются ядром для управления процессом. Оно содержит различную вспомогательную информацию, необходимую ядру во время работы процесса.

Каждому процессу в ядре операционной системы соответствует блок управления процессом (**PCB** – process control block). Вход в процесс (фиксация системой процесса) – это создание его блока управления (PCB), а выход из процесса – это его уничтожение, т. е. уничтожение его блока управления.

Таким образом, для каждой активизированной задачи система создает свой PCB, в котором, в сжатом виде, содержится используемая при управлении информация о процессе.

PCB – это системная структура данных, содержащая определённые сведения о процессе со следующими полями:

1. Идентификатор процесса (имя);
2. Идентификатор родительского процесса;
3. Текущее состояние процесса (выполнение, приостановлен, сон и т.д.);
4. Приоритет процесса;
5. Флаги, определяющие дополнительную информацию о состоянии процесса;
6. Список сигналов, ожидающих доставки;
7. Список областей памяти выделенной программе, подчиненной данному процессу;
8. Указатели на описание выделенных ему ресурсов;
9. Область сохранения регистров;
10. Права процесса (список разрешенных операций);

Ядро ОС

Все операции, связанные с процессами, осуществляются под управлением ядра ОС, которое представляет лишь небольшую часть кода ОС в целом. Ядро ОС скрывает от пользователя частные особенности физической машины, предоставляя ему все необходимое для организации вычислений:

- само понятие процесса, а значит и операции над ним, механизмы выделения времени физическим процессам;
- примитивы синхронизации, реализованные ядром, которые скрывают от пользователя физические механизмы перестановки контекста при реализации операций, связанных с прерываниями.

Ядро ОС содержит программы для реализации следующих функций:

- обработка прерываний;

- создание и уничтожение процессов;
- переключение процессов из состояния в состояние;
- диспетчеризацию заданий, процессов и ресурсов;
- приостановка и активизация процессов;
- синхронизация процессов;
- организация взаимодействия между процессами;
- манипулирование PCB (process control block – см. выше);
- поддержка операций ввода/вывода;
- поддержка распределения и перераспределения памяти;
- поддержка работы файловой системы;
- поддержка механизма вызова-возврата при обращении к процедурам;
- поддержка определенных функций по ведению учета работы машины;

Одна из самых важных функций, реализованная в ядре – обработка прерываний.

Классификация ОС по типу ядра

1. **Монолитное ядро.** Написание ядра системы по принципу “big mess”. Ядро ОС состоит из большого количества отдельных процедур с определенными интерфейсами. Все эти процедуры компилируются и собираются в один большой объектный файл.

Процессор имеет два режима работы: пользовательский и системный. Когда происходит системный вызов, параметры помещаются в соответствующие регистры, процессор переходит в системный режим, находится соответствующая вызову процедура и вызывается.

Процедуры разделены по 3 уровням: Основная процедура (предоставляет интерфейс всем системным вызовам), Сервисные процедуры (реализуют системные вызовы) и Утилиты (используются сервисными процедурами).

2. **Многоуровневая (Кольцевая, Иерархическая).** Схема построения системы по слоям (кольцам). Каждый слой (кольцо), реализует некоторую функциональность и все слои которые находятся над ним, уже не должны заботится о ней. Примером может служить система THE, в которой были следующие слои:

5. Оператор
4. Пользовательские программы
3. Управление вводом-выводом
2. Взаимодействие оператор-процесс
1. Управление памятью и барабаном
0. Выделение процессора и многозадачность

В системе MULTICS была реализована кольцевая структура.

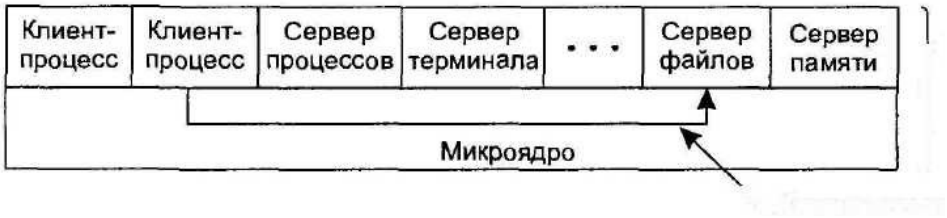
3. **Виртуальная машина.** Этот тип операционных систем берет начало в 70-х годах, когда была очень распространена OS/360, но эта система обеспечивает исключительно пакетную обработку. Поэтому была создана система TSS/370. Эта система создавала несколько виртуальных машин, которые по своим архитектурным возможностям полностью повторяли реальную машину. То есть на каждую из этих виртуальных машин, можно поставить такую же ОС как и на реальную машину.

4. **Экзоядерная.** Это система в которой ресурсы системы разделяются между несколькими виртуальными системами. Но, в отличии от виртуальных машин, в данном случае каждой системе выдается конкретная часть ресурсов. Если одна система

пытается воспользоваться ресурсами, выделенными другой системе, то срабатывает защитным механизм и ее действия пресекаются.

Из конспекта: Многопользовательская система, любой шаг в чужую зону – нарушение безопасности.

5. **Микроядерная (модель клиент-сервер)** – основная идея: сделать ядро как можно меньше и модульней, а большинство функциональности, которая, обычно, реализуется при монолитной архитектуре прямо в ядре вынести в отдельные модули. Ядро делится на несколько модулей, каждый из которых отвечает за отдельную часть общей функциональности (к примеру, управление файлами, процессами, вводом-выводом). Эти модули запускаются в режиме ядра и обеспечивают минимальную достаточную функциональность. Вся остальная функциональность вынесена в модули, запускаемые в пользовательском режиме. Модули ядра называют серверами, а модули пользовательского режима - клиентами. Такая архитектура сравнительно легко преобразуется для работы в распределенных системах.



Многоуровневые системы

1) Языки визуального программирования
2) Языки высокого уровня
3) Уровень ОС
4) Ассемблер высокого уровня
5) Машинный язык
6) Микропрограммный уровень
7) Аппаратура

Дисциплины обслуживания заявок

Классификация дисциплин обслуживания по:

Классификация по приоритетам:

- безприоритетные
- приоритетные.
    - без вытеснения
        - относительные
    - с вытеснением
        - абсолютные
        - выделение равных квантов времени

Если заявка с более высоким приоритетом не прерывает обслуживание заявки с низким приоритетом – относительная ДО без вытеснения, соответственно, если наоборот - абсолютная ДО с вытеснением.

Приоритеты бывают:

- *статические* (сразу задаются заявке и не изменяются в процессе)
- *динамические* (приоритеты меняются в зависимости от t_ожид. или t_обслуж.).

Классификация по количеству очередей

Одноочередные дисциплины

- **FIFO** (первый пришел – первый обслужен). Очередь. Время нахождения в очереди длинных и коротких запросов зависит только от момента их поступления.
- **LIFO** (последний пришел – первый обслужен). Стек.
- **Round Robin** – круговой циклический алгоритм. Запрос обслуживается в течение кванта времени tk. Если за это время обслуживание не завершено, то запрос передается в конец входной очереди на дообслуживание. Короткие запросы находятся в очереди меньшее время, чем длинные.
- **RAND** случайный выбор.

Многоочередные дисциплины

Все новые запросы поступают в очередь 1. Время, выделяемое на обслуживание любого запроса, равно длительности кванта tk. Если запрос обслужен за это время, то он покидает систему, а если нет, то по истечении выделенного кванта времени он поступает в конец очереди i+1. На обслуживание выбирается запрос из очереди i, только если очереди 1,…, i-1 пусты.

Таким образом, длинные запросы поступают сначала в очередь 1, затем постепенно доходят до очереди N и здесь обслуживаются до конца либо по дисциплине FIFO, либо по круговому циклическому алгоритму.

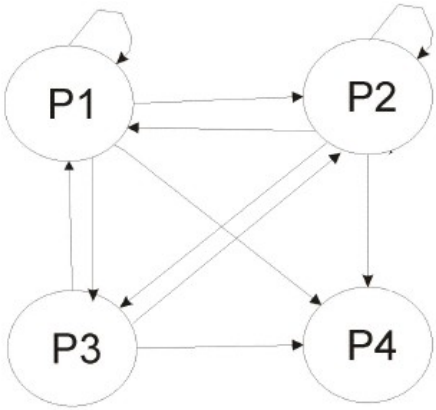
Все остальные многоочередные дисциплины обслуживая - это разные варианты базового алгоритма. Например, время обслуживая может увеличиваться соответственно с номером очереди.

В смешанном алгоритме каждая очередь обслуживается по-своему (FIFO, RR), дисциплины могут чередоваться.

В приоритетных системах каждая очередь отвечает за свой приоритет.

Для любознательных: линк на интуит

Состояние процессора



P1 – обработка системной программы
P2 – обработка программы пользователя
P3 – дешифрация прерывания
P4 – отключение

Структура программ

1) простая структура;
2) структура с запланированным наложением (оверлейная);
3) с динамической последовательной структурой;
4) с динамической параллельной структурой.

Описание каждой структуры:

- Программа с **простой структурой** загружается в основную память и выполняется как отдельный объект. Сегмент программы, помещаемый в основную память, содержит всю программу целиком. (Конкретные команды, выполняемые в этом модуле загрузки, не важны для операционной системы, даже если программа может потребовать услуг операционной системы.)
- Программа со **структурой, в которой запланировано наложение,** создается редактором связей как отдельный модуль загрузки. Однако в нем определяются сегменты программы, которые не должны одновременно присутствовать в основной памяти. Таким образом, одна и та же область в основной памяти может быть повторно использована на основе иерархического построения программы. Этот метод, известный как наложение, требует минимальной помощи от управляющей программы для переноса налагаемого сегмента в основную память.
- Если поведение программы во время их выполнения становится более сложным, появляется тенденция к снижению эффективности и гибкости структур с запланированным наложением. Остальные типы структур программ позволяют использовать при выполнении программы несколько модулей загрузки. В

программе с **динамической последовательной структурой** используется несколько модулей загрузки. Для управления модулями загрузки и установления связей между ними используются следующие четыре макрокоманды:

- LINK (размещение модуля загрузки в основной памяти и последующее его выполнение),
- XCTL, LOAD и DELETE/ Макрокоманда LINK, обеспечивает. Управление вызывающей программе возвращает макрокоманда RETURN, которая освобождает область памяти, занятую модулем, но не перераспределяет ее. Позже, если вновь требуется тот же самый модуль загрузки (и этот модуль загрузки имеет необходимые атрибуты), а его копия до сих пор находится в основной памяти и не повреждена, то он используется повторно без обращения к операции «выборки».
- Макрокоманда XCTL обеспечивает выборку и выполнение модуля загрузки, так же как и макрокоманда LINK-Однако макрокоманда XCTL используется в тех случаях, когда программа выполняется в виде нескольких фаз и выполнение модуля загрузки, содержащего макрокоманду XCTL, заканчивается и больше не возобновляется вновь.
- Макрокоманда LOAD обеспечивает загрузку модуля загрузки, но не его выполнение. В дальнейшем он используется с помощью обычной команды BRANCH (УСЛОВНЫЙ ПЕРЕХОД). Модуль загрузки, загруженный с помощью макрокоманды LOAD, может быть удален из основной памяти.
- Программа с динамической параллельной структурой использует макрокоманду ATTACH для создания подзадачи, которая выполняется совместно с породившей ее задачей. Во всех предыдущих случаях существуют только одна задача и только один блок управления задачей (TCB), при динамической параллельной структуре для каждой выполняемой макрокоманды ATTACH создается дополнительная задача. Каждый модуль загрузки, хранящийся в библиотеке программ, может быть одного из трех типов: однократно используемый, повторно используемый и реентерабельный.
    - Однократно используемый модуль загрузки вызывается из библиотеки всякий раз, когда к нему обращаются.
    - Повторно используемый модуль загрузки является самовосстанавливающемся, так что его команды и константы, измененные при предыдущем выполнении, восстанавливаются перед его повторным выполнением.
    - Реентерабельный модуль загрузки не изменяется в ходе своего выполнения. Системные задачи часто разрабатываются в виде реентерабельных модулей и имеют нулевой ключ памяти.

Модули

Модуль в программировании представляет собой функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом или

поименованной непрерывной его части, предназначенный для использования в других программах. Модули позволяют разбивать сложные задачи на более мелкие в соответствии с принципом модульности. Обычно проектируются таким образом, чтобы предоставлять программистам удобную для многократного использования функциональность (интерфейс) в виде набора функций, классов, констант. Модули могут объединяться в пакеты и, далее, в библиотеки.

Модули могут быть обычными, т. е. написанными на том же языке, что и основная программа, в которой они используются, либо модулями расширения, которые пишутся на отличном от языка основной программы языке. Модули расширения обычно пишутся на более низкоуровневом языке.

Модульное программирование может быть осуществлено, даже когда синтаксис языка программирования не поддерживает явное задание имён модулям.

Программные инструменты могут создавать модули исходного кода, представленные как части групп — компонентов библиотек, которые составляются программой компоновщиком.

Свойства модулей:

- стандартная внутренняя структура
- взаимная независимость
- параметрическая универсальность
- функциональная независимость

Планирование

Статические алгоритмы планирования

Планирование выполняется на другом оборудовании, чем выполнение, сначала создаётся план, потом система его выполняет. Например,

RMS – Rate Monotonic Scheduling. Использует приоритетное вытесняющее планирование. Приоритет присваивается каждой задаче до того, как она начала выполняться. Преимущество отдается задачам с самыми короткими периодами выполнения.

Динамические алгоритмы планирования

Планируются на том же оборудовании, что и выполняется. Должно быть быстрым даже в ущерб качеству планирования. Например,

EDF – Earliest Deadline First Scheduling. Приоритет задачам присваивается динамически, причем предпочтение отдается задачам с наиболее ранним предельным временем начала (завершения) выполнения.

Балансное планирование заключается в стремлении к балансировке нагрузки узлов системы (при перегрузке уменьшается нагрузка на узел за счет миграции процессов).

Планирование бывает долгосрочным (планирование заданий) и краткосрочным (планирование использования процессора). Иногда выделяют среднесрочное планирование (своппинг).

Задачи планирования:

- Справедливость (между пользователями)

- Эффективность – полностью занять процессор
- Сокращение полного времени выполнения (turnaround time)
- Сокращение времени ожидания (waiting time)
- Сокращение времени отклика (response time)

Планирование бывает для

- **однопроцессорных систем** (где 1 процессор и надо распределить *время* между кучей задач). Используются знакомые уже алгоритмы FCFS (FIFO), RR. И, возможно кому-то незнакомые:
    - SJF (Shortest-Job-First) – ставит короткие задачи к началу очереди, в результате чего растет производительность. Бывают вытесняющие и невытесняющие.
    - Гарантированное планирование – каждому пользователю гарантируется равный объем ресурсов.

В перечисленные алгоритмы могут добавляться приоритеты и дополнительные очереди.

Планировщики называются временными.

- **многопроцессорных систем**

Многопроцессорные системы бывают однородные и неоднородные (в неоднородных системах разное оборудование, соответственно одна и та же заявка на разных узлах может быть обслужена с разной скоростью).

В таких системах могут быть явно указаны пары задача-узел (т.е. конкретная задача может решаться только на конкретных узлах), а могут быть не указаны.

Время решения задачи зависит от производительности узла, где она решается, и от времени пересылок между узлами.

Алгоритмы оптимизируют по времени выполнения (так расположить задачи на узлах, чтобы уменьшить время их решения и кол-во пересылок) и по эффективности загрузки системы (чтобы все узлы были более-менее одинаково загружены).

Алгоритмы: венгерский, Шаркара, Янга, ветвей и границ.

Планировщики называются пространственными или пространственно-временными.

Так, братва, выкладываю кое-что по планированию из того, что делал лично я. Если кто любознательный и хочет сверкнуть силой разума перед Симоном - может поможет:

Алгоритмы планирования заданий можно разделить на две основные группы – алгоритмы, основанные на эвристике, и алгоритмы, основанные на случайном поиске. Эвристические алгоритмы можно глобально разделить на три подкатегории – алгоритмы планирования по спискам, алгоритмы кластеризации и алгоритмы дублирования задач.

1.1 Алгоритмы планирования по спискам

Все алгоритмы планирования по спискам создают список всех заданий графа, упорядоченный но их приоритету. Обычно они разделяются на две фазы – фаза выбора задачи, во время которой выбирается задача с наивысшим приоритетом, и фаза выбора процессора, во время которой выбирается такой процессор, который минимизирует

заданную целевую функцию. Среди подобных алгоритмов следует выделить алгоритм Модифицированного Критического Пути (MCP), алгоритм Планирования Динамических Уровней (DLS), алгоритм «Меньшее Время Первое» (ETF), алгоритм Динамического Критического Пути (DCP), а также алгоритм Минимизации Времени Окончания (HEFT). Большинство из подобных алгоритмов рассчитаны на полносвязные однородные системы. Алгоритмы планирования по спискам обычно более производительны и обеспечивают более высокое качество расписаний, чем алгоритмы других категорий.

1.2 Алгоритмы кластеризации

Алгоритмы этого класса присваивают вершины заданного графа неограниченному числу кластеров. На каждом шаге выбираются задачи для кластеризации. На каждой итерации алгоритм улучшает расписание слиянием двух кластеров. Задания, находящиеся в одном кластере, будут выполняться на одном процессоре. Наиболее яркими представителями этой группы можно назвать такие алгоритмы как алгоритм Кластеризации Доминантной Последовательности (DSC), Метод Линейной Кластеризации (LCM), и систему Планирования и Кластеризации (CASS).

1.3 Алгоритмы дублирования заданий

Идея алгоритмов дублирования заданий лежит в том, чтобы при планировании графа приложения некоторые из заданий планировать более чем один раз, что приводит к уменьшению межпроцессорного взаимодействия. Алгоритмы дублирования различаются в зависимости от принятой стратегии дублирования задач. Алгоритмы этой категории обычно предназначены для составления расписаний для неограниченного числа идентичных процессоров и в основном имеют высокую вычислительную сложность.

1.4 Алгоритмы случайного направленного поиска

Алгоритмы случайного направленного поиска используют случайный выбор для направления поиска в пространстве решений. Эти технологии комбинируют знания, полученные на предыдущих шагах с определённым случайным выбором для получения нового результата. Генетические алгоритмы (ГА) являются самыми популярными и широкоиспользуемыми алгоритмами на основе случайного направленного поиска. ГА генерируют хорошие расписания заданий, однако время их работы значительно выше, чем в эвристических алгоритмах. Кроме того, для получения оптимальных результатов необходимо тщательное исследование параметров алгоритма.

Кроме ГА можно отметить еще методы симулирования отжига и методы локального поиска, которые также входят в эту группу.

2. Краткий обзор эвристических алгоритмов для неоднородных систем

В данном разделе представлен краткий обзор двух наиболее известных эвристических алгоритмов статического планирования для неоднородных систем – алгоритм Нормированного - Минимального Времени и алгоритма минимизации времени окончания.

