1. Модель непротиворечивости - последовательная непротиворечивости. Последовательность формирования состава супервизора.

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.

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).

Супервизор

Ядро супервизора – совокупность программ (управляющие программы), обеспечивающих функционирование ВС и находящихся в системной области оперативной памяти, составляет ядро супервизора.

Состав:

1) Пр-мы управления заданиями. Отслеживает вход и выход задания в системе
2) Пр-мы управления задачами (процессами)
3) Система управления файлами.
4) Управление памятью. Делиться на организацию и управление памятью
5) Управление внешними устройствами

Следующие из этого ф-ции:

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; 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.

В супервизоре принимаются решения по использованию ресурсов, необходимых для выполнения его многочисленных функций. В него входят менеджер реальной памяти, менеджер виртуальной памяти, менеджер ресурсов и менеджер вспомогательной памяти, В нем также содержатся программы, управляющие мультипрограммной работой.

Супервизор — обеспечивает управление мультипрограммированием. Он создает диспетчеризуемую единицу работы, осуществляет переключения (диспетчеризация), обеспечивает последовательное использование ресурсов (например, предоставляет средства, обеспечивающие взаимоисключение).

При загрузке ОС необходимо выполнить связывание, размещение всех частей, входящих у ядро супервизора, т.е. размещение резидентных программ на своих местах, формирование специальных системных структур данных в области данных операционной системы, формирование постоянной области, активизацию процессов для начала работы операционной системы. Этот процесс называется инициализацией операционной системы.

Доп. Инфа

Как обрабатывается прерывание по обращению к супервизору.

Прерывания по обращению к супервизору. Вызываются при выполнении процессором команды обращения к супервизору (вызов функции операционной системы). Обычно такая команда инициируется выполняемым процессом при необходимости получения дополнительных ресурсов либо при взаимодействии с устройствами ввода/вывода.

Структура ядра супервизора.

Совокупность программ, обеспечивающих функционирование ВС и находящихся в системной области оперативной памяти, составляет ядро супервизора.

Структура ядра:

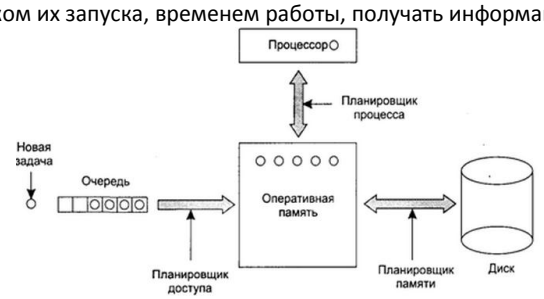
-- резидентные программы
-- системные таблицы

2. Управление задачами и ее функции.

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.

Система управления заданиями предназначена для управления прохождением задач на многопроцессорных вычислительных установках (в том числе кластерных). Она позволяет автоматически распределять вычислительные ресурсы между задачами, управлять порядком их запуска, временем работы, получать информацию о состоянии очередей.



1 уровень. Задание 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, а потом появился программный режим работы, и появились процессы и 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 ре шить, 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 освобожден (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.

Все 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);

3. Проблемы выбора загружаемой части программы в оперативную память и методы ее решения.

При возникновении ошибки отсутствия страницы операционная система должна выбрать «выселяемую» (удаляемую из памяти) страницу, чтобы освободить место для загружаемой страницы. Если предназначенная для удаления страница за время своего нахождения в памяти претерпела изменения, она должна быть переписана на диске, чтобы привести дисковую копию в актуальное состояние. Но если страница не изменялась (например, она содержала текст программы), дисковая копия не утратила своей актуальности, и перезапись не требуется. Тогда считываемая страница просто пишется поверх «выселяемой».

Если бы при каждой ошибке отсутствия страницы можно было выбирать для выселения произвольную страницу, то производительность системы была бы намного выше, если бы выбор падал на редко востребуемую страницу При удалении интенсивно используемой страницы высока вероятность того, что она в скором времени будет загружена опять, что приведет к лишним издержкам. На выработку алгоритмов замещения страниц было потрачено множество усилий как в теоретической, так и в экспериментальной областях. Далее мы рассмотрим некоторые из наиболее важных алгоритмов.

