

Лекция 6. Ядро, shell, команды и файловая система в ОС Unix

1 Ядро, shell, команды в ОС Unix

2 Файловая система в ОС Unix

3 Базовые механизмы сетевых взаимодействий

1 Ядро ОС Unix

Как и в любой другой многопользовательской операционной системе, обеспечивающей защиту пользователей друг от друга и защиту системных данных от любого непривилегированного пользователя, в ОС UNIX имеется защищенное ядро, которое управляет ресурсами компьютера и предоставляет пользователям базовый набор услуг.

Одно из основных достижений ОС UNIX состоит в том, что система обладает свойством высокой мобильности. Смысл этого качества состоит в том, что вся операционная система, включая ее ядро, сравнительно просто переносится на различные аппаратные платформы. Все части системы, не считая ядра, являются полностью машинно-независимыми. Эти компоненты аккуратно написаны на языке Си, и для их переноса на новую платформу (по крайней мере, в классе 32-разрядных компьютеров) требуется только перекомпиляция исходных текстов в коды целевого компьютера.

Однако сравнительно небольшая часть ядра является машинно-зависимой и написана на смеси языка С и языка ассемблера целевого процессора. При переносе системы на новую платформу требуется переписывание этой части ядра с использованием языка ассемблера и учетом специфических черт целевой аппаратуры. Машинно-зависимые части ядра хорошо изолированы от основной машинно-независимой части, и при хорошем понимании назначения каждого машинно-зависимого компонента переписывание машинно-зависимой части является в основном технической задачей (хотя и требует высокой программистской квалификации). Машинно-зависимая часть традиционного ядра ОС UNIX включает следующие компоненты:

- раскрутка и инициализация системы на низком уровне (пока это зависит от особенностей аппаратуры);
- первичная обработка внутренних и внешних прерываний;
- управление памятью (в той части, которая относится к особенностям аппаратной поддержки виртуальной памяти);
- переключение контекста процессов между режимами пользователя и ядра;
- связанные с особенностями целевой платформы части драйверов устройств.

К основным функциям ядра ОС UNIX принято относить следующие:

Инициализация системы –

функция запуска и раскрутки. Ядро системы обеспечивает средство раскрутки (bootstrap), которое обеспечивает загрузку полного ядра в память компьютера и запускает ядро.

Управление процессами и нитями –

функция создания, завершения и отслеживания существующих процессов и нитей ("процессов", выполняемых на общей виртуальной памяти). Поскольку ОС UNIX является мультипроцессной операционной системой, ядро обеспечивает разделение между запущенными процессами времени процессора (или процессоров в мультипроцессорных системах) и других ресурсов компьютера для создания внешнего ощущения того, что процессы реально выполняются в параллель.

Управление памятью –

функция отображения практически неограниченной виртуальной памяти процессов в физическую оперативную память компьютера, которая имеет ограниченные размеры. Соответствующий компонент ядра обеспечивает разделяемое использование одних и тех же областей оперативной памяти несколькими процессами с использованием внешней памяти.

Управление файлами –

функция, реализующая абстракцию файловой системы, - иерархии каталогов и файлов. Файловые системы ОС UNIX поддерживают несколько типов файлов. Некоторые файлы могут содержать данные в формате ASCII, другие будут соответствовать внешним устройствам. В файловой системе хранятся объектные файлы, выполняемые файлы и т.д. Файлы обычно хранятся на устройствах внешней памяти; доступ к ним обеспечивается средствами ядра. В мире UNIX существует несколько типов организации файловых систем. Современные варианты ОС UNIX одновременно поддерживают большинство типов файловых систем.

Коммуникационные средства –

функция, обеспечивающая возможности обмена данными между процессами, выполняющимися внутри одного компьютера (IPC - Inter-Process Communications), между процессами, выполняющимися в разных узлах локальной или глобальной сети передачи данных, а также между процессами и драйверами внешних устройств.

Программный интерфейс –

функция, обеспечивающая доступ к возможностям ядра со стороны пользовательских процессов на основе механизма системных вызовов, оформленных в виде библиотеки функций.

Принципы взаимодействия с ядром.

Выполнение пользовательских процессов в системе UNIX осуществляется на двух уровнях: уровне пользователя и уровне ядра. Когда процесс производит обращение к операционной системе, режим выполнения процесса переключается с режима задачи (пользовательского) на режим ядра: операционная система пытается обслужить запрос пользователя, возвращая код ошибки в случае неудачного завершения операции. Даже если пользователь не нуждается в каких-либо определенных услугах операционной системы и не обращается к ней с запросами, система еще выполняет учетные операции, связанные с пользовательским процессом, обрабатывает прерывания, планирует процессы, управляет распределением памяти и т.д.

