1 Общая схема функционирования 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);
- 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.

2 Структура PCB и роль отдельных частей для управления вычислительным процессом.

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);

Данные структур PCB для всех процессов, в любой момент времени, должны присутствовать в памяти, хотя остальные структуры данных, включая образ процесса, могут быть перемещены во вторичную память, — область свопинга. Это позволяет ядру иметь под рукой минимальную и достаточную информацию, необходимую для определения местонахождения остальных данных, относящихся к процессу, даже если они отсутствуют в памяти. Структура PCB является отдельной записью в системной таблице процессов. Запись этой таблицы для выполняющегося в настоящий момент времени процесса адресуется содержимым регистра TR.

Когда ОС переключает процессор с процесса на процесс, она использует области сохранения регистров в PCB для запоминания информации, необходимой для рестарта (повторного запуска) каждого процесса с точки прерывания, когда он в следующий раз получит в свое распоряжение процессор. Количество процессов в системе ограничено и определяется самой системой или пользователем во время генерации ОС. Неудачное определение количества одновременно исполняемых программ может привести к снижению полезной эффективности работы системы, т.к. переключение процессов требует выполнения дополнительных операций по сохранению и восстановлению состояния процессов. Блоки управления системных процессов создаются при загрузке системы. Это необходимо, чтобы система выполняла свои функции достаточно быстро, а время реакции ОС было минимальным. Однако, количество блоков управления системными процессами меньше, чем количество самих системных процессов. Это связано с тем, что структура ОС имеет либо оверлейную, либо динамически ? последовательную или параллельные структуры иерархического типа, и нет необходимости создавать отдельные PCB для процессов, которые никогда не будут активизироваться одновременно. При такой организации легко учитывать приоритеты системных процессов, выстроив их по приоритетам заранее при инициализации системы. Блоки управления проблемными (пользовательскими) процессами создаются в процессе активизации процессов динамически, а их приоритеты могут изменяться во время жизненного пути процессов. Все PCB находятся в выделенной системной области памяти.

В каждом PCB есть поле состояния процесса. Все блоки управления системными процессами располагаются в порядке убывания приоритетов и находятся в системной области памяти. Если приоритеты системных блоков можно определить заранее, то для проблемных процессов необходима таблица приоритетов проблемных программ. Каждый блок PCB имеет стандартную структуру, фиксированный размер, точку входа и содержит, как правило, указанную выше информацию и дополнительную информацию для синхронизации процессов. Для синхронизации в PCB используются поля:

1. поля для организации связи (два поля):
1.1. Имя вызванного процесса.
1.2. Имя вызвавшего процесса.
2. поля для организации ожидания (два поля).
2.1. Поле ожидания. Имя процесса, освобождения которого ожидает данный.
2.2. Счетчик ожидания. Число процессов, ожидающих данный.

В поле организации связи у порожденного процесса указывается идентификатор родительского процесса и код возврата, передаваемый родительскому процессу при завершении своих действий потомком. Родительский процесс получает от порожденного процесса свой идентификатор.

В полях организации ожидания, указывается адрес PCB вызываемого процесса, если вызываемый процесс занят. В соответствующем поле занятого процесса записывается число процессов, его ожидающих.

Если процесс A пытается вызвать процесс B, а у процесса B в PCB занята цепочка связей, то есть он является занятым по отношению к другим процессам, тогда адрес процесса B записывается в поле ожидания PCB процесса A, а к содержимому поля счетчика ожидания PCB процесса B добавляется 1. Как только процесс B завершает выполнение своих функций, он передает управление вызывающему процессу следующим образом: B проверяет состояние своего счетчика ожидания, и, если счетчик больше 0, то среди PCB других процессов ищется первый (по приоритету или другим признакам) процесс, в поле ожидания PCB которого стоит имя ожидаемого процесса, в данном случае B, тогда управление передается этому процессу.

3 Стратегии выбора страниц для замещения.

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

Только что мы рассмотрели несколько различных алгоритмов замещения страниц.

В этом разделе мы дадим им краткую сравнительную характеристику Список рассмотренных алгоритмов представлен в табл.

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 Системные объекты, обеспечивающие работу 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).

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

Рис. 4.9. Размещение с помощью связанного списка, использующего таблицу размещения файлов в оперативной памяти

(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.

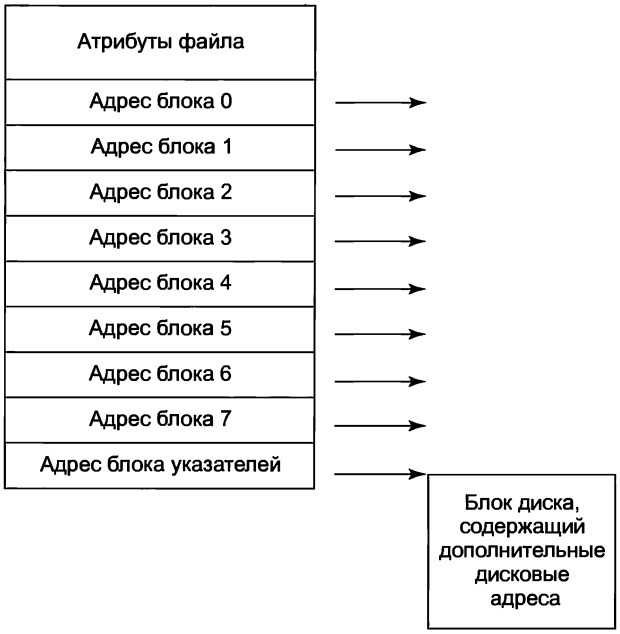

Рис. 4.10. Пример i-узла

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).

Особенности входа в состояние ожидания и выхода из него.

Процес не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояние ожидание; Из состояния ожидание процесс попадает в состояние готовность после того, как ожидаемое событие произошло, и он снова может быть выбран для исполнения.

Переход 1 происходит, когда процесс обнаруживает, что продолжение работы невозможно. В некоторых системах процесс должен выполнить системный запрос, например block или pause, чтобы оказаться в заблокированном состоянии. В других системах, как в UNIX, процесс автоматически блокируется, если при счи тывании из капала или специального файла (предположим, терминала) входные данные не были обнаружены. Переход 4 происходит с появлением внешнего события, ожидавшегося про цессом (например, прибытие входных данных). Если в этот момент не запущен какой-либо другой процесс, то срабатывает переход 3, и процесс запускается. В противном случае процессу придется некоторое время находиться в состоянии готовности, пока не освободится процессор.

Заблокированные процессы находятся в состоянии ожидания. Кому не повезёт – освободят ОП и будут своппированны на диск. **3swap** – процесс с диска можно опять вернуть в ОП в очередь заблокированных.