Следует заметить, что проблема «замещения страниц» имеет место и в других областях проектирования компьютеров. К примеру, у большинства компьютеров имеется от одного и более кэшей памяти, содержащих последние использованные 32-байтные или 64-байтные блоки памяти. При заполнении кэша нужно выбрать удаляемые блоки. Это проблема в точности повторяет проблему замещения страниц, за исключением более коротких масштабов времени (все должно быть сделано за несколько наносекунд, а не миллисекунд, как при замещении страниц). Причиной более коротких масштабов времени является то, что ненайденные блоки кэша берутся из оперативной памяти, без затрат времени на поиск и без задержек на раскрутку диска. В качестве второго примера можно взять веб-сервер. На сервере в его кэше памяти может содержаться некоторое количество часто востребуемых веб-страниц. Но при заполнении кэша памяти и обращении к новой странице должно быть принято решение о том, какую веб-страницу нужно выселить. Здесь используются те же принципы, что и при работе со страницами виртуальной памяти, за исключением того, что веб-страницы, находящиеся в кэше, никогда не подвергаются модификации, поэтому на диске всегда имеется их свежая копия. А в системе, использующей виртуальную память, страницы, находящиеся в оперативной памяти, могут быть как измененными, так и неизмененными.

Во всех рассматриваемых далее алгоритмах замещения страниц ставится вполне определенный вопрос: когда возникает необходимость удаления страницы из памяти, должна ли эта страница быть одной из тех, что принадлежат процессу, при работе которого произошла ошибка отсутствия страницы, или это может быть страница, принадлежащая другому процессу? В первом случае мы четко ограничиваем каждый процесс фиксированным количеством используемых страниц, а во втором мы таких ограничений не накладываем. Возможны оба варианта, а к этому вопросу мы еще вернемся.

Оптимальный алгоритм замещения страниц

Наилучший алгоритм замещения страниц несложно описать, но совершенно невозможно реализовать. В нем все происходит следующим образом. На момент воз никновения ошибки отсутствия страницы в памяти находится определенный набор страниц. К некоторым из этих страниц будет осуществляться обращение буквально из следующих команд (эти команды содержатся на странице). К другим страницам обращения может не быть и через 10,100 или, возможно, даже через 1000 команд. Каждая страница может быть помечена количеством команд, которые должны быть выполнены до первого обращения к странице.

Оптимальный алгоритм замещения страниц гласит, что должна быть удалена страница, имеющая пометку с наибольшим значением. Если какая-то страница не будет использоваться на протяжении 8 миллионов команд, а другая какая-нибудь страница не будет использоваться на протяжении 6 миллионов команд, то удаление первой из них приведет к ошибке отсутствия страницы, в результате которой она будет снова выбрана с диска в самом отдаленном будущем. Компьютеры, как и люди, пытаются по возможности максимально отсрочить неприятные события.

Единственной проблемой такого алгоритма является невозможность его реализации. К тому времени, когда произойдет ошибка отсутствия страницы, у операционной системы не будет способа узнать, когда каждая из страниц будет востребована в следующий раз. (Подобная ситуация наблюдалась и ранее, когда мы рассматривали алгоритм планирования, выбирающий сначала самое короткое задание, — как система может определить, какое из заданий самое короткое?) Тем не менее при прогоне программы на симуляторе и отслеживании всех обращений

к страницам появляется возможность реализовать оптимальный алгоритм замещения страниц при втором прогоне, воспользовавшись информацией об обращении к страницам, собранной во время первого прогона.

Таким образом появляется возможность сравнить производительность осуществимых алгоритмов с наилучшим из возможных. Если операционная система достигает производительности, скажем, на 1% хуже, чем у оптимального алгоритма, то усилия, затраченные на поиски более совершенного алгоритма, дадут не более 1% улучшения.

Чтобы избежать любой возможной путаницы, следует уяснить, что подобная регистрация обращений к страницам относится только к одной программе, прошедшей оценку, и только при одном вполне определенном наборе входных данных. Таким образом, полученный в результате этого алгоритм замещения страниц относится только к этой конкретной программе и к конкретным входным данным. Хотя этот метод и применяется для оценки алгоритмов замещения страниц, в реальных системах он бесполезен. Далее мы будем рассматривать те алгоритмы, которые действительно полезны для реальных систем.