2.1 Алгоритм Нормированного - Минимального Времени

Это двухэтапный алгоритм. На первом этапе задания группируются на основе параметра «уровень», т.е. в одну группу попадают задания, которые могут выполняться параллельно. На втором этапе алгоритм присваивает каждую задачу самому быстрому из доступных процессоров. Задание в более низком уровне имеет более высокий приоритет,

чем задание в более высоком уровне. В пределах уровня наивысший приоритет имеет задание с наибольшим временем выполнения. Каждое задание присваивается такому процессору, который минимизирует сумму времени вычисления задания и суммарного времени взаимодействия задания с заданиями из предыдущего уровня. Алгоритм имеет временную сложность O(v2*q2) где v – количество вершин а q – количество процессоров.

2.2 Алгоритм Минимизации Времени Окончания (HEFT)

Данный алгоритм также является представителем класса алгоритмов планирования по спискам. Алгоритм разделяется на две части. На первом этапе каждой задаче присваивается приоритет, на втором этапе каждая вершина из списка присваивается такому процессору, который минимизирует время окончания её выполнения. Есть также разновидность этого алгоритма, которая известна как разновидность Критического Наследника (CC modification). В качестве критического наследника для текущей вершины выбирается такая вершина, которая является наследницей для текущей и имеет наивысший рейтинг (приоритет). Текущая вершина присваивается такому процессору, который минимизирует время завершения критического наследника текущей вершины. Очевидно, такой алгоритм имеет большую временную сложность, поскольку на каждом шаге производится оптимизация на шаг вперед.

По ряду исследований этот алгоритм даёт лучшие результаты планирования при фиксированной временной сложности и служит некоторым эталоном для разработчиков алгоритмов планирования. Поэтому именно этот алгоритм был выбран для реализации в представленной модели.

Для любознательных: линк1 и линк2, а так же вспомните, кто делал, курсач.

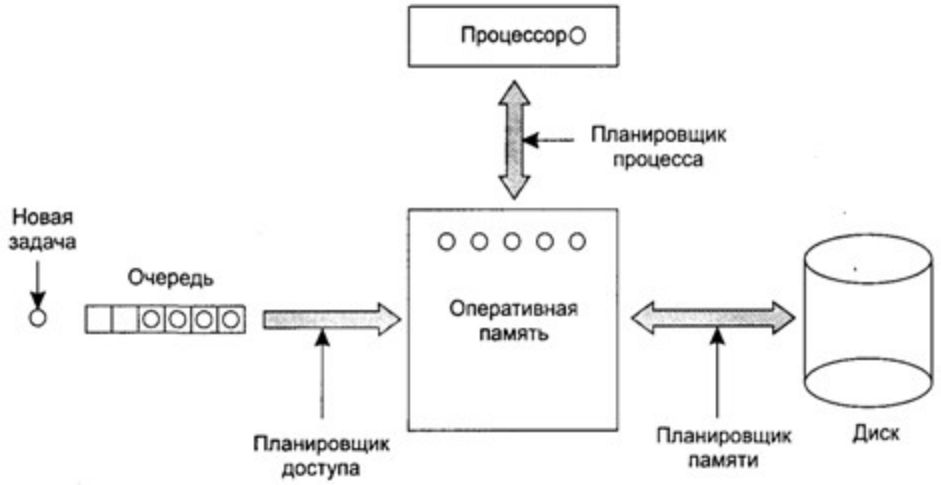
Планировщики

В однопроцессорной системе есть три уровня планировщиков:

- 1 уровень. планировщик заданий, для долгосрочного планирования процессов (отвечает за прохождение новый процессов в систему и следит за количеством процессов, одновременно находящихся в системе)
- 2 уровень. промежуточный или среднесрочный планировщик (по сути это своппинг, то есть перекачка всего (или части) процесса на диск (или с диска)).
- 3 уровень. планировщик использования процессора, для краткосрочного планирования процессов (выбор готового процесса и перемещение его в разряд активных, в следствии того, что предыдущий процесс обратился к устройству ввода/вывода или у него закончился квант времени)

Структура прохождения задачи через ВычСистему

Описана трёхуровневая модель.



1 уровень. Задание (см. определения выше) кто-то формирует. Программист или система. В первом случае, если задание правильно оформлено (синтаксис), оно накапливается в очереди входных заданий.

Для выбора заданий из очереди может использоваться алгоритм, в котором устанавливается приоритет коротких задач перед длинными. Впускной/входной планировщик волен придержать некоторые задания во входной очереди, а пропустить задание, поступившее позже остальных.

Т.е. на 1 уровне происходит фильтрация заданий, но ресурсы к ним не определяются. В задании должна быть указана программа и данные. В результате, имеем очередь входных заданий.

2 уровень. Если в системе есть свободные ресурсы (см. определение выше), то всплывает планировщик второго уровня. Он выбирает из очереди задание, определяет, можно ли его решить с помощью ресурсов, что есть в системе. Если можно, то вызывает *инициализатор/инициатор*, который как только определит ресурсы и расшифрует задание, сформирует блок управления задачей/процессом (*PCB*).

Так исторически сложилось, что сначала был TCB (task control block), а потом появился многопрограммный режим работы, появились процессы и вместе с ними PCB.

Таким образом, задание превращается в задачу/процесс. Возможна ситуация, когда процессов слишком много и они все в памяти не помещаются, тогда некоторые из них будут выгружены на диск. Второй уровень планирования определяет, какие процессы можно хранить в памяти, а какие — на диске. Этим занимается *планировщик памяти*.

Для оптимизации эффективности системы планировщик памяти должен решить, сколько и каких процессов может одновременно находиться в памяти. Кол-во процессов, одновременно находящихся в памяти, называется *степенью многозадачности.*

Планировщик памяти периодически просматривает процессы, находящиеся на диске, чтобы решить, какой из них переместить в память. Среди критериев, используемых планировщиком, есть следующие:

1. Сколько времени прошло с тех пор, как процесс был выгружен на диск или загружен с диска?
2. Сколько времени процесс уже использовал процессор?
3. Каков размер процесса (маленькие процессы не мешают)?

4. Какова важность процесса?

3 уровень. Как только процессор освобождён (либо естественно либо с вытеснением), срабатывает планировщик 3-го (верхнего уровня, он же *планировщик процессора*) и из готовых процессов/задач он должен выбрать процесс/задачу для выполнения и занять время выполнение в процессоре.

4 уровень. Собственно, его можно не выделять отдельно. На последнем этапе результаты выполнения могут быть сохранены или выведены на внешний носитель.

Планирование в параллельных системах

Для планирования в параллельных системах добавляется *транспортный планировщик*. Он служит для распараллеливания заданий, синхронизации процессов по данным – обеспечения поступления требуемых данных и поддержки связей между вычислительными узлами при реализации связи по данным.

к ПП добавляется функция *адаптирования*, при выполнении которой задания распределяются соответственно особенностям данной системы (например: специальные схемы для систем гиперкуб, транспьютерных систем или дополнительная схема для неоднородной среды).

к ПН добавляется функция балансирования в случае реконфигурации (отказа некоторых элементов) системы.

В результате мы приходим к семиуровневой модели.

Семиуровневая модель

Полную систему планирования в этом случае можно представить в виде семиуровневой модели, где каждый уровень часто рассматривают как отдельную задачу. Порядок выполнения уровней может быть изменен в зависимости от особенности системы и целей задачи планирования.

1. (низкий уровень) Предварительное (входное) планирование исходного потока заявок (задача фильтрации)

2. (низкий уровень) Структурный анализ взаимосвязи входного потока заявок по ресурсам с определением общих ресурсов (Анализ)

3. (транспортный) Структурный анализ заявок и определение возможности распараллеливания (Задача распараллеливания)

4. (промежуточный) Адаптация распределения работ соответственно особенностям ВС (Задача адаптирования)

5. (промежуточный) Составление плана – расписания выполнения взаимосвязанных процедур. Оптимизация плана по времени решения и кол-ву ресурсов (Задача Оптимизации)

6. (промежуточный) Планирование потока задач претендующих на захват времени процессора на каждый процессор – задача распределения.

7. (высокий) Выделение процессорного времени, активизация задач. Перераспределение работ в ВС, при отказе оборудования (задача распределения – перераспределения).

Загрузчик программы в ОП

Загрузчик – программа, которая подготавливает объектную программу к выполнению и инициирует ее выполнение.

Более детально **функции** загрузчика следующие:

- **распределение:** выделение места для программ в памяти. Для размещения программы в ОП должно быть найдено и выделено свободное место в памяти. Для выполнения этой функции загрузчик обычно обращается к ОС, которая выполняет его запрос на выделение памяти в рамках механизма управления памятью.
- **загрузка:** фактическое размещение команд и данных в памяти. Считывание образа программы с диска (или другого внешнего носителя) в оперативную память.
- **связывание:** разрешение символических ссылок между объектами. (Связывание с динамическими библиотеками) Компоновка программы из многих объектных модулей. У модулей из-за трансляции по одному, адреса процедур и данных не актуальны и не соответствуют друг другу. Загрузчик же "видит" все объектные модули, входящие в состав программы, и он может вставить в обращения к внешним точкам правильные адреса. Загрузчики, которые выполняют функцию связывания вместе с другими функциями, называются связывающими загрузчиками. Выполнение функции связывания может быть переложено на отдельную программу, называемую редактором связей или компоновщиком.
- **перемещение:** настройка всех величин в модуле, зависящих от физических адресов в соответствии с выделенной памятью. Программа разрабатывается в некотором виртуальном адресном пространстве, в котором адресация ведется относительно начала программы. При размещении в памяти программе выделяется свободный участок с реальными адресами. Все величины в программе, которые должны быть привязаны к реальным адресам, должны быть настроены с учетом адреса, по которому программа загружена.
- **инициализация:** передача управления на входную точку программы.

Не обязательно функции загрузчика должны выполняться именно в той последовательности, в какой они описаны.

Виды загрузчиков

- **Абсолютный.** Выполняет только связывание. Как часть загрузчика может рассматриваться редактор связей (хотя он выполняется отдельно) Абс. загрузчик выделяет загружаемой программе память в пределах памяти, выделенной процессу
- **Настраиваемый.** До этапа выполнения программы в модулях описываются векторы переходов (внутри – и внешнемодульные), и константы, которые нужно переопределить. Загрузчик при просмотре этих записей определяет абсолютные адреса настраиваемых величин.
- **Непосредственно связывающий.** Динамически выполняет все 4 функции. Выделяет память постранично, выполняет связывание программы динамически, по мере необходимости. Для этого к модулю прикрепляются 2 таблицы: ESD и RLD (таблицы векторов переходов и адресных констант).

Редактор связей

Редактор связей выполняет только функцию связывания – сборки программы из многих объектных модулей и формирование адресов в обращениях к внешним точкам. На выходе редактора связей мы получаем загрузочный модуль.

Загрузчик ОС. Схема загрузки ОС

Загрузчик BIOS

Выполняет:

- инициализацию основных компонентов материнской платы;
- обслуживает системные прерывания (как аппаратные, так и программные);
- из Main Boot Record считывает первые 512 байт (бутовый/начальный загрузчик) в ОП. Передает ему управление.

Бутовый (первичный) загрузчик

Определяет активный раздел. Обращается к месту на жестком диске где записан основной загрузчик (обычно к 0 разделу 0 дорожки) и загружает основной загрузчик в память.

Основной (вторичный) загрузчик

Основной загрузчик системы инициализирует некоторые из подсистем (система ввода/вывода => файл конфигурации). Он может загружать несколько операционных систем, давая пользователю выбирать, какую из систем нужно загружать в конкретном случае. После выбора загружаемой системы, загрузчик проводит необходимые приготовления (к примеру, переводит процессор в защищенный режим работы) и начинает загрузку частей ядра в ОП (программы обработки прерываний, управление памятью, управление процессами). После этого загрузчик передает управление ядру (при этом возможна передача параметров. К примеру, при загрузке Linux ядру можно передавать настройки графического и режима и другие параметры).

После формирования ядра начинает работу программа инициализации системы. Подгружается командный интерпретатор (он грузится последним потому, что мы можем указать свой собственный интерпретатор).

Библиотеки

Понятие библиотек появилось еще в первом поколении машин. Но “настоящие” библиотеки появились при разработке компиляторов.

Структура:

- библиотека
- каталог
- управляющая программа

Динамические библиотеки

Часть основной программы, которая загружается в ОС по запросу работающей программы в ходе её выполнения (Run-time), т.е. динамически (Dynamic Link Library, DLL в Windows). Один и тот же набор функций (подпрограмм) может быть использован сразу в нескольких работающих программах, из-за чего они имеют ещё одно название — *библиотеки общего пользования* (Shared Library). Если динамическая библиотека загружена в адресное пространство самой ОС (System Library), то единственная копия может быть использована множеством работающих с нею программ.

При написании программы программисту достаточно указать транслятору, что следует подключить нужную библиотеку и использовать функцию из неё.

Если библиотеки не окажется в системе, программа может не загружаться.

Статические библиотеки

Могут быть в виде исходного текста, подключаемого программистом к своей программе на этапе написания, либо в виде объектных файлов, присоединяемых (линкуемых) к исполняемой программе на этапе компиляции. В результате программа включает в себя все необходимые функции, что делает её автономной, но увеличивает размер. Без статических библиотек объектных модулей (файлов) невозможно использование большинства современных компилирующих языков и систем программирования: Fortran, Pascal, C, C++ и других.

Прерывания

Прерывание — сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается и управление передаётся обработчику прерывания, который реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

Классы прерываний

В зависимости от источника возникновения сигнала прерывания делятся на:

- **асинхронные или внешние** (аппаратные) — события, которые исходят от внешних источников (например, периферийных устройств) и могут произойти в любой произвольный момент: сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши. Факт возникновения в системе такого прерывания трактуется как *запрос на прерывание* (англ. *Interrupt request,* **IRQ**);
- **синхронные или внутренние** — события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода: деление на ноль или переполнение, обращение к недопустимым адресам или недопустимый код операции;
- **программные** (частный случай внутреннего прерывания) — инициируются исполнением специальной инструкции в коде программы. Программные прерывания как правило используются для обращения к функциям встроенного программного обеспечения (firmware), драйверов и операционной системы.

Фазы прерывания

Любая особая ситуация, вызывающая прерывание, сопровождается сигналом, называемым *запросом прерывания (ЗП)*. Запросы прерываний от внешних устройств поступают в процессор по специальным линиям, а запросы, возникающие в процессе выполнения программы, поступают непосредственно изнутри микропроцессора.

После появления сигнала запроса прерывания ЭВМ переходит к выполнению *программы-обработчика прерывания*. Обработчик выполняет те действия, которые необходимы в связи с возникшей особой ситуацией. Например, такой ситуацией может быть нажатие клавиши на клавиатуре компьютера. Тогда обработчик должен передать код нажатой клавиши из контроллера клавиатуры в процессор и, возможно, проанализировать этот код. По окончании работы обработчика управление передается прерванной программе.

Эта операция называется *переключением контекста*. При реализации переключения используются слова состояния программы (*PSW*), с помощью которых осуществляется управление порядком выполнения команд. В PSW содержится информация относительно состояния процесса, обеспечивающая продолжение прерванной программы на момент прерывания.

Существуют три типа PSW: текущее, новое и старое PSW. Адрес следующей команды (активной программы), подлежащей выполнению, содержится в *текущем* PSW, в котором также указываются и типы прерываний, разрешенных и запрещенных в данный момент.

В *новых* PSW содержатся адреса, по которым резидентно размещаются программы обработки прерываний. При возникновении разрешенного прерывания, в одном из старых PSW система сохраняет содержимое текущего PSW – адрес следующей команды данного процесса, которая должна выполниться по окончании обработки прерывания и передаче управления данному прерванному процессу (т.е. адрес команды, которая следует за текущей в данном процессе и которая должна была бы выполниться при отсутствии сигнала прерывания). Таким образом, обеспечивается корректное возвращение в прерванный процесс после обработки прерывания.

Время реакции – это время между появлением сигнала запроса прерывания и началом выполнения прерывающей программы (обработчика прерывания) в том случае, если данное прерывание разрешено к обслуживанию.

Время реакции зависит от момента, когда процессор определяет факт наличия запроса прерывания. Опрос запросов прерываний может проводиться либо по окончании выполнения очередного этапа команды (например, считывание команды, считывание первого операнда и т.д.), либо после завершения каждой команды программы.

Первый подход обеспечивает более быструю реакцию, но при этом необходимо при переходе к обработчику прерывания сохранять большой объем информации о прерываемой программе, включающей состояние буферных регистров процессора, номера завершившегося этапа и т.д. При возврате из обработчика также необходимо выполнить большой объем работы по восстановлению состояния процессора.

Во втором случае время реакции может быть достаточно большим. Однако при переходе к обработчику прерывания требуется запоминание минимального контекста прерываемой программы (обычно это счетчик команд и регистр флагов). В настоящее

время в компьютерах чаще используется распознавание запроса прерывания после завершения очередной команды.

Время реакции определяется для запроса с наивысшим приоритетом.

Насыщение системы прерываний

Если запрос на прерывание окажется необслуженным к моменту прихода нового запроса от того же источника, то возникает так называемое насыщение системы прерываний. В этом случае предыдущий запрос прерывания от данного источника будет машиной утрачен, что недопустимо. Быстродействие ЭВМ, характеристики системы прерываний, число источников прерывания и частоты возникновения запросов должны быть согласованы таким образом, чтобы насыщение было невозможным

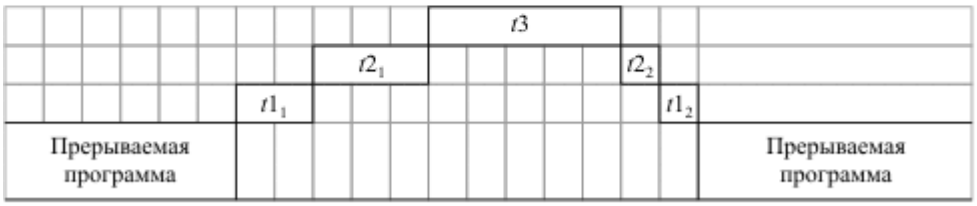
Насыщение системы прерываний возможно при неправильной настройке приоритетов (зацикливание приоритетов), а также при чрезмерном увеличении устройств и процессов, вызывающих прерывания – система постоянно находится в режиме прерывания

Глубина прерывания

Глубина прерывания – максимальное число программ, которые могут прерывать друг друга. Глубина прерывания обычно совпадает с числом уровней приоритетов, распознаваемых системой прерываний. Или же: глубина системы прерывания – это степень вложенности прерываний. Она определяется количеством старых PSW, которые можно поместить в постоянную область памяти или в стек.

Работа системы прерываний при различной глубине прерываний (n) представлена на рис. 14.2. Здесь предполагается, что с увеличением номера запроса прерывания увеличивается его приоритет.

а) $n = 1$



б) $n = 3$

Рис. 14.2. Работа системы прерываний при различной глубине прерываний

Зацикливание прерываний

Это случай когда первая программа прерывает вторую, вторая - третью, третья - первую.

Обработка команд ввода-вывода

Есть три фундаментально различных способа осуществления операций ввода-вывода: программный ввод-вывод, ввод- вывод, управляемый с помощью прерываний, и ввод-вывод, использующий DMA.