Основные различия между этими двумя режимами:

- В режиме задачи процессы имеют доступ только к своим собственным инструкциям и данным, но не к инструкциям и данным ядра (либо других процессов). Однако в режиме ядра процессам уже доступны адресные пространства ядра и пользователей. Например, виртуальное адресное пространство процесса может быть поделено на адреса, доступные только в режиме ядра, и на адреса, доступные в любом режиме.
- Некоторые машинные команды являются привилегированными и вызывают возникновение ошибок при попытке их использования в режиме задачи. Например, в машинном языке может быть команда, управляющая регистром состояния процессора; процессам, выполняющимся в режиме задачи, она недоступна.

В любой операционной системе поддерживается некоторый механизм, который позволяет пользовательским программам обращаться за услугами ядра ОС. В ОС UNIX такие средства называются системными вызовами. Их смысл состоит в том, что для обращения к функциям ядра ОС используются "специальные команды" процессора, при выполнении которых возникает особого рода внутреннее прерывание процессора, переводящее его в режим ядра. При обработке таких прерываний (дешифрации) ядро ОС распознает, что на самом деле прерывание является запросом к ядру со стороны пользовательской программы на выполнение определенных действий, выбирает параметры обращения и обрабатывает его, после чего выполняет "возврат из прерывания", возобновляя нормальное выполнение пользовательской программы.

Конкретные механизмы возбуждения внутренних прерываний по инициативе пользовательской программы различаются в разных аппаратных архитектурах. ОС UNIX потребовался дополнительный уровень, скрывающий особенности конкретного механизма возбуждения внутренних прерываний. Этот механизм обеспечивается так называемой библиотекой системных вызовов. Для пользователя библиотека системных вызовов представляет собой обычную библиотеку заранее реализованных функций системы программирования языка Си.

Суть принятого на сегодня механизма обработки прерываний состоит в том, что каждому возможному прерыванию процессора (будь то внутреннее или внешнее прерывание) соответствует некоторый фиксированный адрес физической оперативной памяти. В тот момент, когда процессору разрешается прерваться по причине наличия внутренней или внешней заявки на прерывание, происходит аппаратная передача управления на ячейку физической оперативной памяти с соответствующим адресом - обычно адрес этой ячейки называется "вектором прерывания" (как правило, заявки на внутреннее прерывание, т.е. заявки, поступающие непосредственно от процессора, удовлетворяются немедленно). Дело операционной системы - разместить в соответствующих ячейках оперативной памяти программный код, обеспечивающий начальную обработку прерывания и инициирующий полную обработку.

Shell

Shell - это программа, которая позволяет вам связываться с операционной системой. Shell считывает команды, которые вы вводите, и интерпретирует их как запросы на выполнение других программ, на доступ к файлу или обеспечение вывода. Shell также является мощным языком программирования, не похожим на язык программирования Си, который обеспечивает условное выполнение и управление потоками данных.

Команды

Программа - это набор инструкций для компьютера. Программы, которые могут быть выполнены компьютером без предварительной трансляции, называются исполняемыми программами или командами. Как обычному пользователю системы UNIX вам доступно множество стандартных программ и инструментальных средств. Если вы используете систему UNIX для написания программ и развития программного обеспечения, то вы также можете задействовать системные вызовы, подпрограммы и другие инструментальные средства. И, конечно, любая написанная вами программа будет в вашем распоряжении.

Что делают команды

Внешний круг системы UNIX образуют программы и инструментальные средства системы, разделенные на категории функционально. Эти функции включают:

программное окружение -

несколько программ системы UNIX, устанавливающих дружественное программное окружение, обеспечивающее интерфейсы между системой и языками программирования и использование обслуживающих программ;

обработка текстов -

система обеспечивает программы, такие как строковый и экранный редакторы, для создания и изменения текстов, орфографическую программу проверки для обнаружения ошибок орфографии, и необязательный форматор текста для создания высококачественных копий, которые подходят для публикаций;

организация информации -

система предоставляет много программ, которые позволяют вам создавать, организовывать и удалять файлы и каталоги;

обслуживающие программы -

инструментальные средства, создающие графику и выполняющие вычисления;

электронная связь -

несколько программ (например, mail) предоставляют вам возможность передавать информацию другим пользователям и в другие системы UNIX.