- Алгоритм «первой пришла, первой и ушла»
- Алгоритм «второй шанс»
- Алгоритм «часы»
- Алгоритм замещения наименее востребованной страницы

- Алгоритм «Рабочий набор»
- Алгоритм WSClock

Краткая сравнительная характеристика алгоритмов замещения страниц.

Алгоритм	Особенности
Оптимальный	Не может быть реализован, но полезен в качестве оценочного критерия
NRU (Not Recently Used) — алгоритм исключения недавно использовавшейся страницы	Является довольно грубым приближением к алгоритму LRU
FIFO (First-In, First-Out) — алгоритм «первой пришла, первой и ушла»	Может выгрузить важные страницы
Алгоритм «второй шанс»	Является существенным усовершенствованием алгоритма FIFO
Алгоритм «часы»	Вполне реализуемый алгоритм
LRU (Least Recently Used) — алгоритм замещения наименее востребованной страницы	Очень хороший, но труднореализуемый во всех тонкостях алгоритм
NFU (Not Frequently Used) — алгоритм нечастого востребования	Является довольно грубым приближением к алгоритму LRU
Алгоритм старения	Вполне эффективный алгоритм, являющийся неплохим приближением к алгоритму LRU
Алгоритм рабочего набора	Весьма затратный для реализации алгоритм
WSClock	Вполне эффективный алгоритм

Оптимальный алгоритм удаляет страницу с самым отдаленным предстоящим обращением. К сожалению, у нас нет способа определения, какая это будет страница, поэтому на практике этот алгоритм использоваться не может. Но он полезен в качестве оценочного критерия при рассмотрении других алгоритмов. Алгоритм исключения недавно использованной страницы — **NRU** делит страницы на четыре класса в зависимости от состояния битов Ru M. Затем он выбирает произвольную страницу из класса с самым низким номером. Этот алгоритм нетрудно реализовать, но он слишком примитивен. Есть более подходящие алгоритмы.

Алгоритм **FIFO** предполагает отслеживание порядка, в котором страницы были загружены в память, путем сохранения сведений об этих страницах в связанном списке. Это упрощает удаление самой старой страницы, но она-то как раз и может все еще использоваться, поэтому FIFO — неподходящий выбор.

Алгоритм «второй шанс» является модификацией алгоритма FIFO и перед удалением страницы проверяет, не используется ли она в данный момент. Если страница все еще используется, то она остается в памяти. Эта модификация существенно повышает производительность. **Алгоритм «часы»** является простой разновидностью алгоритма «второй шанс». Он имеет такой же показатель производительности, но требует несколько меньшего времени на свое выполнение.

Алгоритм LRU превосходен во всех отношениях, но не может быть реализован без специального оборудования. Если такое оборудование недоступно, то он не может быть использован. **Алгоритм NFU** является грубой попыткой приблизиться к алгоритму LRU. Его нельзя признать удачным. А вот **алгоритм старения** — куда более удачное приближение к алгоритму LRU, которое к тому же может быть эффективно реализовано и считается хорошим выбором.

В двух последних алгоритмах используется рабочий набор. **Алгоритм рабочего набора** предоставляет приемлемую производительность, но его реализация обходится слишком дорого. **Алгоритм WS Clock** является вариантом, который не только предоставляет неплохую производительность, но также может быть эффективно реализован.

В конечном итоге наиболее приемлемыми алгоритмами являются алгоритм старения и алгоритм **WSClock**. Они основаны соответственно на LRU и на рабочем наборе. Оба предоставляют неплохую производительность страничной организации памяти и могут быть эффективно реализованы. Существует также и ряд других алгоритмов, но эти два, наверное, имеют наибольшее практическое значение.

4. Методы обеспечения надежности файловых систем.

Надежность файловых систем может быть увеличена при помощи создания инкрементных резервных копий, а также с помощью программы, способной исправлять поврежденные файловые системы. Производительность файловых систем также является важным вопросом. Она может быть увеличена различными способами, включая кэширование, опережающее чтение и размещение блоков файла рядом друг с другом. Файловые системы с журнальной структурой тоже увеличивают производительность, выполняя операции записи большими блоками данных.