Как обрабатывается прерывание ввода/вывода (из шпоры)

1. Обращение к супервизору
2. Сохранение текущего PSW в старый PSW. Выполнение дешифрации прерывания
3. Сохранение нового PSW в текущий
4. Переход по SVC на обработчик прерывания
5. Подготовка операций ввода/вывода
6. В адресное слово канала записывается адрес начала канальной программы
7. Запуск канальной программы
8. Команда SIO
9. Передача ус-вом сигнала об усп./неусп. старте – в слове сост. Канала CSW.
10. Обработчик прерываний запрашивает ответ о успешном старте
11. Ввод-вывод
12. Обработчик анализирует CSW на сигнал о завершении ввода/вывода, записывает старый PSW в текущий

Все. Дальше – продолжение основной программы

II семестр

Организация памяти

Краткий ликбез (проверьте кто-то, могу очень ошибаться):

Совсем давно (во времена 286), был только реальный режим, память была сегментирована, причем все сегменты были по 64к и накладывались друг на друга. Начала сегментов было выровнено на 16 байт.

В 386 появился защищенный режим, который сделал все много более интересным. Сегменты стали переменного размера, при чем один сегмент мог занимать всю память (для 32-разрядных систем это 4 гб). Для описания сегментов были введены дескрипторы сегментов. Дескрипторы хранятся в GDT (Global Descriptor Table). Нужно сказать, что это очень замысловатая таблица и хранятся там совсем не только дескрипторы сегментов. К примеру, еще там могут хранится указатели на LDT (Local Descriptor Table), которые используются для сокрытия некоторых сегментов. Пока дескриптор в GDT - к нему имеют доступ все. В LDT к нему имеет доступ только тот процесс, который данный LDT создал (правда в чем преимущество - не доганяю. Есть идея, что в не захламлении основной таблицы).

И наконец, пришли светлые времена, начали реализовывать виртуальную память. Если раньше, когда не хватало памяти, то на диск свопили целый сегмент (а он может быть ох каким большим), то теперь свопятся только странички. То есть сегменты

остаются, но теперь уже они сами начинают делится на странички. Управлением страничек заведует механизм виртуальной памяти.

Вот такая веселая история. НАПИШИТЕ, ЧТО ЗДЕСЬ НЕ ПРАВИЛЬНО. пожалуйста.

Ссылочка на интуит по теме. Еще одна ссылка по теме

● **плоская (линейная):** состоит из массива байтов, не имеющего линейную структуру; трансляция адреса не требуется, так как логический адрес совпадает с физическим;

● **сегментированная:** состоит из сегментов – непрерывных областей памяти, содержащих в общем случае переменное число байтов; логический адрес содержит 2 части: идентификатор сегмента и смещение внутри сегмента; трансляцию адреса проводит блок сегментации Memory Management Unit (MMU);

● **страничная:** состоит из страниц – непрерывных областей памяти, каждая из которых содержит фиксированное число байтов. Логический адрес состоит из номера (идентификатора) страницы и смещения внутри страницы; трансляция логического адреса в физический проводится блоком страничного преобразования MMU;

● **сегментно-страничная:** состоит из сегментов, которые, в свою очередь, состоят из страниц; логический адрес состоит из идентификатора сегмента и смещения внутри сегмента. Блок сегментного преобразования MMU проводит трансляцию логического адреса в номер страницы и смещение в ней, которые затем транслируются в физический адрес блоком страничного преобразования MMU.

Организация памяти в однопрограммной среде

В данном случае память делится на пользовательскую область и область ОС. Могут быть разные варианты размещения этих областей (например, сначала ОС и потом программа, или наоборот, или драйверы вообще в ПЗУ (BIOS) потом программы, а потом ОС).

Простейшая организация в многопрограммной среде

Если программ множество, то память можно поделить на постоянные блоки (они могут быть как одинаковыми, так и разными, по размеру). Есть множество алгоритмов размещения программ по блокам, к примеру, большим программам – большие блоки, а маленьким – маленькие. Такая система использовалась в OS/360.

Подкачка (swapping)

Основная идея в том, что блоки в памяти переменного размера. Когда программа начинает или возобновляет свою работу, ее считывают (подкачивают) с диска в память. Когда свободного места не хватает, одна или несколько задач сбрасываются на диск (целиком). Здесь стоит заметить, что так как программы могут выделять память динамически, то следует выделять для них пространство несколько большее, чем они реально занимают в данный момент, но, при этом, сбрасывать на диск только актуальную информацию.

При сбросе задач в память и освобождении фрагмента памяти может возникнуть фрагментация, с которой можно бороться уплотнением (сдвигом всех остальных фрагментов к одному из краев). Но оно обычно не производится, так как это длительная операция.

Фрагментация

Фрагментация дискового пространства

Эффект, возникающий в процессе активной работы с файлами (создание, удаление, перемещение, изменение размеров) и выражающийся в отсутствии на жёстком диске достаточного количества последовательных свободных блоков. На фрагментированном диске свободные блоки разбросаны по всей поверхности диска.

Фрагментация данных

Возникает в результате фрагментации дискового пространства: так как на диске отсутствуют последовательные свободные блоки, то новые файлы невозможно записать целиком в одном месте, их приходится делить на фрагменты и записывать в разных частях диска, что замедляет чтение этих файлов и снижает общую производительность файловой системы. Кроме того, если сам файл или его часть по объёму меньше размера кластера, то оставшееся место в кластере остается неиспользованным -> т.е. возникают пустые места.

Фрагментация памяти (выделяем по конспекту)

- **внутренняя**: потеря части памяти, выделенной процессу, но не используемой им. Память процессу при этом выделялась статическим куском.
- **внешняя**: сумма свободных блоков в памяти позволяет загрузить программу, но программа требует непрерывной области памяти, и не влазит.
- **страничная**: память процессу выделяется разным кол-вом страниц, и если процесс занимает не все - остаются свободные места. Идея, сходная с внутренней фрагментацией, но более современная. Иногда для уменьшения фрагментации уменьшают размер страницы, тем самым увеличивая общее кол-во страниц. Тогда таблица страниц, которая содержится в памяти, сильно разрастается.

Методы учета свободного пространства в памяти

Битовая карта

Вся память делится на блоки фиксированного размера. Составляется битовая карта, в которой 0 бит означает, что соответствующий блок свободен, а 1 бит – занят. Здесь нужен поиск компромисса при выборе размера блока – чем больше размер блока, тем меньше битовая карта, но тем больше риск, что в блоке останется неиспользуемое свободное пространство. Чем меньше размер блока – тем больше размер битовой карты.

Связные списки

ОС хранит связный список описателей областей памяти, в каждом описателе записывается свободен ли данный блок, адрес его начала и размер. Список может быть как одно так и двухсвязным, работа с ним тривиальна.

Стратегии поиска места размещения

Когда запрашивается некоторый объём памяти, есть несколько стратегий поиска подходящих фрагментов. К примеру:

- **первый подходящий**: можно идти по свободным блокам и искать первый подходящий блок (размер которого больше запрошенного),
- **наиболее подходящий**: искать блок, размер которого наиболее близок к запрашиваемому (при этом остается много маленьких свободных участков – фрагментация)
- **наименее подходящий**: поиск блока, который наиболее далек по размеру (остается большой свободный участок, который с большой вероятностью еще можно будет использовать).

Способы организации оперативной памяти

Страничная организация

В наиболее простом и наиболее часто используемом случае страничной ВП она (и ОП) представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальные адреса делятся на страницы (page). Соответствующие единицы в ОП образуют страничные кадры (page frames). В целом, система поддержки страничной ВП памяти называется пейджингом (paging).

Передача данных между памятью и диском всегда осуществляется целыми страницами.

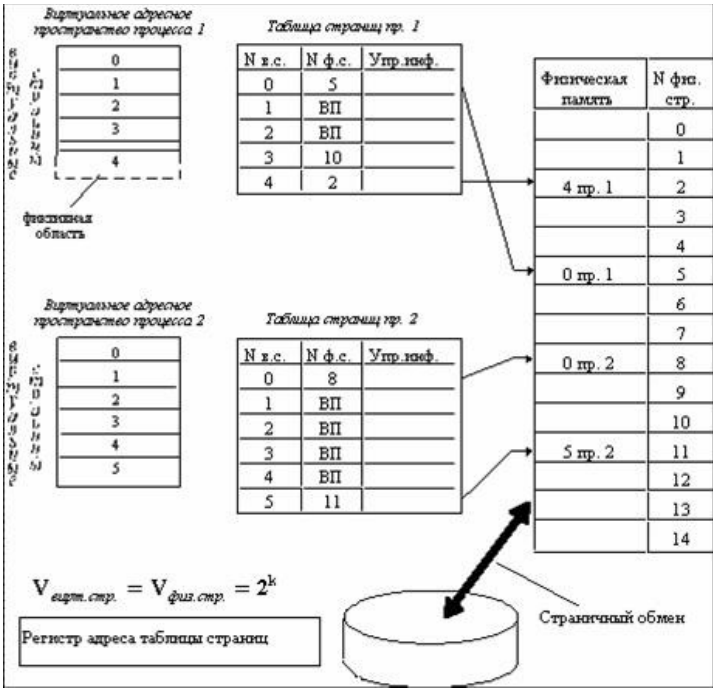
Выполняемый процесс обращается по виртуальному адресу v = (page, shift).

Механизм отображения ищет номер страницы page в таблице отображения и определяет, что эта страница находится в страничном кадре n, формируя реальный адрес из n и s

ОС поддерживает одну или несколько таблиц страниц для каждого процесса. При использовании одной таблицы страниц для ссылки на нее обычно используется специальный регистр. При переключении процессов диспетчер должен найти его таблицу страниц, указатель на которую, таким образом, входит в контекст процесса.

Формирование адреса более наглядно на картинке.

Как программы делятся на страницы. Нумерация страниц внутри программы и глобально в системе.



Виртуальная память

Виртуальная память позволяет ОС использовать больший объём памяти, чем реально установлено в компьютере. При этом часть программы может находится в реальной ОП, а часть - на диске. При чем, эти части могут меняться местами, если будет необходимость воспользоваться частью программы, которая в текущий момент находится на диске.

В случае использования виртуальной памяти, она делится на блоки фиксированного размера (виртуальные страницы). Когда происходит обращение к памяти, адрес сначала

попадает в Memory Management Unit (MMU). В MMU хранится таблица описателей страниц, в которой записано отображена ли в данный момент вирутальная страница в реальную память, и если отображена, то в какую область памяти. MMU определяет на какую из страниц ссылается адрес. Если соответствующая страница отображена в память, то он преобразует адрес. Если страница не отображена, то происходит page fault, в результате которого соответствующая страница подгружается с диска в память, замещая какую-то из малоиспользуемых страниц. После этого происходит преобразование адреса.

Таблица страниц

При больших объёмах памяти, таблица описателей может занимать очень много, поэтому нецелесообразно хранить всю таблицу. Организовывается иерархия таблиц (обычно 2-3 уровня). То есть первая часть в виртуальном адресе указывает на строку в таблице верхнего уровня, вторая часть – в таблице второго у ровня и, наконец, последняя часть это смещение.

Каждая строка в таблице страниц содержит запись-описатель страницы. Обычно такой описатель содержит следующие поля: номер страничного блока на который отображена данная страницы, бит присутствия (отображена ли она в данный момент), бит защиты (какие операции можно производить над этой страницей), биты изменения и обращения (позволяет определить “грязная” или “чистая” данная страница). Стоит заметить, что адрес страницы на диске, не содержится в описателе, поскольку о нем должна знать ОС, а не аппаратура.

Для ускорения поиска в таблицах, используют аппаратные средства, а именно Translation Lookaside Buffer - TLB. Это ассоциативная память, в которую копируется часть, наиболее интенсивно используемых страниц (а такое множество можно выделить, исходя из принципа локальности). Сначала соответствующая страница ищется в TLB и только в случае если ее там нет, начинается поиск в таблицах страниц.

Алгоритмы замещения страниц

При организации виртуальной памяти, нужно обеспечить механизм вытеснения страниц из реальной оперативной памяти. В теории, невозможно предсказать какие из страниц будут использованы системой в ближайшее время. Исходя из таких размышлений, можно сделать вывод, что нельзя оптимизировать алгоритм замещения страниц, так как мы никогда не знаем какая страница будет использоваться следующей.

Но на практике оказывается, что создать сравнительно эффективные алгоритма выбора страниц для замещения - возможно. Это связанно с так называемым **принципом пространственно-временной локальности**. Этот принцип основывается на таких наблюдениях: программа, в большинстве случаев, в ближайшее время будет использовать текущую или близкую к текущей страницу (к примеру, если она в цикле проходит по элементам массива). Пользуясь этим принципом, разработчики создали достаточно большое количество алгоритмов замещения страниц.

Аппаратная поддержка

В большинстве компьютеров имеются простейшие аппаратные средства, позволяющие собирать некоторую статистику обращений к памяти. Два специальных флага на каждый элемент таблицы страниц:

- **Флаг обращения (R)** автоматически устанавливается, когда происходит любое обращение к странице
- **Флаг изменения (M)** устанавливается, если производится запись в страницу

Чтобы использовать эти возможности, ОС должна периодически сбрасывать эти флаги.

Политики поиска кандидатов на замещение

- **Локальные алгоритмы** распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процессу требуется новая страница в ОП, система удаляет из ОП одну из его страниц, а не из страниц других процессов.
- **Глобальные алгоритмы** замещения выбирают для замещения физическую страницу независимо от того, какому процессу она принадлежит

Оптимальный алгоритм (OPT)

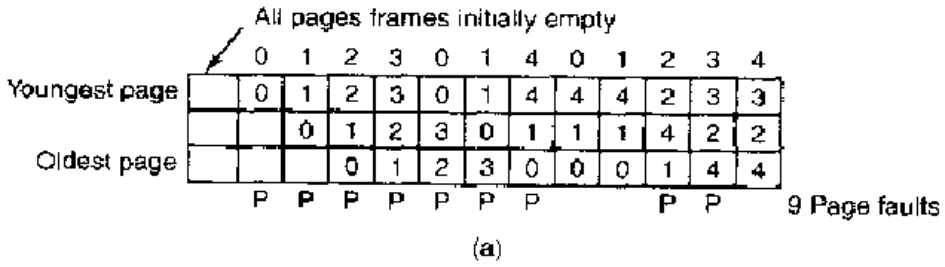
Идеальный алгоритм заключается в том, что бы выгружать ту страницу, которая будет запрошена позже всех. То есть все операции работы с памятью заранее запланированны. На практике не реализуем.

Алгоритм FIFO

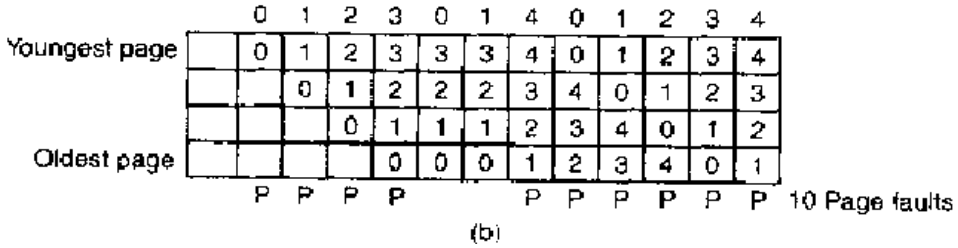
Каждой странице присваивается временная метка. Алгоритм реализуется путем создания очереди страниц, в конец которой страницы попадают при загрузке в ОП, а из начала берутся, когда требуется освободить память. Для замещения выбирается старейшая страница. Стратегия с достаточной вероятностью может приводить к замещению активно используемых страниц.

Аномалия Биледи (Belady) (Аномалия FIFO)

Интуитивно ясно, что чем больше страничных кадров имеет память, тем реже будут иметь место страничные нарушения. Удивительно, но это не всегда так. Как установил Биледи, некоторые последовательности обращений к страницам в действительности приводят к увеличению числа страничных нарушений при увеличении кадров, выделенных процессу.



(a)



(b)

При выборе стратегии FIFO для строки обращений к памяти 012301401234 три кадра (9 faults) оказываются лучше чем 4 кадра (10 faults). Аномалию FIFO следует считать курьезом, иллюстрирующим сложность ОС, где интуитивный подход не всегда приемлем.

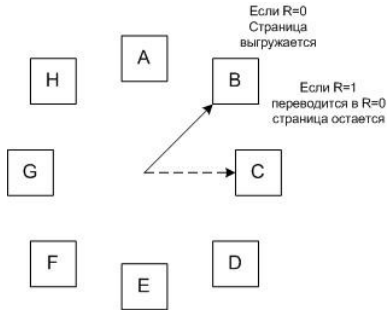
Идея: суть аномалии в том, что в первом случае у нас в память помещается 3 страницы, но в результате есть только 9 страничных промахов. Во втором случае в память помещается 4 страницы, а промахов становится 10. То есть мы увеличиваем объем памяти, а промахов становится больше - в этом и аномалия. То есть обобщая получим: увеличение объема физической памяти, в некоторых случаях приводит к увеличению количества страничных промахов (хотя на интуитивном уровне, кажется, что такое невозможно).

Алгоритм “вторая попытка”

Подобен FIFO, но если R=1, то страница переводится в конец очереди, если R=0, то страница выгружается. В этом случае активно используемая страница не покинет ОП, но каждый раз происходит перемещение страницы.

Алгоритм “часы”

Дескрипторы страниц памяти организовываются в виде закольцованного связного списка. Позволяет не перестраивать очередь. Есть указатель (“стрелка”) текущего кандидата. При страничном промахе, проверяется страница на которую указывает стрелка, если бит R=0 (ее не читали в последнее время), то страница замещается. Если R=1, то R сбрасывается в 0, а стрелка передвигается на следующую страницу.



Алгоритм LRU (Least Recently Used)

Нужно иметь связанный список всех страниц в памяти, в начале которого будут часто используемые страницы. Список должен обновляться при каждой обращении. 64-битный регистр, значение которого увеличивается на 1 после выполнения каждой инструкции. В таблице страниц - поле, в которое заносится значение регистра при каждом обращении к странице. При возникновении page fault выгружается страница с наименьшим значением этого поля.

Алгоритм LFU (Least Frequently Used)

Выталкивается та страница, которая наименее интенсивно используется или обращения к которой наименее часты. Подобный подход опять-таки кажется интуитивно оправданным, однако в то же время велика вероятность того, что удаляемая страница будет выбрана нерационально. Например, наименее интенсивно используемой может оказаться та страница, которую только что переписали в основную память и к которой успели обратиться только один раз, в то время как к другим страницам могли уже обращаться более одного раза.

Алгоритм NRU (Not Recently Used)

(Not Recently Used - не использовавшаяся в последнее время страница)

Используются биты обращения (R-Referenced) и изменения (M-Modified) в таблице страниц. При обращении бит R выставляется в 1, через некоторое время ОС переведёт его в 0 (например, по таймеру). M переводится в 0 только после записи на диск.

Все страницы, в соответствии с битами, разбиваются на 4 класса:

1. не было обращений и изменений (R=0, M=0)
2. не было обращений, было изменение (R=0, M=1)
3. было обращение, не было изменений (R=1, M=0)
4. было обращений и изменений (R=1, M=1)

При страничном промахе замещается страница из наименьшего (по номеру) класса.

Алгоритм рабочего набора

При страничном промахе исследуется время последнего обращения к старинце. Если R = 1 (было обращение), то текущее время становится последним временем использования. Если R = 0, то вычисляется “возраст” страницы, то есть от текущего времени отнимается время последнего использования данной страницы. Если полученный результат > tau, то удаление. Алгоритм считается тяжеловесным.