Как выполнять команды

Чтобы ваш запрос был понятен системе UNIX вы должны ввести каждую команду в корректном формате или синтаксисе командной строки. Этот синтаксис определяет порядок, в котором вы вводите компоненты командной строки. И вы должны расположить все составные

части командной строки в требуемом синтаксисом порядке, иначе shell не сможет интерпретировать ваш запрос.

После завершения выполнения программы, shell сигнализирует, что готов выполнить следующую команду, напечатав подсказку.

2 Файловая система в ОС Unix

Файловая система является краеугольным камнем операционной системы UNIX. Она обеспечивает логический метод организации, восстановления и управления информацией. Файловая система имеет иерархическую структуру.

Файл, который является основной единицей системы UNIX, может быть: обыкновенным файлом, справочником, специальным файлом.

Обыкновенные файлы

Обыкновенные файлы являются набором символов. Обыкновенные файлы используются для хранения любой информации. Они могут содержать тексты для писем или отчетов, коды программ, которые вы написали, либо команды для запуска ваших программ. Однажды создав обыкновенный файл вы можете добавить нужный материал в него, удалить материал из него, либо удалить файл целиком.

Справочники

Справочники являются супер-файлами, которые могут содержать файлы или другие справочники. Обычно файлы, содержащиеся в них, устанавливают отношения каким-либо способом. Например, справочник, названный sales может хранить файлы, содержащие цифры ежемесячных продаж, названные jan, feb, mar, и т.д. Вы можете создать каталоги, добавить или удалить файлы из них или удалить каталоги.

Все справочники, которые вы создаете, будут размещены в вашем собственном справочнике. Этот справочник назначается вам системой во время входа в систему. Никто кроме привилегированных пользователей не может читать или записывать файлы в этот справочник без вашего разрешения и вы определяете структуру этого справочника.

Система UNIX также содержит несколько справочников для собственного использования. Структура этих справочников аналогична во всех системах UNIX. Этот справочник, включающий в себя несколько системных справочников, размещается непосредственно под справочником root. Справочник root (обозначенный /) является исходным в файловой структуре UNIX. Все справочники и файлы иерархически располагаются ниже.

Специальные файлы

Специальные файлы соответствуют физическим устройствам, таким как терминал, дисковое устройство или канал связи. Система читает и записывает из/в специальные файлы также как и в обыкновенные файлы. Однако запросы системы на чтение и запись не приводят в действие нормальный механизм доступа к файлу. Вместо этого они активизируют драйвер устройства, связанный с файлом, приводя, возможно, в действие головки диска.

Специальные файлы не хранят данные. Они обеспечивают механизм отображения физических внешних устройств в имена файлов файловой системы. Каждому устройству, поддерживаемому системой, соответствует, по меньшей мере, один специальный файл.

Различаются два типа специальных файлов - *блочные* и *символьные*.

Блочные специальные файлы ассоциируются с такими внешними устройствами, обмен с которыми производится блоками байтов данных, размером 512, 1024, 4096 или 8192 байтов. Типичным примером подобных устройств являются магнитные диски. Файловые системы всегда находятся на блочных устройствах.

Символьные специальные файлы ассоциируются с внешними устройствами, которые не обязательно требуют обмена блоками данных равного размера. Примерами таких устройств являются терминалы (в том числе, системная консоль), последовательные устройства, некоторые

виды магнитных лент. Иногда символьные специальные файлы ассоциируются с магнитными дисками.

При обмене данными с блочным устройством система буферизует данные во внутреннем системном кеше. Через определенные интервалы времени система "выталкивает" буфера, при которых содержится метка "измененный". Основная проблема состоит в том, что при аварийной остановке компьютера (например, при внезапном выключении электрического питания) содержимое системного кеша может быть утрачено. Тогда внешние блочные файлы могут оказаться в рассогласованном состоянии.

Обмены с символьными специальными файлами производятся напрямую, без использования системной буферизации.

Связывание файлов с разными именами.

Файловая система ОС UNIX обеспечивает возможность связывания одного и того же файла с разными именами. Часто имеет смысл хранить под разными именами одну и ту же команду (выполняемый файл) командного интерпретатора.

При символической связи в соответствующем каталоге создается элемент, в котором имени связи сопоставляется некоторое имя файла (этот файл даже не обязан существовать к моменту создания символической связи), создается отдельный i-узел, и даже заводится отдельный блок данных для хранения потенциально длинного имени файла.

Файлы, отображаемые в виртуальную память.