На современные файловые системы оказывают влияние и технологические изменения. В частности, постоянно растущая скорость центральных процессоров, увеличение емкости и удешевление дисковых накопителей (при не столь впечатляющем увеличении скорости их работы), рост в геометрической прогрессии объема оперативной памяти. Единственным параметром, не демонстрирующим столь стремительного роста, остается время позиционирования блока головок на нужный цилиндр диска. Сочетание всех этих факторов свидетельствует о том, что у многих файловых систем возникает узкое место в росте производительности. Во время исследований, проведенных в университете Беркли, была предпринята попытка смягчить остроту этой проблемы за счет создания совершенно нового типа файловой системы — LFS (Log-structured File System — файловая система с журнальной структурой). Этот раздел будет посвящен краткому описанию работы LFS. В основу LFS заложена идея о том, что по мере повышения скорости работы центральных процессоров и увеличения объема оперативной памяти существенно повышается и уровень кэширования дисков. Следовательно, появляется возможность удовлетворения весьма существенной части всех дисковых запросов на чтение прямо из кэша файловой системы без обращения к диску. Из этого наблюдения следует, что в будущем основную массу обращений к диску будут составлять операции записи, поэтому механизм опережающего чтения, применявшийся в некоторых файловых системах для извлечения блоков еще до того, как в них возникнет потребность, уже не дает значительного прироста производительности. Усложняет ситуацию то, что в большинстве файловых систем запись производится очень малыми блоками данных. Запись малыми порциями слишком неэффективна, поскольку записи на диск, занимающей 50 мкс, зачастую предшествует позиционирование на нужный цилиндр, на которое затрачивается 10 мс, и ожидание подхода под

головку нужного сектора, на которое уходит 4 мс. При таких параметрах эффективность работы с диском падает до долей процента. Чтобы понять, откуда берутся все эти мелкие записи, рассмотрим создание нового файла в системе UNIX. Для записи этого файла должны быть записаны i-узел для каталога, блок каталога, i-узел для файла и сам файл. Эти записи могут быть отложены, но если произойдет сбой до того, как будут произведены все записи, файловая система столкнется с серьезными проблемами согласованности данных. Поэтому, как правило, записи i-узлов производятся немедленно. Исходя из этих соображений разработчики LFS решили переделать файловую систему UNIX таким образом, чтобы добиться работы диска с полной пропускной способностью, даже если объем работы состоит из существенного количества небольших произвольных записей. В основу была положена идея структурировать весь диск в виде журнала. Периодически, когда в этом возникает особая надобность, все ожидающие осуществления записи, находящиеся в буфере памяти, собираются в один сегмент и записываются на диск в виде одного непрерывного сегмента в конец журнала. Таким образом, отдельный сегмент может вперемешку содержать i-узлы, блоки каталога и блоки данных. В начале каждого сегмента находится его сводная информация, в которой сообщается, что может быть найдено в этом сегменте. Если средний размер сегмента сможет быть доведен примерно до 1 Мбайт, то будет использоваться практически вся пропускная способность диска. В этой конструкции по-прежнему используются i-узлы той же структуры, что и в UNIX, но теперь они не размещаются в фиксированной позиции на диске, а разбросаны по всему журналу. Тем не менее когда определено местоположение i-узла, определение местоположения блоков осуществляется обычным образом. Разумеется, теперь нахождение i-узла значительно усложняется, поскольку его адрес не может быть просто вычислен из его i-номера, как в UNIX. Для осуществления поиска i-узлов ведется массив i-узлов, проиндексированный по i-номерам. Элемент i в таком массиве указывает на i-узел на диске. Массив хранится на диске, но он также подвергается кэшированию, поэтому наиболее интенсивно использующиеся части большую часть времени будут находиться в памяти.