Сегментная организация

Сегментная адресация памяти — схема логической адресации памяти компьютера в архитектуре x86. Линейный адрес конкретной ячейки памяти, который в некоторых режимах работы процессора будет совпадать с физическим адресом, делится на две части: *сегмент* и *смещение*. *Сегментом* называется условно выделенная область адресного пространства определённого размера, а *смещением* — адрес ячейки памяти относительно начала сегмента. *Базой сегмента* называется линейный адрес (адрес относительно всего объёма памяти), который указывает на начало сегмента в адресном пространстве. В результате получается *сегментный (логический) адрес*, который соответствует линейному адресу *база сегмента+смещение* и который выставляется процессором на шину адреса. (Вики)

Каждая программа использует определенные области памяти для разных задач (как минимум для хранения кода, данных и стека). Удобно разделять адресные пространства, которые используются для хранения разных типов информации. Для такого разделения используются сегменты. Сегмент это абстракция, которая содержит в себе информацию одного типа (например, код программы). Различные сегменты могут иметь разные длины, которые могут меняться динамически (например, сегмент стека).

У каждого сегмента имеются имя (номер), размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия).

Зная тип информации, которая хранится в данной области, можно применять политики безопасности к ней. К примеру, если мы точно знаем, что в определенной области хранится массив целочисленных данных, то при попытке выполнения данного участка, должно возникать сообщение об ошибке.

Сегментация в микропроцессорах Intel основана на использовании двух таблиц: **GDT** (Global Descriptor Table) и **LDT** (Local Descriptor Table). В GDT содержатся дескрипторы сегментов, которые используются системой. LDT может создаваться для каждого процесса и содержит сегменты, которые он использует.

В элементе таблицы сегментов, помимо физического адреса начала сегмента (если виртуальный сегмент содержится в основной памяти), содержится размер сегмента.

Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает прерывание.

Для того что бы загрузить дескриптор используется **селектор**, следующего вида:

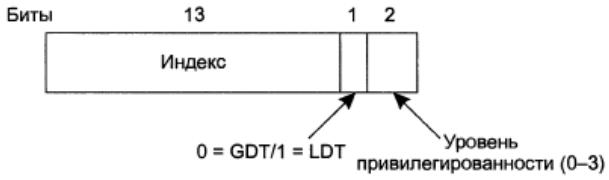
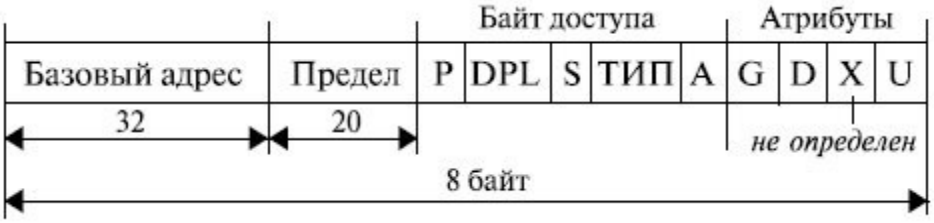


Рис. 4.23. Селектор в системе Pentium

Второй бит которого, показывает в какой из таблиц находится дескриптор (в локальной или глобальной), а следующие 13 - номер дескриптора в таблице.

Дескриптор сегмента имеет следующую структуру:



Базовый адрес – позволяет определить начальный адрес сегмента в любой точке адресного пространства в 2^32 байт (4 Гбайт).

Поле предела (limit) – указывает длину сегмента - 1

G (Granularity – **гранулярность)** – единица измерения длины сегмента (0 – байты, 1 – страницы (4 кб))

Бит размерности (Default size) – определяет длину адресов и операндов, используемых в команде по умолчанию (0 – 16 разрядов, 1 – 32 разряда)

Байт доступа определяет основные правила обращения с сегментом.

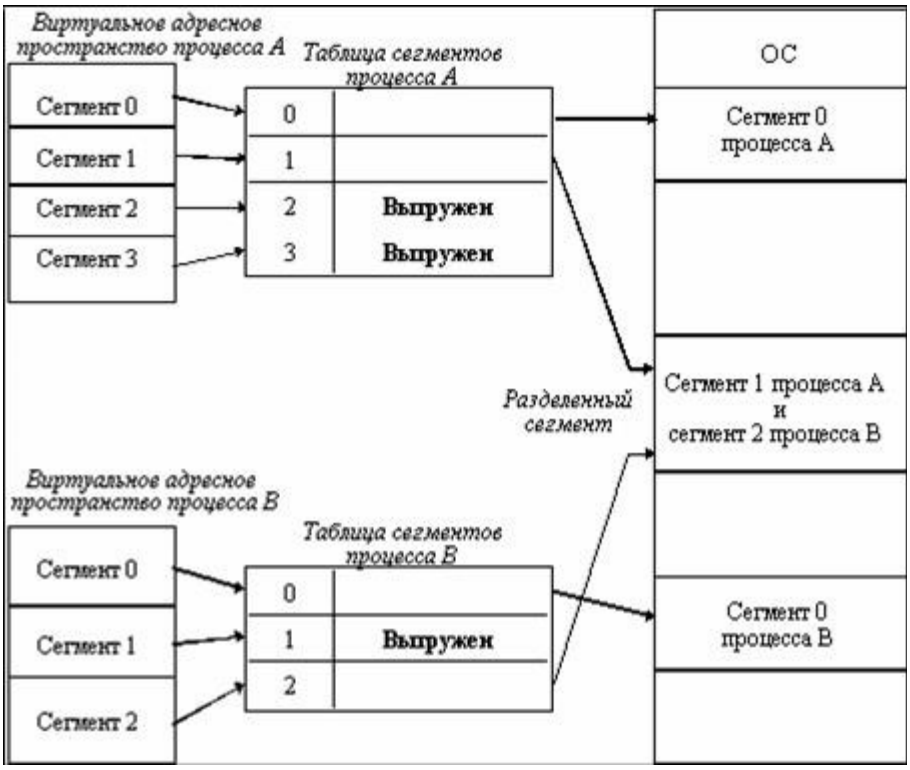
Бит присутствия P (Present) показывает возможность доступа к сегменту (если он равен 1 – сегмент находится в ОП).

Двухразрядное поле DPL (Descriptor Privilege Level) указывает один из четырех возможных (от 0 до 3) уровней привилегий дескриптора, определяющий возможность доступа к сегменту со стороны тех или иных программ (уровень 0 соответствует самому высокому уровню привилегий).

Бит обращения A (Accessed) устанавливается в "1" при любом обращении к сегменту.

Поле типа в байте доступа определяет назначение и особенности использования сегмента. Если бит S (System - бит 4 байта доступа) равен 1, то данный дескриптор описывает реальный сегмент памяти. Если S = 0, то этот дескриптор описывает специальный системный объект, который может и не быть сегментом памяти, например, шлюз вызова, используемый при переключении задач, или дескриптор локальной таблицы дескрипторов LDT.

На картинке показано, как программы разбиваются на сегменты. Разные программы могут использовать один сегмент (в котором, например, содержится динамически подключаемая библиотека).



Когерентность

Когерентность кэша (англ. *cache coherence*) — свойство кэшей, означающее целостность данных, хранящихся в локальных кэшах для разделяемого ресурса. Когерентность кэшей — частный случай когерентности памяти.

Механизмы реализации когерентности могут быть явными и неявными для программиста. Все архитектуры делятся на 4 категории:

1. **Явное размещение и явный доступ.** Все операции над данными явны с точки зрения размещения и доступа. Программист явно задает действия по поддержке когерентности памяти командами Send() и Receive(). Каждый процессор имеет свое собственное адресное пространство (память ВС распределена), а согласованность элементов данных выполняется путем установки соответствия между областью памяти, предназначенной для передачи командой send, и областью памяти, предназначенной для приема данных командой receive, в другом блоке памяти.
2. **Неявное размещение и неявный доступ.** Доступ к данным прозрачен для программиста. Обычно используется в ВС с разделяемой памятью. Оперирование данными происходит на уровне команд "чтение" (load) и "запись" (store) без явного указания адресов. ВС использует единое адресное пространство, легко масштабируется, все операции работы с данными атомарны. Главное – сохранять порядок следования операций и данных, чтоб обновления приходили вовремя.
3. **Неявное размещение и явный доступ.** Используется разделяемое множество страниц памяти на ВУ. При запросе страницы система автоматически обеспечивает согласование (когерентность) пересылки путём пересылки уже запрошенных ранее

страниц не из внешней памяти, а из памяти модулей, имеющих эти страницы. Данные берутся у наиболее поздней версии.

4. **Явное размещение и неявный доступ.** При этом используется технология MEMORY CHANNEL. Каждый компьютер имеет виртуальную память и страницы, разделяемые этим компьютером с другими. Отображение этих страниц имеется во всех остальных компьютерах системы. При этом используется специальная сетевая карта, обеспечивающая взаимодействие память-память. Доступ к удалённым (чужим) страницам выполняется посредством команд чтения (load) и записи (store), как к обычным страницам виртуальной памяти без обращений к ОС и runtime libraries.

Неявная когерентность кеша в однопроцессорной системе

Используется 3 вида памяти:

- кэш-память с прямым отображением (direct-mapped cache)
- частично ассоциативная кэш-память (set-associative cache)
- ассоциативная кэш-память (fully associative cache)

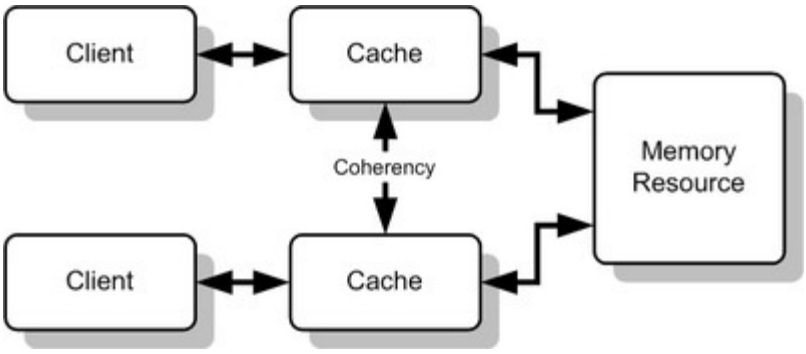
Когерентность осуществляется с использованием отслеживания (*snooping*) запросов на шине, связывающей процессор, память и интерфейс ввода/вывода. Контроллер кэша отслеживает адреса памяти, выдаваемые процессором, и если адрес соответствует данным, содержащимся в одной из строк кэша, то отмечается "попадание в кэш", и данные из кэша направляются в процессор. Если данных в кэше не оказывается, то фиксируется "промах" и инициируются действия по доставке в кэш из памяти требуемой строки.

Могут использоваться такие правила согласования:

- кэши со сквозной записью (изменения переносятся сразу же, как только возникают в кэшах; быстродейственна)
- кэши с обратной записью (смена страниц и dirty page)
- кэши с буферизацией (накапливается в буфере, когда достигает предела, сбрасывается)

Когерентность кеша в многопроцессорных системах

Когда клиенты в системе используют кэширование общих ресурсов, например, памяти, могут возникнуть проблемы с противоречивостью данных. Это особенно справедливо в отношении процессоров в многопроцессорной системе. На рисунке «Несколько кэшей для разделяемого ресурса памяти», если клиент в верхней части имеет копию блока памяти из предыдущего чтения, и нижней клиент изменяет блок памяти, копия данных в кэше верхнего клиента становится устаревшей, если не используются какие-либо уведомления об изменении или проверки изменений. Когерентность кэша предназначена управления такими конфликтами и поддержания соответствия между разными кэшами.



Идентичность данных в кэшах (когерентность кэшей) поддерживается с помощью межмодульных пересылок.

Общая (сосредоточенная) память

MESI (Modified, Exclusive, Shared, Invalid) – один из алгоритмов *неявной* реализации когерентности, поддерживает организацию когерентности кэш-памяти с обратной записью. Предотвращает лишние передачи данных между кэш-памятью и основной памятью. Так, если данные в кэш-памяти не изменялись, то незачем их пересылать. Память сосредоточена и все подключатся к ней через шину. Все транзакции также реализуются только через шину, поэтому есть возможность их отслеживания. Каждая строка в кэшах имеет следующие состояния :

1. M (modified) – строка модифицирована (доступна по чтению и записи только в этом узле, потому что модифицирована командой записи по сравнению со строкой основной памяти);
2. E (exclusive) – строка монопольно копированная (доступна по чтению и записи в этом узле и в основной памяти);
3. S (shared) – строка множественно копированная или разделяемая (доступна по чтению и записи в этом узле, в основной памяти и в кэш-памятях других узлов, в которых содержится ее копия);
4. I (invalid) – строка, невозможная к использованию (строка не доступна ни по чтению, ни по записи)

Используется бит WT, состояние 1 которого задает режим сквозной (write-through) записи, а состояние 0 – режим обратной (write-back) записи в кэш-память.
Основная цель кешированя - минимизация использования общем памяти.
Чтение и запись происходит по следующим правилам:

- Из кеша можно читать любые строки, кроме помеченых I (invalid)
- Запись может производится в строки M и E. Для того что бы записать в S строку, сначала нужно сделать Invalid все остальные копии.
- Кэш может переписывать non-Modified строки новыми. M строки перед переписью нужно записать в память.
- Кэши, которые содержат M строки обязаны отслеживать все обращения через шину, и если обращаются к их M строкам - переписывать их в основную память.
- Кэши, которые содержат S строки должны отслеживать их актуальность.

- Кэши, которые содержат E строки должны отслеживать, до сих пор ли они эксклюзивны.

Распределённая память

Применяют сложные алгоритмы, например, DASH, который заключается следующем. Оперативная память распределена по узлам вычислительной системы. Модуль, где изначально находилась страница, называется резидентным модулем, и сама страница - резидентной. В других модулях эти страницы транзитные. Резидентная страница имеет глобальное состояние, а скопированные страницы имеют локальное состояние. Каждая страница у владельца имеет признак глобального состояния и список адресов копий. Каждый модуль памяти имеет для каждой строки, резидентной в модуле, список модулей, в кэшах которых размещены копии строк.

С каждой строкой в резидентном для нее модуле связаны три ее возможных глобальных состояния:

- "некэшированная", если копия строки не находится в кэше какого-либо другого модуля, кроме, возможно, резидентного для этой строки;
- "удалённо-разделённая", если копии строки размещены в кэшах других модулей;
- "удалённо-изменённая", если строка изменена операцией записи в каком-либо модуле.

Кроме этого, каждая строка кэша находится в одном из трёх локальных состояний:

- "невозможная к использованию";
- "разделяемая", если есть неизмененная копия, которая, возможно, размешается также в других кэшах;
- "измененная", если копия изменена операцией записи.

Дальнейшие действия по алгоритму зависят от глобального и локального состояния читаемой строки. Например, каждый процессор может читать из своего кэша, если состояние читаемой строки "разделяемая" или "измененная". Если строка отсутствует в кэше или находится в состоянии "невозможная к использованию", то посылается запрос "промах чтения", который направляется в модуль, резидентный для требуемой строки.

Репликация данных

Важным вопросом для распределенных систем является репликация данных. Данные обычно реплицируются для повышения надежности и увеличения производительности. Одна из основных проблем при этом — сохранение непротиворечивости реплик. Говоря менее формально, это означает, что если в одну из копий вносятся изменения, то нам необходимо обеспечить, чтобы эти изменения были внесены и в другие копии, иначе реплики больше не будут одинаковыми.

Вопросы, связанные с репликацией

1. как должны расходиться обновления по копиям
2. поддержка непротиворечивости
3. проблема кеширования

Основные проблемы репликации

Первая проблема - **обеспечение взаимного исключения** при доступе к объекту. Существует два решения:

- объект защищает себя сам (к примеру, доступ к объекту совершается при помощи синхронизированных методов в Java - каждому клиенту будет создано по потоку и виртуальная Java-машина не даст воспользоваться методом доступа к ресурсу обоим потокам в один момент времени)
- система защищает объект (ОС сервера создает некоторый адаптер, который делает доступным объект, только одному клиенту в один момент времени)

Вторая проблема - **поддержка непротиворечивости реплик**. Здесь снова выделяют два пути решения:

- решение основанное на осведомленности объекта о том, что он был реплицирован. По сути, обязанность поддерживать непротиворечивость реплик перекладывается на сам объект. То есть в системе нет централизованного механизма поддержки непротиворечивости репликаций. Преимущество в том, что объект может реализовывать некоторые специфичные для него методы поддержки непротиворечивости.
- обязанность поддержки непротиворечивости накладывается на систему управления распределенной системой. Это упрощает создание реплицируемых объектов, но если для поддержки непротиворечивости нужны некоторые специфичные для объекта методы, это создает трудности.

Модели непротиворечивости, ориентированные на данные

По традиции непротиворечивость всегда обсуждается в контексте операций чтения и записи над совместно используемыми данными, доступными в распределенной памяти (разделяемой) или в файловой системе (распределенной).

Модель непротиворечивости (consistency model), по существу, представляет собой контракт между процессами и хранилищем данных. Он гласит, что если процессы согласны соблюдать некоторые правила, хранилище соглашается работать правильно. То есть, если процесс соблюдает некоторые правила, он может быть уверен, что данные которые он читает являются актуальными. Чем сложнее правила - тем сложнее их соблюдать процессу, но тем с большей вероятностью прочитанные данные действительно являются актуальными.

- **Строгая**

Наиболее жесткая модель непротиворечивости называется строгой непротиворечивостью (strict consistency). Она определяется следующим условием: *всякое чтение элемента данных x возвращает значение, соответствующим результату последней записи X.*

Это определение естественно и очевидно, хотя косвенным образом подразумевает существование абсолютного глобального времени (как в классической физике), в котором определение «последней» однозначно. Однопроцессорные системы традиционно соблюдают строгую непротиворечивость, и программисты таких систем склонны рассматривать такое поведение, как естественное. Рассмотрим следующую программу:

a = 1; a = 2; print(a);

Система, в которой эта программа напечатает 1 или любое другое значение, кроме 2, быстро приведет к появлению толпы возбужденных программистов и массе полезных мыслей.

Основная идея: это идеальный случай. То есть система параллельная, но все процессы видят последовательность записей в общий ресурс, в таком порядке, как если бы все процессы выполнялись на однопроцессорной машине.

- **последовательная**

Последовательная непротиворечивость (sequential consistency) — это менее строгая модель непротиворечивости. В общем, хранилище данных считается последовательно непротиворечивым, если оно удовлетворяет следующему условию: *результат любого действия такой же, как если бы операции (чтения и записи) всех процессов в хранилище данных выполнялись бы в некотором последовательном порядке, причем операции каждого отдельного процесса выполнялись бы в порядке, определяемом его программой.*

Это определение означает, что когда процессы выполняются параллельно на различных (возможно) машинах, любое правильное чередование операций чтения и записи является допустимым, но все процессы видят одно и то же чередование операций. Отметим, что о времени никто не вспоминает, то есть никто не ссылается на «самую последнюю» операцию записи объекта. Отметим, что в данном контексте процесс «видит» записи всех процессов, но только свои собственные чтения.

Основная идея: все процессы видят записи в общий ресурс (от любых процессов) в одинаковом порядке, хотя этот порядок может меняться (потому, что вариантов последовательности записей от разных процессов может быть много).