В современных версиях ОС UNIX появилась возможность отображать обычные файлы в виртуальную память процесса с последующей работой с содержимым файла не с помощью системных вызовов read, write и lseek, а с помощью обычных операций чтения из памяти и записи в память.

Для отображения файла в виртуальную память, после открытия файла выполняется системный вызов mmap, действие которого состоит в том, что создается сегмент разделяемой памяти, ассоциированный с открытым файлом, и автоматически подключается к виртуальной памяти процесса. После этого процесс может читать из нового сегмента (реально будут читаться байты, содержащиеся в файле) и писать в него (реально все записи отображаются в файл). При закрытии файла соответствующий сегмент автоматически отключается от виртуальной памяти процесса и уничтожается, если только файл не подключен к виртуальной памяти некоторого другого процесса.

Несколько процессов могут одновременно открыть один и тот же файл и подключить его к своей виртуальной памяти системным вызовом mmap. Тогда любые изменения, производимые путем записи в соответствующий сегмент разделяемой памяти, будут сразу видны другим процессам.

Синхронизация при параллельном доступе к файлам.

Исторически в ОС UNIX всегда применялся очень простой подход к обеспечению параллельного доступа к файлам: система позволяла любому числу процессов одновременно открывать один и тот же файл в любом режиме (чтения, записи или обновления) и не предпринимала никаких синхронизационных действий. Вся ответственность за корректность совместной обработки файла ложилась на использующие его процессы, и система даже не предоставляла каких-либо особых средств для синхронизации доступа процессов к файлу.

В System V.4 появились средства, позволяющие процессам синхронизировать параллельный доступ к файлам. Ядро ОС UNIX поддерживает дополнительный системный вызов fcntl, обеспечивающий такие вспомогательные функции, относящиеся к файловой системе, как получение информации о текущем режиме открытия файла, изменение текущего режима открытия и т.д. С помощью этого системного вызова можно установить монопольную или совместную блокировку файла целиком или заблокировать указанный диапазон байтов внутри файла. Допускаются два варианта синхронизации:

1. с ожиданием, когда требование блокировки может привести к откладыванию процесса до того момента, когда это требование может быть удовлетворено,

2. без ожидания, когда процесс немедленно оповещается об удовлетворении требования блокировки или о невозможности ее удовлетворения в данный момент времени.

Установленные блокировки относятся только к тому процессу, который их установил, и не наследуются процессами-потомками этого процесса. Более того, даже если некоторый процесс пользуется синхронизационными возможностями системного вызова `fcntl`, другие процессы по-прежнему могут работать с тем файлом без всякой синхронизации. Другими словами, это дело группы процессов, совместно использующих файл, - договориться о способе синхронизации параллельного доступа.

Защита файлов.

Как и принято, в многопользовательской операционной системе, в UNIX поддерживается единообразный механизм контроля доступа к файлам и справочникам файловой системы. Любой процесс может получить доступ к некоторому файлу в том и только в том случае, если права доступа, описанные при файле, соответствуют возможностям данного процесса.

Защита файлов от несанкционированного доступа в ОС UNIX основывается на трех фактах.

1. Во-первых, с любым процессом, создающим файл (или справочник), ассоциирован некоторый уникальный в системе идентификатор пользователя (UID - User Identifier), который в дальнейшем можно трактовать как идентификатор владельца вновь созданного файла.
2. Во-вторых, с каждым процессом, пытающимся получить некоторый доступ к файлу, связана пара идентификаторов - текущие идентификаторы пользователя и его группы.
3. В-третьих, каждому файлу однозначно соответствует его описатель - *i*-узел.

Важно понимать, что имена файлов и файлы как таковые - это не одно и то же. В частности, при наличии нескольких жестких связей с одним файлом несколько имен файла реально представляют один и тот же файл и ассоциированы с одним и тем же *i*-узлом. Любому используемому в файловой системе *i*-узлу всегда однозначно соответствует один и только один файл. *I*-узел содержит достаточно много разнообразной информации (большая ее часть доступна пользователям через системные вызовы `stat` и `fstat`), и среди этой информации находится часть, позволяющая файловой системе оценить правомочность доступа данного процесса к данному файлу в требуемом режиме.