Подытоживая все ранее сказанное, все записи сначала буферизуются в памяти, и периодически все, что попало в буфер, записывается на диск в один сегмент, в конец журнала. Открытие файла теперь состоит из использования массива для определения местоположения i-узла для этого файла. После определения местоположения i-узла из него могут быть извлечены адреса блоков. А все блоки будут в сегментах, расположенных в различных местах журнала. Если бы диски были безразмерными, то представленное описание на этом бы и закончилось. Но существующие диски не безграничны, поэтому со временем журнал займет весь диск и новые сегменты не смогут быть записаны в журнал. К счастью, многие существующие сегменты могут иметь уже ненужные блоки, к примеру, если файл перезаписан, его i-узел теперь будет указывать на новые блоки, но его старые блоки все еще будут занимать пространство в ранее записанных сегментах. Чтобы справиться с этой проблемой, у LFS есть очищающий поток, который занимается тем, что осуществляет круговое сканирование журнала с целью уменьшения его размера. Сначала он считывает краткое содержание первого сегмента журнала, чтобы увидеть, какие i-узлы и файлы в нем находятся. Затем он проверяет текущий массив i-узлов, чтобы определить, актуальны ли еще i-узлы и используются ли еще файловые блоки. Если они уже не нужны, то информация выбрасывается. Те i-узлы и блоки, которые еще используются, перемещаются в память для записи в следующий сегмент. Затем исходный сегмент помечается как свободный, и журнал может его использовать для новых данных. Таким же образом очищающий поток перемещается по журналу, удаляя позади устаревшие сегменты и помещая все актуальные данные в память для их последующей повторной записи в следующий сегмент. В результате диск становится большим кольцевым буфером с пишущим потоком, добавляющим впереди новые сегменты, и с очищающим потоком, удаляющим позади устаревшие сегменты. Управление использованием блоков на диске в этой системе имеет необычный характер, поскольку, когда файловый блок опять записывается на диск в новый сегмент, должен быть найден i-узел файла (который находится где-то в журнале), после чего он должен быть обновлен и помещен в память для записи в следующий сегмент. Затем должен быть обновлен массив i-узлов, чтобы в нем присутствовал указатель на новую копию. Тем не менее такое администрирование вполне осуществимо, и результаты производительности показывают, что все эти сложности вполне оправданны. Результаты замеров, приведенные в процитированной выше статье, свидетельствуют о том, что LFS превосходит UNIX на малых записях на целый порядок, обладая при этом производительностью чтения и записи больших объемов данных, которая, по крайней мере, не хуже, чем у UNIX.

Журналируемые файловые системы.

При всей привлекательности идеи файловых систем с журнальной структурой они не нашли широкого применения, отчасти из-за их крайней несовместимости с существующими файловыми системами. Тем не менее одна из идей, позаимствованная у них, — устойчивость к отказам может быть внедрена и в более привычные файловые системы. Основная идея заключается в журналировании всех намерений файловой системы перед их осуществлением, поэтому если система терпит аварию еще до того, как у нее появляется возможность осуществить запланированные действия, то после перезагрузки она может посмотреть в журнал, определить, что она собиралась сделать на момент аварии, и завершить свою работу. Такие файловые системы, которые называются журналируемыми файловыми системами, нашли свое применение. Журналируемыми являются файловая система NTFS, разработанная Microsoft, а также файловые системы Linux ext3 и ReiserFS.

Чтобы вникнуть в суть проблемы, рассмотрим заурядную, часто встречающуюся операцию удаления файла. Для этой операции в UNIX нужно выполнить три действия:

1. Удалить файл из его каталога.
2. Освободить i-узел, поместив его в пул свободных i-узлов.
3. Вернуть все дисковые блоки файла в пул свободных дисковых блоков.

В Windows требуются аналогичные действия. В отсутствие отказов системы порядок выполнения этих трех действия не играет роли, чего нельзя сказать о случае возникновения отказа. Представьте, что первое действие завершено, а затем в системе возник отказ. Не станет файла, из которого возможен доступ к i-узлу и к блокам, занятым данными файла, но они также не будут доступны и для переназначения; они превратятся в ничто, сокращая объем доступных ресурсов. А если отказ произойдет после второго действия, то будут потеряны только блоки. Если последовательность действий изменится и сначала будет освобожден i-узел, то после перезагрузки системы этот i-узел можно будет переназначить, но на него будет по-прежнему указывать старый элемент каталога, приводя к неверному файлу. Если первыми будут освобождены блоки, то отказ до освобождения i-узла будет означать, что действующий элемент каталога указывает на i-узел, в котором