- **причинная**

Модель причинной непротиворечивости (causal consistency) представляет собой ослабленный вариант последовательной непротиворечивости, при которой проводится разделение между событиями, потенциально обладающими причинно-следственной связью, и событиями, ею не обладающими. Если событие B вызвано предшествующим событием A или находится под его влиянием, то причинно-следственная связь требует, чтобы все окружающие наблюдали сначала событие A, а затем B.

Для того чтобы хранилище данных поддерживало причинную непротиворечивость, оно должно удовлетворять следующему условию: *операции записи, которые потенциально связаны причинно-следственной связью, должны наблюдаться всеми процессами в одинаковом порядке, а параллельные операции записи могут наблюдаться на разных машинах в разном порядке.*

Основная идея: Последовательность выполнения задач, которые связанны между собой (к примеру, одна задача пользуется результатами другой) должны быть одинаково видна всем процессам, причем задачи в ней должны идти в последовательности, которую налагает на них зависимость (то есть если сначала выполняется А и его результат передается Б, то все процессы видят последовательность А Б)

- **непротиворечивость FIFO**

В случае причинно-следственной непротиворечивости допустимо, чтобы на различных машинах параллельные операции записи наблюдались в разном порядке, но

операции записи, связанные причинно-следственной связью, должны иметь одинаковый порядок выполнения, с какой бы машины ни велось наблюдение. Следующим шагом в ослаблении непротиворечивости будет освобождение от последнего требования. Это приведет нас к непротиворечивости FIFO (FIFO consistency), которая подчиняется следующему условию: *операции записи, осуществляемые единичным процессом, наблюдаются всеми остальными процессами в том порядке, в котором они осуществляются, но операции записи, происходящие в различных процессах, могут наблюдаться разными процессами в разном порядке.*

Основная идея: Еще более ослабленный вариант причинной непротиворечивости. В данном случае, порядок в последовательности должен сохранятся только в том случае, если операции выполняются на одном процессоре.

- **слабая**

Рассмотрим случай, когда процесс внутри критической области заносит записи в реплицируемую базу данных. Хотя другие процессы даже не предполагают работать с новыми записями до выхода этого процесса из критической области, система управления базой данных не имеет никаких средств, чтобы узнать, находится процесс в критической области или нет, и может распространить изменения на все копии базы данных.

Наилучшим решением в этом случае было бы позволить процессу покинуть критическую область и затем убедиться, что окончательные результаты разосланы туда, куда нужно, и не обращать внимания на то, разосланы всем копиям промежуточные результаты или нет. Это можно сделать, введя так называемую переменную синхронизации (synchronization variable). Переменная синхронизации S имеет только одну ассоциированную с ней операцию, synchronize(S), которая синхронизирует все локальные копии хранилища данных. Напомним, что процесс P осуществляет операции только с локальной копией хранилища. В ходе синхронизации хранилища данных все локальные операции записи процесса P распространяются на остальные копии, а операции записи других процессов — на копию данных P.

Используя переменные синхронизации для частичного определения непротиворечивости, мы приходим к так называемой слабой непротиворечивости (weak consistency).

Основная идея: Если процесс активно работает с репликой, ему, скорее всего, не нужно что бы промежуточные результаты его работы попадали в общий доступ. Поэтому он сначала проводит несколько операций с репликой (блок операций), а после этого вызывает процедуру синхронизации, которая делает актуальной все реплики.

- **свободная**

Слабая непротиворечивость имеет проблему следующего рода: когда осуществляется доступ к переменной синхронизации, хранилище данных не знает, то ли это происходит потому, что процесс закончил запись совместно используемых данных, то ли наоборот начал чтение данных. Соответственно, оно может предпринять действия, необходимые в обоих случаях, например, убедиться, что завершены (то есть распространены на все копии) все локально инициированные операции записи и что учтены все операции записи с других копий. Если хранилище должно распознавать разницу между входом в критическую область и выходом из нее, может потребоваться

более эффективная реализация. Для предоставления этой информации необходимо два типа переменных или два типа операций синхронизации, а не один.

Свободная непротиворечивость (release consistency) предоставляет эти два типа. Операция захвата (acquire) используется для сообщения хранилищу данных о входе в критическую область, а операция освобождения (release) говорит о том, что критическая область была покинута. Эти операции могут быть реализованы одним из двух способов: во-первых, обычными операциями над специальными переменными; во-вторых, специальными операциями. В любом случае программист отвечает за вставку в программу соответствующего дополнительного кода, реализующего, например, вызов библиотечных процедур acquire и release или процедур enter_critical_region и leave_critical_region.

Основная идея: Это модификация слабой непротиворечивости, но теперь есть не одна процедура “синхронизировать”, а две процедуры “вход в критическую секцию” (надо убедится что у меня актуальные данные), “выход из критической секции” (надо разослать всем то, что я понаделал и узнать что понаделали другие). Так же вводятся барьеры.

- **ленивая**

У свободной непротиворечивости имеется такая реализация, как ленивая свободная непротиворечивость (lazy release consistency). При ленивой свободной непротиворечивости в момент освобождения ничего никуда не рассылается. Взамен этого в момент захвата процесс, пытающийся произвести захват, должен получить наиболее свежие данные из процесса или процессов, в которых они хранятся. Для определения того, что эти элементы данных действительно были переданы, используется протокол отметок времени.

Основная идея: Когда процесс выходит из критической секции он не рассылает всем обновления, которые он произвел. Поэтому другие процессы, при входе в критическую секцию должны сами запрашивать у всех остальных изменения.

- **поэлементная**

Еще одна модель непротиворечивости, созданная для применения в критических областях, — поэлементная непротиворечивость (entry consistency). Как и оба варианта свободной непротиворечивости, она требует от программиста (или компилятора) вставки кода для захвата и освобождения в начале и конце критической области. Однако в отличие от свободной непротиворечивости, поэлементная непротиворечивость дополнительно требует, чтобы каждый отдельный элемент совместно используемых данных был ассоциирован с переменной синхронизации — блокировкой или барьером. Если необходимо, чтобы к элементам массива имелся независимый параллельный доступ, то различные элементы массива должны быть ассоциированы с различными блокировками. Когда происходит захват переменной синхронизации, непротиворечивыми становятся только те данные, которые ассоциированы с этой переменной синхронизации. Поэлементная непротиворечивость отличается от ленивой свободной непротиворечивости тем, что в последней отсутствует связь между совместно используемыми элементами данных и блокировками или барьерами, потому при захвате необходимые переменные определяются эмпирически.

Модели непротиворечивости, ориентированные на клиента

Модели непротиворечивости, описанные в предыдущем разделе, ориентированы на создание непротиворечивого представления хранилища данных. При этом делалось важное предположение о том, что параллельные процессы одновременно изменяют данные в хранилище и необходимо сохранить непротиворечивость хранилища в условиях этой параллельности. Так, например, в случае поэлементной непротиворечивости на базе объектов хранилище данных гарантирует, что при обращении к объекту процесс получит копию объекта, отражающую все произошедшие с ним изменения, в том числе и сделанные другими процессами. В ходе обращения гарантируется также, что нам не помешает никакой другой объект, то есть обратившемуся процессу будет предоставлен защищенный механизмом взаимного исключения доступ.

Возможность осуществлять параллельные операции над совместно используемыми данными в условиях последовательной непротиворечивости является базовой для распределенных систем. По причине невысокой производительности последовательная непротиворечивость может гарантироваться только в случае использования механизмов синхронизации — транзакций или блокировок.

Далее мы рассмотрим специальный класс распределенных хранилищ данных, которые характеризуются отсутствием одновременных изменений или легкостью их разрешения в том случае, если они все-таки случатся. Большая часть операций подразумевают чтение данных. Подобные хранилища данных соответствуют очень слабой модели непротиворечивости, которая носит название потенциальной непротиворечивости. После введения специальных моделей непротиворечивости, ориентированных на клиента, оказывается, что множество нарушений непротиворечивости можно относительно просто скрыть.

- **потенциальная непротиворечивость**

Степень, в которой процессы действительно работают параллельно, и степень, в которой действительно должна быть гарантирована непротиворечивость, могут быть разными. Существует множество случаев, когда параллельность нужна лишь в урезанном виде. Так, например, во многих системах управления базами данных большинство процессов не производит изменения данных, ограничиваясь лишь операциями чтения. Изменением данных занимается лишь один, в крайнем случае несколько процессов. Вопрос состоит в том, как быстро эти изменения могут стать доступными процессам, занимающимся только чтением данных. В качестве примера можно привести случай распределенных и реплицируемых баз данных (крупных), нечувствительных к относительно высокой степени нарушения непротиворечивости. Обычно в них длительное время не происходит изменения данных, и все реплики постепенно становятся непротиворечивыми. Такая форма непротиворечивости называется *потенциальной непротиворечивостью (eventual consistency).*

Потенциально непротиворечивые хранилища данных имеют следующее свойство: *в отсутствие изменений все реплики постепенно становятся идентичными.*

Потенциальная непротиворечивость, в сущности, требует только, чтобы изменения гарантированно расходились по всем репликам. Конфликты двойной записи часто относительно легко разрешаются, если предположить, что вносить изменения может

лишь небольшая группа процессов. Поэтому реализация потенциальной непротиворечивости часто весьма дешева.

Основная идея: Если есть большая база данных со множеством реплик, но все в основном читают из нее, а пишет лишь ограниченное количество потоков, то есть изменения вносятся не очень часто. Потенциальная непротиворечивость гарантирует, что если базу никто не будет трогать на запись какое-то время, то в конце концов все реплики станут актуальными.

- **монотонное чтение**

Хранилище данных обеспечивает непротиворечивость монотонного чтения (monotonic-read consistency), если удовлетворяет следующему условию: *если процесс читает значение элемента данных x, любая последующая операция чтения x всегда возвращает то же самое или более позднее значение.* Другими словами, непротиворечивость монотонного чтения гарантирует, что если процесс в момент времени t видит некое значение x, то позже он никогда не увидит более старого значения x.

Основная идея: Если процесс один раз прочитал значение, то при повторном считывании этого значения (даже из другой реплики) ему будет возвращено это же значение, или более позднее значение (которое реально появилось позже, чем первоначально считанное значение).

- **монотонная запись**

Во многих ситуациях важно, чтобы по всем копиям хранилища данных в правильном порядке распространялись операции записи. Это можно осуществить при условии непротиворечивости монотонной записи (monotonic-xmite consistency). Хранилище должно выполнять следующее условие: *операция записи процесса в элемент данных x завершается раньше любой из последующих операций записи этого процесса в элемент x.*

Здесь завершение операции записи означает, что копия, над которой выполняется следующая операция, отражает эффект предыдущей операции записи, произведенной тем же процессом, и при этом не имеет значения, где эта операция была инициирована. Другими словами, операция записи в копию элемента данных X выполняется только в том случае, если эта копия соответствует результатам предыдущей операции записи, выполненных над другими копиями x.

Основная идея: Процесс может совершить операцию записи в реплику, только в том случае если все предыдущие операции записи (возможно от других потоков и совершенные над другими репликами) уже применены к данной реплике.

- **чтение собственных записей**

Хранилище данных обладает свойством непротиворечивости чтения собственных записей (read-your-writes consistency), если оно удовлетворяет следующему условию: *результат операций записи процесса в элемент данных x всегда виден последующим операциям чтения x этого же процесса.*

Другими словами, операция записи всегда завершается раньше следующей операции чтения того же процесса, где бы ни происходила эта операция чтения.

Основная идея: Если процесс записал что-то в реплику, для него эти изменения произойдут моментально и он сразу же сможет ими воспользоваться. (Пример: мы

меняем пароль, новый пароль отправляется на основной сервер, там хешируется и рассылается репликам. Пока до локальной реплики не дойдет новый хеш, процесс использующий эту реплику не сможет использовать новый пароль. Чтение собственных записей подразумевает, что локальную реплику мы обновим немедленно, не дожидаясь пока до нее дойдет обновление от основного сервера)

- **запись за чтением**

Модель, в которой изменения распространяются как результаты предыдущей операции чтения. В этом случае говорят, что хранилище данных обеспечивает непротиворечивость записи за чтением (writes-follow-reads consistency), если соблюдается следующее условие: *операция записи в элемент данных x процесса, следующая за операцией чтения x того же процесса, гарантирует, что будет выполняться над тем же самым или более свежим значением x, которое было прочитано предыдущей операцией.*

Иными словами, любая последующая операция записи в элемент данных x, производимая процессом, будет осуществляться с копией x, которая имеет последнее считанное тем же процессом значение x.

Пример: Непротиворечивость записи за чтением позволяет гарантировать, что пользователи сетевой группы новостей увидят письма с ответами на некое письмо только позже оригинального письма. Поясним проблему. Представьте себе, что пользователь сначала читает письмо A. Затем он реагирует на него, посылая письмо B. Согласно требованию непротиворечивости записи за чтением, письмо B будет разослано во все копии группы новостей только после того, как туда будет послано письмо A.

Размещение реплики

Основную проблему проектирования распределенных хранилищ данных, которую мы должны решить, — это когда, где и кому размещать копии хранилища. Различают три различных типа реплик:

Постоянные реплики

Постоянные реплики можно рассматривать как исходный набор реплик, образующих распределённое хранилище данных. Во многих случаях число постоянных реплик невелико. Рассмотрим, например, web-сайт. Распределение web-сайтов обычно происходит в одном из двух вариантов. В первом варианте распределения файлы, которые составляют сайт, реплицируются на ограниченном числе серверов одной локальной сети. Когда приходит запрос, он передается одному из серверов, например, с использованием стратегии обхода кольца.

Второй способ - использование зеркал, то есть серверов, разбросанных по всей сети (интернету), хранящих постоянные реплики. Обычно сам пользователь выбирает каким зеркалом пользоваться.

Реплики, инициируемые сервером

В противоположность постоянным репликам, реплики, инициируемые сервером, являются копиями хранилища данных, которые создаются для повышения производительности и создание которых инициируется хранилищем данных (его

владельцем). Рассмотрим, например, web-сервер, находящийся в Нью-Йорке. Обычно этот сервер в состоянии достаточно быстро обрабатывать входящие запросы, но может случиться так, что внезапно в течение нескольких дней из неизвестного удаленного от сервера места пройдет поток запросов. (Такой поток, как показывает короткая история Web, может быть вызван множеством причин.) В этом случае может оказаться разумным создать в регионах несколько временных реплик, призванных работать с приходящими запросами. Эти реплики известны также под названием выдвинутых кэшей (push caches).

Основная идея: Если система видит, что множество запросов (больше некоторого порога -- порога создания) приходят из одного региона, она создает реплику и пытается разместить ее как можно ближе к этому региону. Если количество запросов падает ниже некоторого порога (порога удаления), то реплика удаляется (если это не последняя оставшаяся в сети реплика).

Реплики, инициируемые клиентом

Ещё один важный тип реплик — реплики, создаваемые по инициативе клиентов. Инициируемые клиентом реплики часто называют клиентскими кэшами (client caches), или просто кэшами. В сущности, кэш — это локальное устройство хранения данных, используемое клиентом для временного хранения копии запрошенных данных. В принципе управление кэшем полностью возлагается на клиента. Хранилище, из которого извлекаются данные, не делает ничего для поддержания непротиворечивости кэшированных данных. Однако, как мы увидим, существует множество случаев, в которых клиент полагается на то, что хранилище данных известит его об устаревании кэшированных данных.

Основная идея: браузер кеширует сайт погоды, и при открытии браузера мы можем увидеть сайт, ещё не подключившись к инету. В кеше будет старая копия (с прошлого подключения), допустим за 10 декабря. Как только мы (клиент) нажмём ф5, браузер полезет на сервер, выяснит, что данные в его кеше устарели, и обновит их до сегодняшего числа. Если мы не нажмём ф5, и будем пялиться на старые данные - это наши проблемы.

Распространение обновлений

Операции обновления в распределенных и реплицируемых хранилищах данных обычно инициируются клиентом, после чего пересылаются на одну из копий. Оттуда обновление распространяется на остальные копии, гарантируя тем самым сохранение непротиворечивости. Как мы увидим далее, существуют различные аспекты проектирования, связанные с распространением обновлений.

Состояние против операций

Один из важных вопросов проектирования заключается в том, что же именно мы собираемся распространять. Существует три основных возможности:

- распространять только извещение об обновлении;
- передавать данные из одной копии в другую;
- распространять операцию обновления по всем копиям.

Извещения об обновлении Распространение извещения производится в соответствии с протоколами о несостоятельности (invalidation protocols). Согласно протоколу о несостоятельности другие копии информируются об имевшем место обновлении, а также о том, что хранящиеся у них данные стали неправильными. Эта информация может определять, какая именно часть хранилища данных была изменена, то есть какая часть данных перестала соответствовать действительности. Важно отметить, что при этом не передается ничего кроме извещения. Независимо от требуемой для неправильной копии операции, обычно сначала нужно обновить эту копию. Конкретные действия по обновлению зависят от поддерживаемой модели непротиворечивости.

Идея: Рассылаем только извещения о том, что что-то изменилось (сами измененные данные не пересылаем). Преимущество в том, что сеть не нагружается. Эффективно если отношение количества чтений к количеству записей относительно невелико.

Передача обновленных данных между репликами — это вторая альтернатива, она применяется, когда отношение операций чтения к операциям записи относительно велико. В этом случае высока вероятность того, что обновление окажется эффективным в том смысле, что модифицированные данные будут прочитаны до того, как произойдет следующее обновление. Вместо того чтобы распространять обновленные данные, достаточно вести журналы обновлений и для снижения нагрузки на сеть передавать только эти журналы. Кроме того, передачу данных часто можно агрегировать в том смысле, что несколько модификаций можно упаковать в одно сообщение. Это также снизит затраты на взаимодействие.

Идея: Распространяются сами измененные данные. Эффективно в случае, если отношение количества чтений к количеству записей относительно велико (большая вероятность, что обновленные данные кому-то понадобятся и будут прочитаны). Можно распространять не реплики целиком, а только журналы изменений (как в журналируемой ФС) - снижает нагрузку на сеть.

Распространение операций обновления Третий подход состоит в том, чтобы отказаться от переноса модифицированных данных целиком, а указывать каждой реплике, какую операцию с ней следует произвести. Этот метод, называемый также *активной репликацией (active replication)*, предполагает, что каждая реплика представлена процессом, способным «активно» сохранять актуальность своих данных путем выполнения операций над ними. Основной выигрыш от активной репликации состоит в том, что обновления часто удается распространять с минимальными затратами на взаимодействие, определяемыми тем, что параметров, ассоциированных с той или иной операцией, относительно немного. С другой стороны, каждой реплике может потребоваться большая процессорная мощность, особенно если операции окажутся сложными.

Идея: Распространяются не сами новые данные, а операция, которая их породила. Предполагается, что каждая реплика в состоянии сама применить к себе эту операцию. Еще называется *активной репликацией*. Не нагружает сеть, зато нагружает машины на которых находятся реплики.