Общие принципы защиты одинаковы для всех существующих вариантов системы: Информация *i*-узла включает UID и GID текущего владельца файла (немедленно после создания файла идентификаторы его текущего владельца устанавливаются соответствующими действующим идентификатором процесса-создателя, но в дальнейшем могут быть изменены системными вызовами `chown` и `chgrp`). Кроме того, в *i*-узле файла хранится шкала, в которой отмечено, что может делать с файлом пользователь - его владелец, что могут делать с файлом пользователи, входящие в ту же группу пользователей, что и владелец, и что могут делать с файлом остальные пользователи.

Распределенные файловые системы.

Основная идея распределенной файловой системы состоит в том, чтобы обеспечить совместный доступ к файлам локальной файловой системы для процессов, которые, вообще говоря, выполняются на других компьютерах. В среде ОС UNIX все известные подходы основываются на монтировании удаленной файловой системы к одному из каталогов локальной файловой системы. После выполнения этой процедуры файлы, хранимые в удаленной файловой системе, доступны процессам локального компьютера точно таким же образом, как если бы они хранились на локальном дисковом устройстве.

В принципе, такая схема обладает и достоинствами, и недостатками. К достоинствам, конечно, относится то, что при работе в сети можно экономить дисковое пространство, поддерживая совместно используемые информационные ресурсы только в одном экземпляре. Но, с другой стороны, пользователи удаленной файловой системы неизбежно будут работать с удаленными файлами существенно более медленно, чем с локальными.

3 Базовые механизмы сетевых взаимодействий

Операционная система UNIX с самого своего возникновения была по своей сути сетевой операционной системой.

Потоки (Streams).

В самых ранних вариантах UNIX коммуникационные средства основывались на символьном вводе/выводе, главным образом потому, что аппаратной основой являлись модемы и терминалы. Поскольку такие устройства являются относительно медленными, в ранних вариантах не требовалось особенно заботиться о модульности и эффективности программного обеспечения.

С появлением многоуровневых сетевых протоколов, таких как TCP/IP (Transmission Control Protocol/Internet Protocol), SNA (IBM's System Network Architecture), OSI (Open Systems Internetworking), X.25 и др. стало понятно, что в ОС UNIX требуется некоторая общая основа организации сетевых средств, основанных на многоуровневых протоколах.

Во многом эта проблема была решена компанией AT&T, которая предложила и реализовала механизм потоков (STREAMS), обеспечивающий гибкие и модульные возможности для реализации драйверов устройств и коммуникационных протоколов.

Streams представляют собой связанный набор средств общего назначения, включающий системные вызовы и подпрограммы, а также ресурсы ядра. В совокупности эти средства обеспечивают стандартный интерфейс символьного ввода/вывода внутри ядра, а также между ядром и соответствующими драйверами устройств, предоставляя гибкие и развитые возможности разработки и реализации коммуникационных сервисов. При этом механизм потоков не навязывает какой-либо конкретной архитектуры сети и/или конкретных протоколов. Как и любой другой драйвер устройства, потоковый драйвер представляется специальным файлом файловой системы со стандартным набором операций: open, close, read, write и т.д. Когда пользовательский процесс открывает потоковое устройство, пользуясь системным вызовом open, ядро связывает с драйвером заголовок потока. После этого пользовательский процесс общается с заголовком потока так, как если бы он представлял собой обычный драйвер устройства. Другими словами, заголовок потока отвечает за обработку всех системных вызовов, производимых пользовательским процессом по отношению к потоковому драйверу. Если процесс выполняет запись в устройство (системный вызов write), заголовок потока передает данные драйверу устройства в нисходящем направлении. Аналогично, при реализации чтения из устройства (системный вызов read) драйвер устройства передает данные заголовку потока в восходящем направлении.

В описанной схеме данные между заголовком потока и драйвером устройства передаются в неизменяемом виде без какой-либо промежуточной обработки. Однако можно добиться того, чтобы данные подвергались обработке при передаче их в любом направлении, если включить в поток между заголовком и драйвером устройства один или несколько потоковых модулей. ***Потоковый модуль** является обработчиком данных, выполняющим определенный набор функций над данными по мере их прохождения по потоку.* Простейшими примерами потокового модуля являются разного рода перекодировщики символьной информации. Более сложным примером является потоковый модуль, осуществляющий разборку нисходящих данных в пакеты для их передачи по сети и сборку восходящих данных с удалением служебной информации пакетов.

Каждый потоковый модуль является, вообще говоря, независимым от присутствия в потоке других модулей, обрабатывающих данные. Данные могут подвергаться обработке произвольным числом потоковых модулей, пока в конце концов не достигнут драйвера устройств при движении в нисходящем направлении или заголовка потока при движении в восходящем направлении. *Для передачи данных от заголовка к драйверу или модулю, от одного модуля другому и от драйвера или модуля к заголовку потока используется механизм сообщений.* Каждое сообщение представляет собой набор блоков сообщения, каждый из которых состоит из

- заголовка,
- блока данных
- буфера данных.