перечислены блоки, которые теперь находятся в пуле освободившихся блоков, и которые в ближайшее время, скорее всего, будут использованы повторно, что приведет к произвольному совместному использованию одних и тех же блоков двумя и более файлами. Ни одно из этих последствий нас не устраивает. В журналируемой файловой системе сначала делается запись в журнале, в которой перечисляются три намеченных к выполнению действия. Затем журнальная запись сбрасывается на диск (и дополнительно, возможно, эта же запись считывается с диска, чтобы убедиться в ее целостности). И только после сброса журнальной записи на диск выполняются различные операции. После успешного завершения операций журнальная запись удаляется. Теперь при отказе системы, после ее восстановления, файловая система может проверить журнал, чтобы определить наличие какой-либо незавершенной операции. Если таковая найдется, то все операции могут быть запущены заново (причем по нескольку раз в случае повторных отказов) до тех пор, пока файл не будет удален по всем правилам. При журналировании все операции должны быть идемпонентными, что означает возможность их повторения необходимое число раз без нанесения какого-либо вреда. Такие операции, как «Обновить битовый массив, пометив i-узел k или блок n свободными», могут повторяться до тех пор, пока все не завершится должным и вполне безопасным образом. По аналогии с этим поиск в каталоге и удаление любой записи с именем foobar также является идемпонентным действием. С другой стороны, добавление только что освободившихся блоков из i-узла К к концу перечня свободных блоков не является идемпонентным действием, поскольку они уже могут присутствовать в этом перечне. Более ресурсоемкая операция «Просмотреть перечень свободных блоков и добавить к нему блок п, если он в нем не присутствовал» является идемпонентной. Журналируемые файловые системы должны выстраивать свои структуры данных и журналируемые операции таким образом, чтобы все они были идемпонентными. При таких условиях восстановление после отказа может проводиться быстро и безопасно. Для придания дополнительной надежности в файловой системе может быть реализована концепция атомарной транзакции, присущая базам данных. При ее использовании группа действий может быть заключена между операциями начала транзакции — begin transaction и завершения транзакции — end transaction. Распознающая эти операции файловая система должна либо полностью выполнить все заключенные в эту пару операции, либо не выполнить ни одной из них, не допуская никаких других комбинаций. NTFS обладает исчерпывающей системой журналирования и ее структура довольно редко повреждается в результате системных отказов. Ее разработка продолжалась и после своего первого выпуска в Windows NT в 1993 году Первой журналируемой файловой системой Linux была ReiserFS, но развитию ее популярности помешала несовместимость с применяемой в ту пору стандартной файловой системой ext2. В отличие от нее ext3, представляющая собой менее амбициозный проект, чем ReiserFS, также журналирует операции, но при этом обеспечивает совместимость с предыдущей системой ext2.

Согласование файловой системы

Программа согласования файловой системы ведет борьбу с рассогласованием файловой системы. Система пытается проверить соответствуют записи занятых файлов реальным записям. Рассогласования может появится например при непредвиденном выключении системы, когда информация о записях файлов (например таблицы ФАТА), не была скопирована на жесткий диск, хотя физически информация поменялась. В программах согласования составляют 2 таблицы: занятых и свободных кластеров. Далее проверяются записи, если есть указатель на кластер тогда в таблицу занятых кластеров вносится 1. Если он свободен – тогда 1 в таблицу свободных кластеров. Тогда сочетание 1(таблица занятых) и 0 (таблица свободных) – это нормальное сочетание, а вот 1 (таблица занятых) и 1 (таблица свободных) – несогласования. Если на 1 и тот же кластер ссылается 2 файла, то фиксируется ошибка и кластер дублируется еще раз и ссылка расшивается. Часто для повышения эффективной работы с файлом рекомендуется отображение файла в ОП. В случае збоя файловой системы для согласования составляется два вектора (битовых масива). Для занятых и для свободных секторов. Если бит занят и не свободен (после сбоя) то система ставит что он свободен. Если для сектор и бит занятости и бит свободности равен 1 , то в векторе для свободных ставится 0. Если в векторе6 занятых стоит 2, значит из двух файлов идёт обращение к этому сектору, тогда система копирует этот сектор и раскидывает эти два указателя (расшивка).