Продвижение против извлечения

Другой вопрос проектирования состоит в том, следует ли обновления продвигать (push) или извлекать (pull). В протоколах, основанных на продвижении (push-based protocols), известных также под названием серверных протоколов (server-based protocols), обновления распространяются по другим репликам, не ожидая прихода от них запросов на получение обновлений. Подобный подход часто используется в промежутке между постоянными и инициированными сервером репликами, но может применяться также и для передачи обновлений в клиентские кэши. Серверные протоколы применяются, когда в репликах в основном следует поддерживать относительно высокий уровень непротиворечивости. Другими словами, когда реплики должны быть идентичными.

В противоположность им, в подходах, основанных на извлечении (pull-based approach), сервер или клиент обращается с запросом к другому серверу, требуя отправить ему все доступные к этому моменту обновления. Методы, основанные на извлечении, называемые клиентскими протоколами (client-based protocols), часто используются при работе с клиентским кэшем. Так, например, стандартная стратегия, применяемая при работе с кэшами в Web, предполагает предварительную проверку актуальности кэшированных данных. Когда кэш получает запрос на локально кэшированный элемент данных, он обращается к «родному» web-серверу, проверяя, не был ли этот элемент данных модифицирован после его кэширования. В том случае, если модификация имела место, модифицированные данные передаются сначала в кэш, а затем — запросившему их клиенту.

Если модификации не было, клиенту сразу передаются кэшированные данные.

Другими словами, клиент опрашивает сервер в поисках обновлений.

Недостатки и достоинства обоих подходов приводят нас к смешанной форме распространения обновлений, основанной на аренде. *Аренда* (lease) — это обещание сервера определенное время поставлять обновления клиенту. Когда срок аренды истекает, клиент запрашивает у сервера обновления и при необходимости путем извлечения получает от него модифицированные данные. Клиент может также после окончания предыдущего срока аренды запросить у сервера новый, чтобы и далее получать обновления путем продвижения их с сервера. Были выделены следующие три типа аренды (отметим, что во всех случаях до истечения срока аренды обновления продвигаются сервером на клиента):

1. аренда, основанная на «возрасте» элемента данных;
2. аренда, основанная на том, насколько часто клиент требует обновления копии в своем кэше;
3. аренда, основанная на объеме дискового пространства, затрачиваемого сервером на сохранение состояния.

Идея: Мы должны сделать выбор между инициатором рассылки: сервером или клиентом. Если одну реплику читает множество клиентов, то разумнее выбрать сервер: информация будет практически всегда актуальной, так как сервер поддерживает ее актуальность. Если записей много и в разные части данных, то разумнее выбрать клиента, так как он сам будет заботится об актуальности, не перегружая сервер.

Так же есть компромиссный вариант, который реализуется “арендой”: если процессу постоянно нужны обновления, он “арендует” сервер, на некоторый промежуток времени, и

постоянно получает обновления. Если же процесс малоактивен, он не делает “аренды” и поддерживает актуальность своей реплики сам.

Целевая рассылка против групповой

С вопросом о том, как рассылать обновления — продвигая или извлекая, — тесно связан и вопрос о методе рассылки — целевой или групповой. При целевой рассылке (unicasting) сервер является частью хранилища данных и передает обновления на N'других серверов путем отправки Л^- отдельных сообщений, по одному на каждый сервер. В случае групповой рассылки (multicasting) за эффективную передачу одного сообщения нескольким получателям отвечает базовая сеть.

Во многих случаях оказывается дешевле использовать доступные средства групповой рассылки. Предельный случай имеет место, когда все получатели сосредоточены в пределах одной локальной сети и в наличии имеются аппаратные средства широковещательной рассылки (broadcasting). Тогда затраты на передачу одного сообщения путем широковещательной или групповой рассылки не превышают затрат на сквозную (от точки к точке) передачу этого сообщения. В такой ситуации передача обновлений путем целевой рассылки была бы значительно менее эффективной.

Групповая рассылка часто может быть эффективна в сочетании с распространением обновлений путем продвижения. В этом случае сервер, который хочет «продвинуть» обновление на несколько других серверов, просто использует одну группу групповой рассылки. В противоположность этому, при извлечении обновление копии требуется обычно только одному серверу или одному клиенту. В этом случае целевая рассылка — наиболее эффективное решение.

Эпидемические протоколы

Мы уже упоминали, что для многих хранилищ данных достаточно лишь причинной непротиворечивости. Другими словами, в отсутствии обновлений нам достаточно лишь гарантировать причинную идентичность копий. Распространение обновлений в причинно непротиворечивых хранилищах данных часто реализуется при помощи класса алгоритмов, известных под названием эпидемических протоколов (epidemic protocols).

Эпидемические алгоритмы не разрешают конфликтов обновления, для этого требуются индивидуальные решения. Вместо этого они ориентированы на то, чтобы распространить обновления при помощи как можно меньшего количества сообщений. Для этой цели несколько обновлений часто объединяются в одно сообщение, которым затем обмениваются два сервера.

Идея: В эпидемических алгоритмах есть два пути распространения инфекции:

- инфицированный узел пытается заразить всех окружающих (неэффективный, поскольку велика вероятность, что в поисках “кого бы заразить” будем натыкаться на уже зараженных соседей).
- не зараженный узел опрашивает соседей, в надежде заразится (более эффективно)

Эти два пути хорошо совмещать, сначала заражаем некоторое количество серверов (по первому пути), а потом заставляем всех заражаться (так как источник заражения будет на нескольких серверах, то зараза будет распространятся шустро).

“Болтовня”

Еще один способ распространения инфекции: узел пытается заразить соседа. Если сосед был здоров - заражаем и идем дальше заражать. Если сосед заранее был заражен, то с некоторой вероятностью (кубик бросаем) решаем прекращать заражать соседей. Это эффективный способ распространения инфекции, но *не дает гарантии*, что будет инфицированы *все* сервера. Для получения таких гарантий, нужно предпринимать дополнительные меры (комбинировать этот способ с еще каким-то, например).

Удаление при инфицировании

При инфицировании возникает проблема удаления информации. Сервер, удалив у себя часть информации, может еще раз “заразится” этой же информацией, то есть восстановить ее в себе. Решение: сохранять действие удаления, как очередное обновление. Или рассылать по всей сети уведомление о “смерти” части информации.

Подробно про репликацию и непротиворечивость Таненбаум. Распределенные системы.

Средства межпроцессного взаимодействия

Взаимодействие процессов необходимо для:

- передачи данных
- извещений (один процесс извещает другой о событии)
- совместного использования данных (вместо репликаций согласовывать общение через общую/разделяемую память)
- использования общих ресурсов (соревнование за монопольное использования сервисом или ресурсом)

Взаимодействие может быть основано на общих переменных или посылке сообщений. В системе должны быть определены операции ждать и сигнал.

Типы взаимодействия:

- один к одному (известно откуда и куда)
- любой к одному (нужна система именования)
- один ко многим (широковещательное/broadcast)
- любой ко многим
- многие к одному

Сигналы

Изначально применялись как средство сообщения об ошибках. Ресурсоёмки т.к. отправка требует системного вызова, а доставка – прерывания процесса получателя. Малоинформативны. Служат для информирования процесса/-ов о наступлении события.

2 фазы:

1. **генерация сигнала:** указывается место возникновения сигнала (ядро, аппаратные сигналы в процессе при выполнения опред. команды, программные сигналы при опред. состояниях системы); формат сигнала; PID процесса.

Причины отправки сигналов:

- особые ситуации (/0)
- терминальные прерывания (клавиши)
- другие процессы
- уведомление о наступлении события (активизации какого-то процесса, например)
- сигналы для управления текущими или фоновыми заданиями
- сигналы *квоты* (если процесс превышает некую квоту, то ему отправляется сигнал)
- *alarm* (сигналы по счетчику таймера, с их помощью реализуются задержки)

2. **доставка и обработка сигнала**

Доставка – после того, как ядро от имени процесса вызывает системную процедуру (isSign), которая проверяет, существуют ли ожидающие доставки сигналы, адресованные процессу.

Доставка вызывается в 3 случаях:

- при переходе процесса из режима ядра в режим задачи после обработки системного вызова
- при переходе процесса в состояние сна
- сразу при пробуждении с приоритетом, допускающим прерывание сигнала

Если процедура IsSign() обнаруживает сигналы, ожидающие доставки, то ядро вызывает функцию доставки.

Обработка – при получении сигнала процесс может:

- действовать по умолчанию (если у него указано это самое умолчание)
- приостановиться
- завершить своё выполнение
- игнорировать
- отложить на время обработку

Следует аккуратно относиться к блокировке и игнору, особенно, если это сигнал исключительных ситуаций.

В интервале от генерации до доставки или принятия сигнал называется ждущим. Обычно он невидим для приложений, однако доставку сигнала потоку управления можно блокировать. Если действие, ассоциированное с заблокированным сигналом, отлично от игнорирования, он будет ждать разблокирования. У каждого потока управления есть маска сигналов, определяющая набор блокируемых сигналов. Обычно она достается в наследство от родительского потока.

Процесс может обрабатывать только тогда, когда он активен (т.е. выбран планировщиком для выполнения) – иначе, сигнал либо тупо будет висеть хрен знает сколько в ожидании обработки, либо просто не будет обработан.

- ## Семафоры

(вспомним Корочкина) некоторая (защищенная) переменная, значение которой можно считывать и изменять только при помощи специальных операций P и V. Взаимодействие в модели общих переменных. (“Забор для процесса”).

Операция P(S): проверяет значение семафора S; если S>0, то S:=S-1 и процесс может входить в критический участок, если (S=0) ждать.

Операция V(S): проверяет, есть ли процессы, приостановленные по данному семафору; если есть, то разрешить одному из них продолжить свое выполнение (войти в КУ); если нет, то S:=S+1.

Задача взаимного исключения: создать зелёный семафор S := 1, поставить P(S) перед КУ, поставить V(S) после КУ

Задача синхронизации: создать красный семафор S := 0, P(S) в точке ожидания, V(S) в месте сигнала.

Возможны множественные семафоры.

Важно: Операции P и V обязательно должны быть атомарны. ОС запрещает все прерывания на время их выполнения. Так как операции очень маленькие (несколько тактов), это отключение не рискованно.

Недостатки: (вернёмся в суровую реальность Симоненка)

- требует внимания от пользователя по установке всех P и V и инициализации семафоров, программист может забыть о том, какой семафор за что отвечает и т.п.
- если в системе используется планирование по приоритетам, то возможна ситуация “инверсии приоритетов”: низкоприоритетный процесс захватил семафор, но после этого вытеснен высокоприоритетным процессом, который ждет этого семафора, в результате получается блокировка, так как низкоприоритетному потоку больше не дается квантов времени и он не в состоянии освободить семафор
- низкоприоритетные процессы не смогут захватить семафор. Чтоб решить, надо временно повышать приоритет семафору до потолка, пока он не освободит семафор

• Мютексы

Мютекс (сокращение от mutual exception) это частный случай семаформа (а именно – двоичного семафора). Мютекс может быть в одном из двух состояний – захвачен или свободен.

• Мониторы

Монитор – это набор процедур, переменных и других структур данных, объединенных в особый модуль, или пакет. Процессы могут вызывать процедуры монитора, но у процедур, объявленных вне монитора, нет прямого доступа к внутренним структура данных монитора.(“Забор для ресурса”).

При обращении к монитору в любой момент времени может быть активен только один процесс. Обращаться к ОР можно только через спец. процедуру монитора (“вход”). Если в данный момент никакой другой процесс не использует монитор то, процесс входит в монитор, а остальные процессы блокируются при попытке входа. Как только процесс завершил работу с ОР, он выходит из монитора, и в монитор входит первый процесс из очереди (по приоритетам или времени).

Процесс, вошедший в монитор (т.е. выполняющий его процедуру), также может быть блокирован им, если не выполняется некоторое *условие* X для выполнения процесса. С ней связывается некоторая внутренняя(она приоритетнее внешней) очередь процессов, блокированных по условию X.

Находясь в мониторе, процесс может вызвать операцию wait(), тем самым переходя в режим ожидания и освобождая монитор. Для того, что бы продолжить выполнения

процесса, вызвавшего wait(), какой либо другой процесс должен совершить операцию signal() по отношению к данному процессу.

В мониторах может быть разрешено одновременное считывание ОР, т.е. нарушаться основное правило мониторов. Не может быть одновременного считывания и записи.

Мониторы это абстракция языка программирования, следовательно, они должны поддерживаться компилятором (в отличие от семафоров и мютексов, работу с которыми можно организовать библиотечными вызовами, при условии что их поддерживает ОС).

● Каналы

Каналы бывают именованными (любой процесс может посылать инфу или забрать канал) и не-именованными (могут быть созданы только между родственными процессами).

Для работы необходимо создать канал, а потом организовывать работу. Канал имеет вход, выход и буфер. Каналы можно создавать на чтение R и запись W.

При работе с каналами возможны ситуации:

- если считывается меньше, чем находится в канале, то ошибки нет, остаток инфы считывается при след. чтении
- если пытаемся считать больше, чем пересылается в канале, то считываем всё доступное, а дальше ответственность лежит на получателе
- при чтении канал пуст, значит, его никто не открыл на W, возвращается 0 (видимо, тому, кто хочет считать)
- если канал открыт на W, то чтение блокируется до появления данных
- если запись доходит до конца – гарантируется атомарность
- если канал заполнен до окончания записи, то запись блокируется до освобождения места, атомарность не гарантируется.

● Сообщения

Сообщения считаются более информативными, чем сигналы. Адресация сообщений может быть прямой и непрямой.

Основными операциями при пересылке сообщений являются send и receive.

Передача сообщений может быть синхронной (с блокировкой) и асинхронной (без блокировки).

Прямая (непосредственная) адресация

В данном случае при посылке сообщения, явно указывается процесс-получатель. То есть для того, что бы переслать сообщение от P к Q мы должны выполнить:

send(Q, message)

receive(P, message)

Непрямая (косвенная) адресация осуществляется при помощи очередей (т.н. почтовых ящиков), которые являются частью ОС, размещаются в пространстве ядра. Можно создать мультиплексирование сообщений (wtf?).

Очереди организуются в ядре в виде списка, поддерживаются таблицей очередей сообщений. Очередь это структура с указателями на начало, конец, текущее состояние, кол-во сообщений и права доступа. В каждой ячейке очереди находится указатель на содержимое сообщения, размер и кому оно предназначено. Одновременно может быть несколько очередей, поэтому они все поименованы.

Процессу надо знать имя очереди и идентификатор сообщения.

Операции:
send (A, message) – послать сообщение в очередь A;
receive (A, message) – принять сообщение из очереди A.

В обоих случаях можно использовать или не использовать буферизацию данных (очереди). Если буферизация данных присутствует, то процессы не должны ждать друг друга (*асинхронная* передача). То есть процесс посылает сообщение, оно буферизуется и будет принято получателем, когда он будет готов.

Если очереди не используются, то процесс, пытающийся послать сообщение (выполнить send) будет блокирован до тех пор, пока получатель не перейдет в режим получения (дойдет до receive). Такая организация называется *рандеву*. В таком случае работа процессов должна быть синхронизированна (для уменьшения времени ожидания), то есть это *синхронная* передача.

Если очередь ограниченной длины, то при поступлении сообщения, проверятся, влезет ли оно в очередь, если нет – процесс ждёт. Если очередь неограниченна, то процесс никогда не ждёт, но увы, это возможно чисто теоретически.

Очередь реализуется при помощи структуры в памяти, которая определяет права доступа, размер, число процессов, которые её используют. Область разделяемой памяти встраивается в область адресного пространства процесса, который R/W.

Для любознательных: линк на интуит

- ## Сокеты

Межпроцессные взаимодействия на разных машинах (сетевое). Сокет создается клиентом для взаимодействия с сервером в рамках коммутируемого домена, подобно тому, как файлы создаются в рамках файловой системы. Однако в отличие от обычных файлов, сокеты представляют собой виртуальный объект, который существует, пока на него ссылается хотя бы один из процессов.

Сокет связан с определенным номером порта, через который клиент и сервер обмениваются информацией. Сервер, со своей стороны, прослушивает порт с заданным номером и создает для этого серверный сокет.

Сокет имеет интерфейс доступа к коммуникационному домену и описывается дескриптором.

В BSD UNIX реализованы следующие основные типы сокетов:

- *Сокет датаграмм* (datagram socket), через который осуществляется теоретически ненадежная, несвязная передача пакетов.
- *Сокет потока* (stream socket), через который осуществляется надежная передача потока байтов без сохранения границ сообщений. Этот тип сокетов поддерживает передачу экстренных данных.
- *Сокет пакетов* (packet socket), через который осуществляется надежная последовательная передача данных без дублирования с предварительным установлением связи. При этом сохраняются границы сообщений.
- *Сокет низкого уровня (raw* socket), через который осуществляется не-посредственный доступ к коммуникационному протоколу.

Наконец, для того чтобы независимые процессы имели возможность взаи-модействовать друг с другом, для сокетов должно быть определено *пространство имен.*

Имя сокета имеет смысл только в рамках коммуникационного домена, в котором он создан.

Взаимодействие с установлением соединения:

Сервер	Клиент
1. Создание дескриптора сокета, в котором определяется комм. домен, тип сокета и протокол взаимодействия 2. Привязка сокета к порту 3. Приведение готовности канала к принятию сообщения (перевод в режим ожидания сообщения). При этом создается новый дескриптор для принятия сообщений.	1. То же, что сервер п.1 (создание сокета) 2. Установление соединения с сервером 3. Уведомление об установке связи 4. Посылка самого сообщения 5. Уведомление о получении сообщения 6. Отключение и закрытие сокета

При взаимодействии без установления соединения сервер выполняет то же самое, а клиент немного иначе. Нет пунктов 2 и 3.

Тупики и методы борьбы с ними

Взаимная блокировка (*deadlock*) — ситуация в многозадачной среде или СУБД, при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими процессами. (c) Wiki

С тупиками борются в спец. системах, где тупики дорого обходятся, иначе на них забивают (см. пункт игнорирование).

Условия возникновение тупиков

1. условие взаимоисключения (“захватил и не отдаёт” – монопольный захват ресурсов до завершения процесса)
2. условие ожидания и удержания (монопольный захват ресурсов и требование нового)
3. условие неперераспределяемости ресурсов (ресурс нельзя отобрать и отдать другому процессу, пока текущий процесс не завершён)
4. условие кольцевого ожидания (процесс требует ресурс, захваченный следующим по кругу процессом)

Методы борьбы с тупиками

1. Игнорировать тупики

Считать, что маловероятный случайный тупик предпочтительнее, чем неудобные правила, заставляющие пользователей ограничивать число процессов, открытых файлов и т. п. Подходит для пользовательских ОС, обычных систем без заморочек.

2. Предотвратить возникновение тупика

Путем введения жестких правил для процедуры распределения ресурсов. То есть сделать так, чтоб тупик был невозможным – убрать одно из условий возникновения тупика.

2.1 Убрать 2 условие (ожидания и удержания)*.* При активизации процесса дать ему все ресурсы.

Недостатки: Система может работать не эффективно (ненужные ресурсы работают вхолостую), кроме того, программист может неправильно предусмотреть кол-во необходимых ресурсов –> бесконечное откладывание.

2.2 Убрать 1 и 3 условие (взаимоисключение и неперераспределяемость)**.** Если процесс, захватив ресурс, требует новый, а система не может их предоставить, у процесса отбираются все ресурсы. При необходимости процесс должен будет запросить их снова вместе с дополнительными.