Стек протоколов TCP/IP.

TCP/IP (Transmission Control Protocol/Internet Protocol) представляет собой семейство протоколов, основным назначением которых является обеспечение возможности полезного сосуществования компьютерных сетей, основанных на разных технологиях.

В UNIX System V Release 4 протокол TCP/IP реализован как набор потоковых модулей плюс дополнительный компонент TLI (Transport Level Interface - Интерфейс транспортного уровня). TLI является интерфейсом между прикладной программой и транспортным механизмом. Приложение, пользующееся интерфейсом TLI, получает возможность использовать TCP/IP.

Интерфейс TLI основан на использовании классической семиуровневой модели ISO/OSI, которая разделяет сетевые функции на семь областей, или уровней. Цель модели в обеспечении стандарта сетевой связи компьютеров независимо от производителя аппаратуры компьютеров и/или сети. Семь уровней модели можно кратко описать следующим образом.

Уровень 1: Физический уровень (Physical Level) - среда передачи (например, Ethernet). Уровень отвечает за передачу неструктурированных данных по сети.

Уровень 2: Канальный уровень (Data Link Layer) - уровень драйвера устройства, называемый также уровнем ARP/RARP в TCP/IP. Этот уровень, в частности, отвечает за преобразование данных при исправлении ошибок, происходящих на физическом уровне.

Уровень 3: Сетевой уровень (Network Level) - отвечает за выполнение промежуточных сетевых функций, таких как поиск коммуникационного маршрута при отсутствии возможности прямой связи между узлом-отправителем и узлом-получателем. В TCP/IP этот уровень соответствует протоколам IP и ICMP.

Уровень 4: Транспортный уровень (Transport Level) - уровень протоколов TCP/IP или UDP/IP семейства протоколов TCP/IP. Уровень отвечает за разборку сообщения на фрагменты (пакеты) при передаче и за сборку полного сообщения из пакетов при приеме таким образом, что на более старших уровнях модели эти процедуры вообще незаметны. Кроме того, на этом уровне выполняется посылка и обработка подтверждений и, при необходимости, повторная передача.

Уровень 5: Уровень сессий (Session Layer) - отвечает за управление переговорами взаимодействующих транспортных уровней. В NFS (Network File System - Сетевая файловая система) этот уровень используется для реализации механизма вызовов удаленных процедур (RPC - Remote Procedure Calls,).

Уровень 6: Уровень представлений (Presentation Layer) - отвечает за управление представлением информации. В NFS на этом уровне реализуется механизм внешнего представления данных (XDR - External Data Representation), машинно-независимого представления, понятного для всех компьютеров, входящих в сеть.

Уровень 7: Уровень приложений - интерфейс с такими сетевыми приложениями, как telnet, rlogin, mail и т.д.

Интерфейс TLI соответствует трем старшим уровням этой модели (с пятого по седьмой) и позволяет прикладному процессу пользоваться сервисами сети (без необходимости знать о деталях транспортного и более низких уровней).

3.7 Управление памятью

Оперативная память всегда была и остается до сих пор наиболее критическим ресурсом компьютеров. Операционная система UNIX начинала свое существование с применения очень простых методов управления памятью (простой своппинг). В современных вариантах системы для управления памятью применяются следующие механизмы.

Виртуальная память.

В ОС UNIX используются страничная, сегментная и сегментно-страничная организация виртуальной памяти.

Как правило, все таблицы страниц хранятся в основной памяти, а быстрота доступа к элементам таблицы текущей виртуальной памяти достигается за счет наличия сверхбыстродействующей буферной памяти (кэша).

Существует большое количество разнообразных алгоритмов подкачки применяемых в ОС UNIX.

Во-первых, алгоритмы подкачки делятся на

- глобальные
- локальные.

При использовании *глобальных алгоритмов* операционная система при потребности замещения ищет страницу основной памяти среди всех страниц, независимо от их принадлежности к какой-либо виртуальной памяти.

Локальные алгоритмы предполагают, что если возникает требование доступа к отсутствующей в основной памяти странице виртуальной памяти ВП1, то страница для замещения будет искаться только среди страниц основной памяти, приписанных к той же виртуальной памяти ВП1.