5. Этапы входа в прерывающую программу.

Основная функция – формирование начального адреса прерывающей программы. Любому запросу соответствует своя прерывающая программа. Существует три различных способа, используемых при формировании адреса:

- Код приоритета
- запроса
- Перераспределены приоритеты

1) Размещение прерывающих программ по фиксированным адресам. В некоторой постоянно распределенной области основной памяти по фиксированным адресам размещаются прерывающие программы, это размещение не меняется. Вход реализуется аппаратно, то есть адрес формируется аппаратно – это самый быстрый способ. Но у данного способа существуют серьезные ограничения:

- привязка к адресам
- количество причин прерывания должно быть достаточно малым

Этот способ применяется при малых системах прерывания и для тех причин, которые требуют немедленной реакции.

2) вход на основании слов состояния программы (PSW). Типичная структура слова состояния программы:

Маска прерывания **Ключ** защиты памяти **Код** состояния **CPU Адрес** команды (пр-мы)

Схема входа в прерывающую программу.

В некоторую постоянно распределенной области основной памяти формируется два массива: массив старых PSW и массив новых PSW. Любая пара слов состояния соответствует определенному запросу на прерывание. После выполнения активного запроса по соответствующему адресу в массив старых PSW загружается PSW текущей (прерываемой) программы. Характеристика программы в виде PSW записывается по определенному адресу. Из второго массива загружается новое (соответствующее прерывающей программе) PSW. Адрес записан в памяти – из PSW. Массив новых PSW всегда формируется при загрузке ОС. Массив старых PSW формируется в процессе работы. В отличие от предыдущего способа, использование PSW позволяет обслуживать и вложенные прерывания, если их приоритет выше текущей программы. Все это позволяет прерывать прерывающую программу.

Недостаток: Вход в прерывающую программу требует загрузки достаточно больших слов (большого процессорного времени), следовательно, данный способ не очень быстрый. Этот вариант используется в универсальных компьютерах (для решения расчетных задач, т.е. не критичных ко времени)

3) Векторное прерывание – самый распространенный способ. Данный способ является программно-аппартным, т.е. для любого запроса (для любого выделенного запроса) аппаратно формируется адрес вектора прерывания. Чаще всего эти адреса фиксированы. Адреса векторных прерываний хранятся в системной области памяти. На основе адреса вектора из таблицы векторов прерывания извлекается начальный адрес прерывающей программы. Это приводит к тому, что код запроса может быть малобитным, но таблица векторов прерывания должна хранится в начальной области памяти. В качестве вектора прерывания используются:

- адрес начала прерывающей программы (применяется в PC)
- команда безусловного перехода к программе

Способ 1 оказывается универсальным и допускает любую глубину прерывающих программ. Если таблица векторов является загружаемой, то это дает возможность пользователю изменять прерывающие программы даже в процессе решения задачи, которые обрабатывают один и тот же запрос.

Запоминание состояния прерванной программы.

Вся запоминаемая информация делится на основную и дополнительную. Основная информация должна запоминаться всегда – адрес текущей программы, в которой произошло прерывание, состояние процессора, уровень приоритетности программы. Основная информация компонуется в слово-состояние. Основная информация запоминается аппаратно. Дополнительную информацию запоминает сам пользователь. При запоминании основной информации используются два способа:

1) Использование PSW (запоминание старого PSW – основная информация). Маска прерывания Ключ защиты памяти Код состояния CPU Адрес команды (пр-мы).
2) Запоминание основной информации в системном стеке, который поддерживается ОС.

Использование стековых структур при входе в прерывающую программу позволяет не ограничивать глубину вложения прерываний. Ограничения только в связи с размерами стека. Дополнительная информация с точки зрения объема различна. В каждом конкретном случае определяется самостоятельно – ресурсы процессора, которые используются при работе самой прерывающей программы.

Восстановление состояния прерванной программы: Инвертирование тех действий, которые выполнены при запоминании.

Возврат: Передача управления в ту точку, где произошло прерывание. Реализуется обычно аппаратно. Зависит от организации входа в прерывающую программу.