Недостатки:

- возможна деградация системы: каждый процесс может несколько раз запрашивать, получать и отдавать ресурсы системе, т.е. система будет выполнять массу лишней работы.
- процесс может потерять всю сделанную работу; Чтоб не терять всё, надо решать задачу модульно (по кускам, судя по всему – тогда потеряем только часть).
- возникает вероятность бесконечного откладывания;
- дискриминация бедных процессов, у которых постоянно отбирают ресурсы.

2.3 Убрать 4 условие (кольцевое ожидание)**.** “Стратегия линейности”. Все ресурсы выстраиваются в заранее определенном порядке (им присваиваются номера, например). Разрешить процессам захватывать ресурсы в порядке возрастания. Такой порядок захвата исключает возможность появления циклов на графе задач и ресурсов, то есть иключает возможность блокировки.

Недостаток: не всегда можно пронумеровать ресурсы таким образом, что бы их порядок подходил всем ресурсам.

3. Избежание (обход) тупика

Путем тщательного анализа каждого запроса на ресурсы и его удовлетворения только в случае, если запрос признан безопасным.

3.1 Алгоритм банкира.

Предусматривает: знание количества ресурсов каждого вида, знание max кол-ва ресурсов для каждого процесса, сколько ресурсов занимает каждый процесс в данный момент.

Система может находится в *надёжном* состоянии и *ненадёжном*:

- система надёжна, если при текущем количестве свободных ресурсов хотя бы один процесс может завершиться до конца;
- система ненадёжна, если ни один процесс не может завершиться до конца.

Идея в том, что каждый процесс требует определённое кол-во ресурсов и гарантирует, что вернёт их все после работы. Система удовлетворяет только те запросы, которые оставляют её в надежном состоянии, и отклоняет остальные.

+ нет необходимости в перераспределении ресурсов и откате процессов назад

- жесткие требования алгоритма:

○ фиксированное кол-во ресурсов

○ постоянное число работающих пользователей

○ распределитель должен гарантированно удовлетворять запросы за конечный период времени

○ клиенты должны гарантированно возвращать ресурсы

○ пользователи должны заранее указать свои максимальные потребности в ресурсах. При динамическом распределении ресурсов трудно оценить максимальные потребности пользователей.

3.2 Просчёт траектории ресурсов

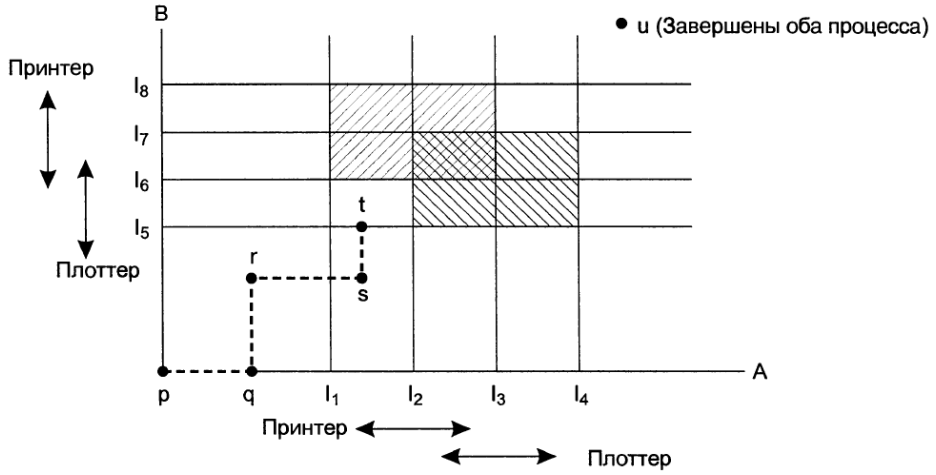


Рис. 3.10. Две траектории ресурсов для процессов

Рассматривается случай для двух процессов и двух ресурсов (принтера и плоттера). Этапы (такты) выполнения процесса A (I1, I2, I3, I4) откладываются на горизонтальной оси, процесса B (I5, I6, I7, I8) на вертикальной.

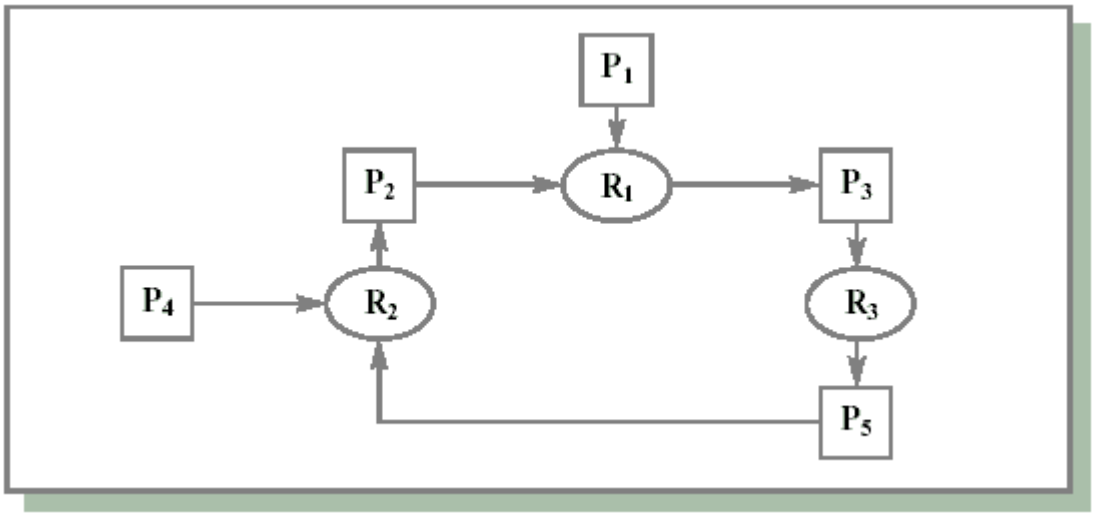
Траектория может двигаться только по горизонтальным и вертикальным линиям, только вверх (если совершается такт процесса B) или вправо (если совершается такт процесса A). Так как процессор только один – не может бить диагональных линий, за один такт выполняется только один процесс.

Области, в которых оба процесса требуют одного и того же ресурса, считаются небезопасными (на рисунке они заштрихованы). Траектории должны обходить небезопасные области. То есть, в ситуации показанной на рисунке, единственным выходом, является временная блокировка процесса B. Он будет блокирован до тех пор, пока процесс A не дойдет до этапа I4 (если мы попытаемся отвернуть траекторию вверх раньше этого момента, то мы обязательно попадем в небезопасную зону).

4. Обнаружение и устранение тупика

4.1 Граф ресурсов. Если система видит, что процесс не завершается за какое-то время, рисуется граф состояний процессов и ресурсов (или же граф распределения

ресурсов). По нему ищутся циклы, если они есть, то система в тупике, тогда определяются процессы, которые тупят.



На картинке процессы P2, P3 и P5 циклично ожидают ресурсов.

Существуют и другие способы обнаружения тупиков, применимые также в ситуациях, когда имеется несколько ресурсов каждого типа.

4.2 редукция графа распределения ресурсов в Дейтеле.

Описание, которое есть в конспекте (возможно, это совсем не Дейтель):

Удаляем из матрицы, построенной по графу, процессы, которые могут быть завершены. Если граф редуцировался до конца, то мы избежали тупика.

4.3 матричный алгоритм в Таненбауме

5. Восстановление системы при обнаружении тупика

Обнаружив тупик, можно вывести из него систему такими способами:

5.1 нарушить одно из условий существования тупика. При этом, возможно, несколько процессов частично или полностью потеряют результаты проделанной работы.

5.2 завершить выполнение одного или более процессов, чтобы впоследствии использовать его ресурсы. Тогда в случае удачи остальные процессы смогут выполняться. Если это не помогает, можно ликвидировать еще несколько процессов. После каждой ликвидации должен запускаться алгоритм обнаружения тупика.

По возможности лучше ликвидировать тот процесс, который может быть без ущерба возвращен к началу.

5.3 временно забрать ресурс у текущего владельца и передать его другому процессу. Возможность забрать ресурс у процесса, дать его другому процессу и затем без ущерба вернуть назад сильно зависит от природы ресурса.

5.4 реализовать в системе средства отката и перезапуска или рестарта с контрольной точки (сохранение состояния системы в какой-то момент времени)

5.5 отбросить процесс в начало выполнения, если его ресурс входит в тупик и освобождение этого ресурса как-то поможет остальным процессам.

6. Тупики в системах спулинга

SPOOLing — simultaneous peripheral operation on-line, то есть имитация работы с устройством в режиме “он-лайн”. Главная задача спулинга — создать видимость параллельного разделения устройства ввода/вывода с последовательным доступом, которое фактически должно использоваться только монопольно и быть закрепленным. Можно каждому вычислительному процессу предоставлять не реальный, а виртуальный принтер и поток выводимых символов (или управляющих кодов для их печати) сначала направлять в специальный файл (*спул-файл,* spool-file) на магнитном диске. Затем, по окончании виртуальной печати, в соответствии с принятой дисциплиной обслуживания и приоритетами приложений выводить содержимое спул-файла на принтер. Системный процесс, который управляет спул-файлом, называется *спулером* (spool-reader или spool-writer).

Идея спулинга в том, что для медленных (а значит, для всех) УВВ делаются входные/выходные буферы данных. И если разным задачам внезапно понадобится доступ к одному и тому же УВВ, они сначала сбрасывают инфу в буфер, а из буфера она идёт на УВВ.

Системы спулинга часто оказываются подвержены тупикам — например, несколько незавершенных заданий, формирующих строки данных и записывающих их в файл спулинга для печати, могут оказаться в тупиковой ситуации, если предусмотренная емкость буфера будет заполнена до того, как завершится выполнение какого-либо задания.

Для восстановления или выхода из подобной тупиковой ситуации мог бы потребоваться перезапуск, рестарт системы с потерей всей работы, выполненной до этого момента. Если система попадает в тупиковую ситуацию таким образом, что у оператора ЭВМ остаются возможности управления, то в качестве менее радикального способа восстановления работоспособности можно предложить уничтожение одного или нескольких заданий, пока у остающихся заданий не окажется достаточно свободного места в буфере, чтобы они могли завершиться.

Когда системный программист производит генерацию операционной системы, он задает размер буферных файлов для спулинга. Один из способов уменьшить вероятность тупиков при спулинге заключается в том, чтобы *предусмотреть значительно больше места для файлов спулинга*. Подобное решение не всегда осуществимо, если память дефицитна. Более распространенное решение состоит в том, что *для процессов входного спулинга устанавливаются ограничения, с тем чтобы они не могли принимать дополнительные задания, когда файлы спулинга начинают приближаться к некоторому порогу насыщения*, например, оказываются заполненными на 75 процентов. Такой подход может привести к некоторому снижению производительности системы, однако это — та цена, которую приходится платить за уменьшение вероятности тупика.

Современные системы являются в этом смысле гораздо более совершенными. Они могут позволять начинать распечатку до того, как завершится очередное задание, с тем чтобы полный или почти полный файл спулинга опустошался полностью или частично уже в процессе выполнения задания. Во многих системах предусматривается динамическое распределение буферной памяти, так что, если отведенного места в памяти оказывается мало, для файлов спулинга выделяется дополнительная память.

Хозяйке на заметку: кроме ресурсных тупиков, рассмотренных выше, есть и другие тупиковые ситуации, например, когда один из процессов ждёт чего-то от другого. Это часто случается при некорректном использовании семафоров, когда один из процессов забывает освободить семафор после окончания пользования ресурсом. При этом другой процесс будет ждать события, которое не наступит -- освобождение семафора.

Для любознательных: вот ссылка на интуит. Но написанное выше довольно полное и пережёванное.

Организация файловых систем

Иерархии данных

Файл – именованная логически связанная совокупность данных,которая с т.з. пользователя представляет собой единое целое.

Логическая иерархия данных

- базы данных / базы знаний
- файловые системы – совокупность файлов
- файл – совокупность записей
- запись – совокупность полей, логическая единица объёма данных, к которым можем обратиться за 1 обращение к памяти
- поле – смысловая единица инфы, различаемая на уровне программы, совокупность идентификаторов (символов)
- символьный набор – внутренний, определяются кодировкой внутри машины; внешний определяет общение между машинами; совокупность байт
- байт (символ) – последовательность бит
- бит – неделимая единица инфы

Физическая иерархия данных

- том – объём инфы, доступный одному устройству чтения/записи (“что можно открутить и унести”). Бывают многофайловые тома и многотомные файлы. Совокупность физ. записей
- запись – объём данных, записываемых/считываемых за 1 раз с диска. Объём физ. и лог. записей может не совпадать (физ. больше). Для связи физ-лог записей используют операции блокирование и разблокирование записи. Запись = кластер, совокупность секторов, это наименьшее место на диске, которое может быть выделено для хранения файла.
- сектор – наименьший физический блок памяти на диске, определённого размера (обычно 512 байт).

Блокирование записей

Размер физической и логической записей не совпадают. Для того что бы связать физическую и логическую записи используют операцию блокирования (для записывания) и разблокирования (для чтения) записи.

Блокировка – при любой записи в файл запись производится в буфер до тех пор, пока он не будет заполнен, тогда весь кластер пишется на диск. Когда закрываем файл, то независимо от заполненности буфера, он скидывается в файл.

При открытии файла выделяется буфер, равный размеру кластера.

При загрузке ОС создается таблица дескрипторов файлов(какому процессу принадлежит файл, буфер файла). Каждому буферу подчинен указатель на текущую позицию в файле.

Разблокирование – считывание кластера целиком, из него выбирается логическая запись.

Физическое устройство ФС

Файловая система – это часть операционной системы, назначение которой состоит в том, чтобы обеспечить пользователю удобный интерфейс при работе с данными, хранящимися на диске, и обеспечить совместное использование файлов несколькими пользователями и процессами.

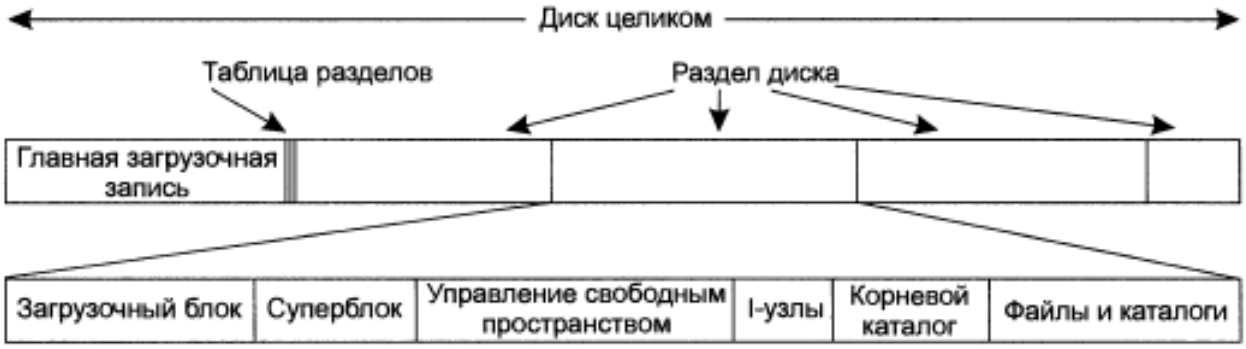


Рис. 5.6. Вариант организации файловой системы

В широком смысле понятие "файловая система" включает:

- совокупность всех файлов на диске,
- наборы структур данных, используемых для управления файлами, такие, например, как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске,
- комплекс системных программных средств, реализующих управление файлами, в частности: создание, уничтожение, чтение, запись, именование, поиск и др.

Для файловой системы важно:

- методы доступа, определяющие организацию доступа к данным
- средства управления ФС
- стратегия распределения свободного пространства
- стратегия поиска файлов
- средства обеспечения целостности данных
- средства управления внешним носителем

Способы повышения производительности ФС:

- **кэширование**. Запросы к внешним устройствам перехватываются промежуточным программным слоем – подсистемой буферизации. Подсистема буферизации представляет собой буферный пул, располагающийся в ОП и комплекс программ, управляющих этим пулом. При поступлении запроса на чтение некоторого блока подсистема буферизации просматривает свой буферный пул и, если находит требуемый блок, то копирует его в буфер запрашивающего процесса. Операция ввода-вывода считается выполненной, хотя физического обмена с устройством не происходило. Очевиден выигрыш во времени доступа к файлу. Для быстрого поиска в кэше используется хэш-таблица.
- **опережающее чтение блока** Получение блоков диска в кэш прежде, чем они потребуются. Многие файлы считываются последовательно. Когда файловая система получает запрос на чтение блока к файла, она выполняет его, но после этого сразу проверяет, есть ли в кэше блок к + 1. Если этого блока в кэше нет, файловая система читает его в надежде, что к тому моменту, когда он понадобится, этот блок уже будет считан в кэш. В крайнем случае, он уже будет на пути туда.

Если обращения к блокам файла производятся в случайном по рядке, опережающее чтение не помогает.

- снижение времени перемещения блока головок (см. оптимизация доступа к диску)
- файловая система с журнальной структурой LFS (очень мутная)

Именования

Правила именования в разных системах различны. У разных систем может быть, а может и не быть чувствительности к регистру.

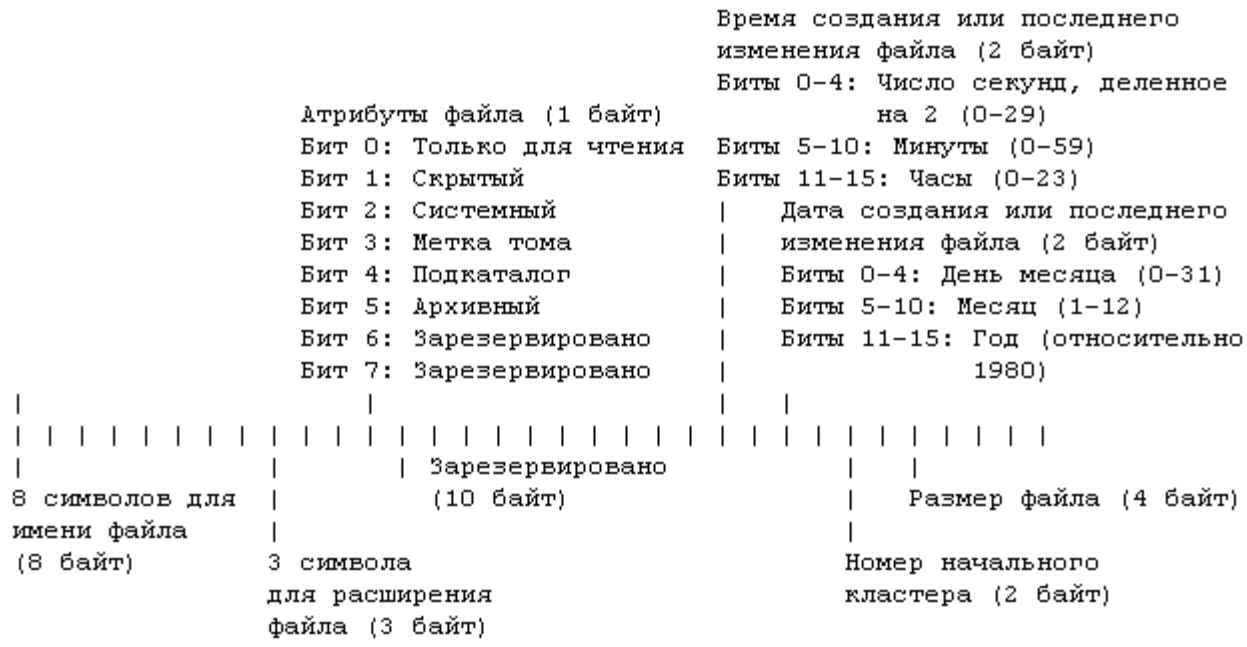
Имя пути

Абсолютное имя – имя всего пути к файлу.