Наиболее распространенными традиционными алгоритмами (как в глобальном, так в локальном вариантах) являются алгоритмы FIFO (First In First Out) и LRU (Least Recently Used). При использовании алгоритма FIFO для замещения выбирается страница, которая дольше всего остается приписанной к виртуальной памяти. Алгоритм LRU предполагает, что замещать следует ту страницу, к которой дольше всего не происходили обращения. Хотя интуитивно кажется, что критерий алгоритма LRU является более правильным, известны ситуации, в которых алгоритм FIFO работает лучше (и, кроме того, он гораздо более дешево реализуется).

В 1968 году американский исследователь Питер Деннинг сформулировал принцип локальности ссылок (называемый принципом Деннинга) и выдвинул идею алгоритма подкачки, основанного на понятии *рабочего набора*. Идея алгоритма подкачки Деннинга (иногда называемого алгоритмом рабочих наборов) состоит в том, что операционная система в каждый момент времени должна обеспечивать наличие в основной памяти текущих рабочих наборов всех процессов, которым разрешена конкуренция за доступ к процессору.

Аппаратно-независимый уровень управления памятью.

Одним из достижений ОС UNIX является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части.

ОС UNIX опирается на некоторое собственное представление организации виртуальной памяти, которое используется в аппаратно-независимой части подсистемы управления виртуальной памятью и связывается с конкретной аппаратной реализацией с помощью аппаратно-зависимой части. В чем же состоит это абстрактное представление виртуальной памяти?

Во-первых, виртуальная память каждого процесса представляется в виде набора сегментов

Виртуальная память процесса ОС UNIX разбивается на сегменты пяти разных типов. Три типа сегментов обязательны для каждой виртуальной памяти, и сегменты этих типов присутствуют в виртуальной памяти в одном экземпляре для каждого типа

Сегмент программного кода содержит только команды. Реально в него помещается соответствующий сегмент выполняемого файла, который указывался в качестве параметра системного вызова `exec` для данного процесса. Сегмент программного кода не может модифицироваться в ходе выполнения процесса и потому возможно использование одного экземпляра кода для разных процессов.

Сегмент данных содержит инициализированные и неинициализированные статические переменные программы, выполняемой в данном процессе (на этом уровне изложения под статическими переменными лучше понимать области виртуальной памяти, адреса которых фиксируются в программе при ее загрузке и действуют на протяжении всего ее выполнения). Понятно, что поскольку речь идет о переменных, содержимое сегмента данных может изменяться в ходе выполнения процесса, следовательно, к сегменту должен обеспечиваться доступ и по чтению, и по записи. С другой стороны, нельзя разрешить нескольким процессам совместно использовать один и тот же сегмент данных (по причине несогласованного изменения одних и тех же переменных разными процессами ни один из них не мог бы успешно завершиться).

Сегмент стека - это область виртуальной памяти, в которой размещаются автоматические переменные программы, явно или неявно в ней присутствующие. Этот сегмент, очевидно, должен быть динамическим (т.е. доступным и по чтению, и по записи), и он, также очевидно, должен быть частным (приватным) сегментом процесса.

Разделяемый сегмент виртуальной памяти образуется при подключении к ней сегмента разделяемой памяти. По определению, такие сегменты предназначены для координированного совместного использования несколькими процессами. Поэтому разделяемый сегмент должен допускать доступ по чтению и по записи и может разделяться несколькими процессами.

Сегменты файлов, отображаемых в виртуальную, представляют собой разновидность разделяемых сегментов. Разница состоит в том, что если при необходимости освободить оперативную память страницы разделяемых сегментов копируются ("откачиваются") в специальную системную область подкачки (swapping space) на диске, то *страницы сегментов файлов, отображаемых в виртуальную память, в случае необходимости откачиваются прямо на свое место в области внешней памяти, занимаемой файлом*. Такие сегменты также допускают доступ и по чтению, и по записи и являются потенциально совместно используемыми.

На аппаратно-независимом уровне сегментная организация виртуальной памяти каждого процесса описывается структурой as, которая содержит указатель на список описателей сегментов, общий текущий размер виртуальной памяти (т.е. суммарный размер всех существующих сегментов), текущий размер физической памяти, которую процесс занимает в данный момент времени, и наконец, указатель на некоторую аппаратно-зависимую структуру, данные которой используются при отображении виртуальных адресов в физические. Описатель каждого сегмента содержит

- индивидуальные характеристики сегмента, в том числе, виртуальный адрес начала сегмента (каждый сегмент занимает некоторую непрерывную область виртуальной памяти),
- размер сегмента в байтах,
- список операций, которые можно выполнять над данным сегментом,
- статус сегмента (например, в каком режиме к нему возможен доступ, допускается ли совместное использование и т.д.),
- указатель на таблицу описателей страниц сегмента и т.д.
- Кроме того, описатель каждого сегмента содержит прямые и обратные ссылки по списку описателей сегментов данной виртуальной памяти и ссылку на общий описатель виртуальной памяти as.