Относительное имя – имя относительно текущего каталога.

Имя файла

Короткие имена по схеме 8.3 (имя.расширение). Смотри картинку внизу.

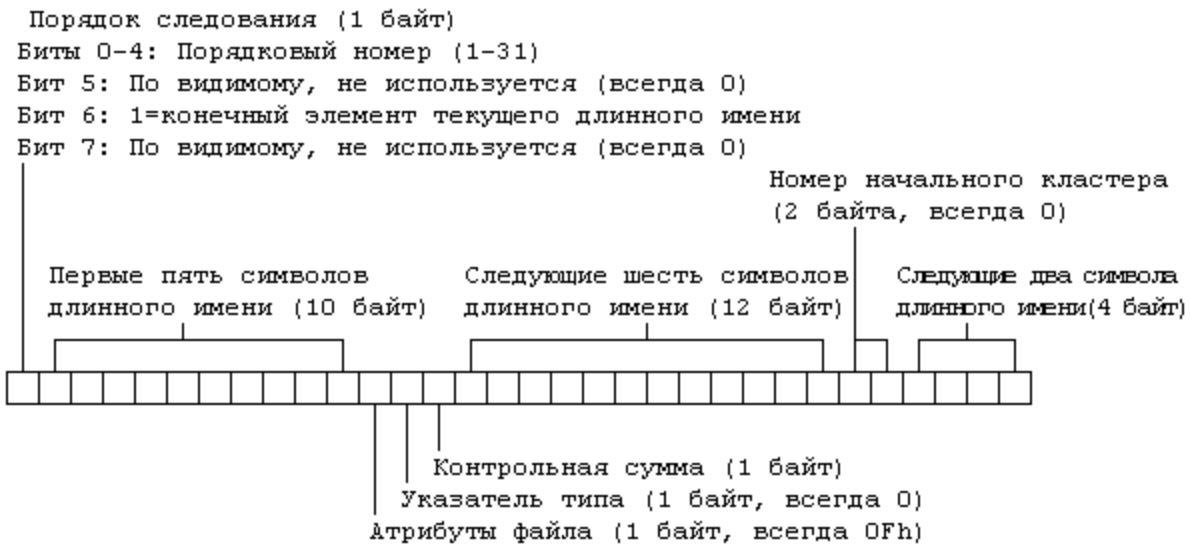


Длинные имена

В Windows 95 поддерживаются имена длиной 255 символов. Для обеспечения обратной совместимости ОС создает его псевдоним, удовлетворяющий стандарту 8.3. Делается это следующим образом.

1. Первые 3 символа после последней точки в длинном имени файла становятся расширением псевдонима.
2. Первые шесть символов длинного имени файла, за исключением пробелов, которые игнорируются, преобразуются в символы верхнего регистра и становятся шестью символами стандартного имени файла. Недопустимые символы (+ , ; = [28]), которые могут использоваться в Windows 95, преобразуются в символы подчеркивания.
3. Добавляются символы ~1 (седьмой и восьмой) к псевдониму имени файла.

Если первые шесть символов нескольких файлов одни и те же, то добавляются символы ~2, ~3 и т.д.



Атрибуты файлов

Атрибуты это служебная инфа в файле. Защита/права доступа (r-w-x), пароль, создатель, имя текущего владельца, флаг доступа (скрытый, временный, архивный), длина записи, позиция ключа, длина ключа, время создания, время последнего изменения, размер файла, максимально допустимый размер.

Директория/Каталог

Каталог – объект в файловой системе, упрощающий организацию файлов. Каталог, находящийся в другом каталоге, называется *подкаталогом* или *вложенным каталогом*. Все вместе каталоги представляют иерархическую структуру в виде дерева каталогов.

Содержит дескрипторы файлов, входящих в него. Атрибуты могут храниться либо в самих дескрипторах (старый вариант), либо в i-узлах.

В системах Linux корневой каталог обозначается как /, в Windows каждый из дисков имеет свою корневой каталог (C:\, D:\ и т. д). На самом деле, в Windows вся информация хранится подобно тому, как это происходит в Linux, доступ к корневому каталогу запрещён.

Операции над файлами и каталогами

Ф&К: создание, удаление, открытие, закрытие, чтение, переименование, поиск, позиционирование;

Ф: запись/изменение данных в файле, получение/установка/удаление/изменение атрибутов, выполнение;

К: добавление/удаление элемента, чтение след. элемента, связывание файла и каталога.

Расположение файлов в ФС:

- **непрерывное расположение** файлов друг за другом, при удалении образуются пустые места, сильная фрагментация. Хотя пока диск не заполнен - высокая

производительность (головки позиционируются только один раз, для первого блока). В настоящее время используется на неперезаписываемых носителях (например, DVD).

- **связные списки** каждая запись имеет адрес след. записи – чтоб получить инфу о записи, надо пройти по всему списку => медленный прямой доступ. Размер полезной информации уменьшается из-за хранения служебной. Размер блока не обязательно кратен степени двойки, а информация, обычно, считывается именно такими блоками. Это приводит к издержкам при копировании.
- **FAT-таблицы** связные списки, но заголовки блоков хранятся в таблице. Таким образом сильно ускоряется прямой доступ. Но в таблицу нужно хранить в оперативной памяти. Обычно дескриптор одного блока занимает 4 байта, то есть таблица получается достаточно объемной и занимает большое количество оперативной памяти. Несмотря на это FAT системы использовались в DOS и Windows (до NT).
- **i-узлы** (index-node) Для каждого файла создается специальная структура, которую называют index-node. В ней хранятся атрибуты файла и указатели на блоки в которых он хранится. Главное преимущество в сравнении с FAT в том, что в памяти хранятся только index-node открытых файлов. Соответственно, объем необходимой памяти пропорционален не общему объему диска, а количеству реально открытых файлов. Так как на один узел отводится ограниченный объем пространства, то его может не хватить на описание всех частей файла. Эту проблему решают путем создания косвенных узлов (то есть, создания иерархии таблиц, описывающих части файла).

Хранение свободных мест:

- битовая карта – за каждый кластер отвечает бит 1 – свободен, 0 – занят (CP/M, HPFS)
- прямые указатели в FAT таблице на сектор (свободен, занят, испорчен)
- список свободных кластеров (ФС в UNIX)

Для любознательных: Дополнительное описание и картинки. Только не сломайте глаза =)

Типы организации файлов на диске и доступа к ним

Файлы по способу расположения на внешнем носителе
(помните, файл это куча записей):

- **последовательный** – файл является последовательностью записей, обработка предполагает последовательное чтение записей от начала файла (модель ленты), причем конкретная запись определяется ее положением в файле.
- **прямой** – содержимое файла может быть разбросано по разным блокам диска, которые можно считывать в произвольном порядке. Для доступа к середине файла просмотр всего файла с самого начала не обязателен, надо только знать физ. адрес записи.
- **индексно-последовательный** – файл представляется в виде последовательности записей переменной длины, к каждой записи присвоен ключ/индекс. Записи отсортированы по значениям ключей/индексов, доступ осуществляется по индексу.

- **индексированный** – для всех записей файла вычисляется пара значений “ID–место”, т.е. идентификатор записи и указатель на то, где она расположена в файле. Совокупность всех таких пар и называется индексом файла. Доступ происходит так: берут ID нужной записи, ищут в индексе указатель на то, где она расположена, переходят по указателю в сам файл и считывают запись.
- **иерархичная** структура – как в библиотеке

Методы доступа к файлам:

- базовые (последовательная обработка, прямое чтение)
- с очередями

Для любознательных: на горячо любимом интуите есть картинки по теме.

Примеры файловых систем

Свое название **FAT** получила от одноименной таблицы размещения файлов, где хранится информация о кластерах логического диска (свободен/занят/помечен как сбойный). Если кластер занят под файл, то в соответствующей записи в таблице размещения файлов указывается адрес кластера, содержащего следующую часть файла. Из-за этого FAT называют файловой системой со связанными списками.

Размер раздела	Размер кластера	Тип FAT
< 16 Мб	4 Кб	FAT12
16 Мб – 127 Мб	2 Кб	FAT16
128 Мб – 255 Мб	4 Кб	FAT16
256 Мб – 511 Мб	8 Кб	FAT16
512 Мб – 1023 Мб	16 Кб	FAT16
1 Гб – 2 Гб	32 Кб	FAT16

FAT может поддерживать разделы размером до 2 Гб.

Если кластер принадлежит файлу, то соответствующая ему ячейка содержит номер следующего кластера этого же файла.

- свободный кластер 0x000
- кластер занят файлом и не является последним кластером файла – номер следующего кластера файла
- bad кластер 0xFF7 / 0xFFF7 / 0x0FFFFFF7 – FAT12/16/32
- последний кластер диапазон 0xFF8-0xFFF соответствующей разрядности (как в бэдах выше)

При удалении файла первый знак имени заменяется специальным кодом E5 и цепочка кластеров файла в таблице размещения обнуляется.

Единственным обязательно присутствующим каталогом является корневой каталог. В FAT12/FAT16 корневой каталог имеет фиксированный размер в секторах.

FAT32 – усовершенствованная версия файловой системы VFAT, поддерживающая жесткие диски объемом до 2 терабайт.

В FAT32 корневой каталог, как любой другой, имеет переменный размер и является цепочкой кластеров.

Кроме того, для учета свободных кластеров, в зарезервированной области на разделе FAT32 имеется сектор, содержащий число свободных кластеров и номер самого последнего использованного кластера. Это позволяет системе при выделении следующего кластера не перечитывать заново всю таблицу размещения файла.

NTFS (New Technology File System) позволяет использовать имена файлов длиной до 255 символов, при этом она использует тот же алгоритм для генерации короткого имени, что и VFAT. NTFS обладает возможностью самостоятельного восстановления в случае сбоя ОС или оборудования, так что дисковый том остается доступным, а структура каталогов не нарушается.

NTFS имеет встроенные возможности разграничивать доступ к данным для различных пользователей и групп пользователей (списки контроля доступа — Access Control Lists (ACL)), а также назначать квоты (ограничения на максимальный объём дискового пространства, занимаемый теми или иными пользователями). NTFS использует систему журналирования для повышения надёжности файловой системы.

Каждый файл на томе NTFS представлен записью в специальном файле – главной файловой таблице MFT (Master File Table). NTFS резервирует первые 16 записей таблицы размером около 1 Мб для специальной информации.

Файловая система UNIX V7

Имена файлов ограничены 14 символами ASCII и чувствительны к регистру, есть поддержка ссылок. Используется схема i-узлов. Позволяет монтировать разделы в любое место дерева системы.

Файловая система BSD

Основу составляет классическая файловая система UNIX. Особенности (отличие от предыдущей системы):

* Увеличена длина имени файла до 255 символов
* Реорганизованы каталоги
* Было добавлено кэширование имен файлов, для увеличения производительности.
* Применено разбиение диска на группы цилиндров.

Файловые системы LINUX

Изначально использовалась файловая система MINIX с ограничениями: 14 символов для имени файла и размер файла 64 Мбайта. После была создана файловая система EXT с расширением: 255 символов для имени файла и размер файла 2Гбайта. Система была достаточно медленной.

Файловая система EXT2 стала основой для LINUX, она очень похожа BSD систему.

Вместо групп цилиндров используются группы блоков. Особенности работы файловой системы:

* Создание новых каталогов распределяется равномерно по группам блоков, чтобы в каждой группе было одинаковое количество каталогов.

* Новые файлы старается создавать в группе, где и находится каталог.

* При увеличении файла система старается новые блоки записывать ближе к старым.

Благодаря этому файловую систему не нужно дефрагментировать, она не способствует фрагментации файлов (в отличии от NTFS), что проверено многолетним использованием.

Файловая система EXT3

В отличие от EXT2, EXT3 является журналируемой файловой системой, т.е. не попадет в противоречивое состояние после сбоев. Драйвер Ext3 хранит полные точные копии модифицируемых блоков (1КБ, 2КБ или 4КБ) в памяти до завершения операции. Такой подход называется "физическим журналированием", что отражает использование "физических блоков" как основную единицу ведения журнала. Подход, когда хранятся только изменяемые байты, а не целые блоки, называется "логическим журналированием" (используется XFS, JFS). Поскольку ext3 использует "физическое журналирование", журнал в ext3 имеет размер больший, чем в XFS. За счет использования в ext3 полных блоков, как драйвером, так и подсистемой журналирования нет сложностей, которые возникают при "логическом журналировании".

XFS – журналируемая файловая система, разработанная Silicon Graphics.

Ориентирована на очень большие файлы. Особенностью этой файловой системы является устройство журнала – туда пишется часть метаданных самой файловой системы таким образом, что весь процесс восстановления сводится к копированию этих данных из журнала в файловую систему. Размер журнала задается при создании системы, он должен быть не меньше 32 мегабайт; а больше и не надо – такое количество незакрытых транзакций тяжело получить. Есть возможность выноса журнала на другой диск. Сохраняет данные кэша только при переполнении памяти, а не периодически как остальные.

* В журнал записываются только мета-данные.

* Используются B+ trees.

RFS (RaiserFS) – журналируемая файловая система, разработанная Namesys.

Некоторые особенности:

* Более эффективно работает с большим количеством мелких файлов, в плане производительности и эффективности использования дискового пространства.

* Использует специально оптимизированные b* balanced tree (усовершенствованная версия B+ дерева)

* Динамически ассигнует i-узлы вместо их статического набора, образующегося при создании "традиционной" файловой системы.

* Динамические размеры блоков.

JFS (Journaled File System) – журналируемая файловая система разработанная IBM.

Некоторые особенности:

* В журнал (не больше 32 мегабайт) записываются только мета-данные

* Асинхронный режим записи в журнал – производится в моменты уменьшения трафика ввода/вывода

NFS (Network File System) – сетевая файловая система. Создана для объединения файловых систем по сети. Используется два протокола(набора запросов и ответов клиента и сервера):

1. Протокол управления монтирования каталогов
2. Протокол управления доступа к каталогам и файлам

VFS (Virtual File System) – виртуальная файловая система. Необходима для управления таблицей открытых файлов.

Записи для каждого открытого файла называются v-узлами (virtual i-node).

VFS используется не только для NFS, но и для работы инородными файловыми системами (FAT, /proc и т.д.)

Оптимизация доступа к диску

Существует 2 подхода к повышению эффективности работы с внешними носителями:

1. **Оптимизация физического обращения к диску** (перемещения головок,…).

1.1 В диске с *фиксированными головками на каждую дорожку приходится одна головка чтения/записи*. В таком диске отсутствует радиальное перемещение головок.

Запросы выстраиваются в очереди, которые формируются 2 методами:

- **FCFS** (First Come First Served – Первый пришел первый обслужен). Одна очередь, когда все запросы обрабатываются в порядке их поступления.
- **Своя очередь для каждого сектора диска**. Запросы переупорядочиваются и обслуживаются с учетом текущего состояния диска.

1.2 В дисках с *подвижными головками* существует дополнительное к вращательному радиальное перемещение для поиска нужной дорожки. Т.е. время поиска складывается из времени поиска сектора и времени поиска дорожки. Радиальное перемещение трудоемко и требует значительных затрат времени по сравнению с вращательным движением.

Запросы к секторам переупорядочиваются и обслуживаются по мере прохождения секторов над головками.

Для поиска дорожки существует целый ряд методов, которые рассмотрены ниже. Эти методы стремятся минимизировать время поиска дорожки при определенных условиях.

- **FCFS** – Справедливый метод, но очень неэффективен с ростом нагрузки.
- **SSTF** (Shortest Seek Time First) – Первым обслуживается запрос с наименьшим расстоянием от текущего положения головки. Ещё до ввода-вывода определяется текущее перемещение головки. Из очереди заявок выбирается заявка с ближайшим перемещением. Метод не справедлив к запросам, которые далеки от текущего положения головки. Возможно бесконечное откладывание таких запросов.

- **SCAN** – Метод сканирования. Головка перемещается в одном направлении, по пути обслуживая заявки Для обслуживания выбирается запрос, ближайший в данном направлении. Направление меняется, если головка достигла граничной дорожки или нет запросов в данном направлении. Это базовый метод работы с дисковыми устройствами.
- **C-SCAN** (Circular scan – Циклическое сканирование). Головки движутся от наружного цилиндра к внутреннему и по ходу движения обслуживают запросы. После завершения движения головки скачком возвращаются к наружному цилиндру и снова повторяют свое движение. Это самая эффективная стратегия.
- **N-step-SCAN** В процессе движения головки в данном направлении обслуживаются только те запросы, которые появились к моменту начала движения в данном направлении. Запросы, которые появились после начала движения в данном направлении, группируются и обслуживаются на следующем проходе.

2. **Оптимизация логического обращения (RAID)**

Redundant array of independent disks — избыточный массив независимых жёстких дисков — массив из нескольких дисков, управляемых контроллером, взаимосвязанных скоростными каналами и воспринимаемых внешней системой как единое целое. В зависимости от типа используемого массива может обеспечивать различные степени отказоустойчивости и быстродействия. Служит для повышения надёжности хранения данных и/или для повышения скорости чтения/записи информации

- **RAID 0** (*striping — «чередование»*) — дисковый массив из 2 и больше жёстких дисков. Информация разбивается на блоки данных и записывается на оба/несколько дисков одновременно.

(+) За счёт этого существенно повышается производительность (от количества дисков зависит кратность увеличения производительности).

(−) Страдает надёжность всего массива (при выходе из строя любого из входящих в RAID 0 винчестеров вся содержащаяся на них информация становится недоступной)

- **RAID 1** (*mirroring — «зеркалирование»*)

(+) Обеспечивает приемлемую скорость записи и выигрыш по скорости чтения при распараллеливании запросов.

(+) Имеет высокую надёжность — работает до тех пор, пока функционирует хотя бы один диск в массиве.

(−) Нужно иметь 2 раза больше дисков, для 1 объёма полезной инфы.

- **RAID 5** Блоки данных и контрольные суммы циклически записываются на все диски массива. Под контрольными суммами подразумевается результат операции XOR. Особенность в том, что если один диск вышел из строя, имея результат XOR и данные 2 диска, можно восстановить инфу. Пример: *a xor b = c,* тогда *c xor b = a.*

(+) Для хранения результата xor требуется всего 1 диск

(−) На запись информации на том RAID 5 тратятся дополнительные ресурсы и падает производительность, так как требуются дополнительные вычисления и операции записи

(+) зато при чтении (по сравнению с отдельным винчестером) имеется выигрыш, потому что потоки данных с нескольких дисков массива могут обрабатываться параллельно.

Для любознательных: если вам интересно узнать о всех-всех видах рэйд и их комбинациях, смотрите Вику.