На уровне страниц поддерживается два вида описательных структур. Для каждой страницы физической оперативной памяти существует описатель, входящий в один из трех списков:

Первый список

включает описатели страниц, не допускающих модификации или отображаемых в область внешней памяти какого-либо файла (например, страницы сегментов программного кода или страницы сегмента файла, отображаемого в виртуальную память). Для таких страниц не требуется пространство в области подкачки системы; они либо вовсе не требуют откачки (перемещения копии во внешнюю память), либо откачка производится в другое место.

Второй список

это список описателей свободных страниц, т.е. таких страниц, которые не подключены ни к одной виртуальной памяти. Такие страницы свободны для использования и могут быть подключены к любой виртуальной памяти.

Третий список страниц

включает описатели так называемых анонимных страниц, т.е. таких страниц, которые могут изменяться, но для которых нет "родного" места во внешней памяти.

В любом описателе физической страницы сохраняются копии признаков обращения и модификации страницы, вырабатываемых конкретной используемой аппаратурой.

Для каждого сегмента поддерживается таблица отображения, связывающая адреса входящих в него виртуальных страниц с описателями соответствующих им физических страниц из первого или третьего списков описателей физических страниц для виртуальных страниц, присутствующих в основной памяти, или с адресами копий страниц во внешней памяти для виртуальных страниц, отсутствующих в основной памяти. (Правильнее сказать, что поддерживается отдельная таблица отображения для каждого частного сегмента и одна общая таблица отображения для каждого разделяемого сегмента.)

Страничное замещение основной памяти и swapping.

В ОС UNIX используется некоторый облегченный вариант алгоритма подкачки, основанный на использовании понятия рабочего набора. Основная идея заключается в оценке рабочего набора процесса на основе использования аппаратно (а в некоторых реализациях - программно) устанавливаемых признаков обращения к страницам основной памяти.

Периодически для каждого процесса производятся следующие действия:

- Просматриваются таблицы отображения всех сегментов виртуальной памяти этого процесса. Если элемент таблицы отображения содержит ссылку на описатель физической страницы, то анализируется признак обращения.
- Если признак установлен, то страница считается входящей в рабочий набор данного процесса, и сбрасывается в нуль счетчик старения данной страницы. Если признак не установлен, то к счетчику старения добавляется единица, а страница приобретает статус кандидата на выход из рабочего набора процесса.
- Если при этом значение счетчика достигает некоторого критического значения, страница считается вышедшей из рабочего набора процесса, и ее описатель заносится в список страниц, которые можно откатить (если это требуется) во внешнюю память. По ходу просмотра элементов таблиц отображения в каждом из них признак обращения гасится.

Откачку страниц, не входящих в рабочие наборы процессов, производит специальный системный процесс-stealer. Он начинает работать, когда количество страниц в списке свободных страниц достигает установленного нижнего порога. Функцией этого процесса является анализ необходимости откачки страницы (на основе признака изменения) и запись копии страницы (если это требуется) в соответствующую область внешней памяти (т.е. либо в системную область подкачки - swapping space для анонимных страниц, либо в некоторый блок файловой системы для страниц, входящей в сегмент отображаемого файла).

Рабочий набор любого процесса может изменяться во время его выполнения. Другими словами, возможна ситуация, когда процесс обращается к виртуальной странице, отсутствующей в основной памяти. В этом случае, как обычно, возникает аппаратное прерывание, в результате которого начинает работать операционная система. Дальнейший ход событий зависит от обстоятельств. Если список описателей свободных страниц не пуст, то из него выбирается некоторый описатель, и соответствующая страница подключается к виртуальной памяти процесса (конечно, после считывания из внешней памяти содержимого копии этой страницы, если это требуется).

Но если возникает требование страницы в условиях, когда список описателей свободных страниц пуст, то начинает работать механизм своппинга. Основной повод для применения другого механизма состоит в том, что простое отнятие страницы у любого процесса (включая тот, который затребовал бы страницу) потенциально вело бы к ситуации thrashing, поскольку разрушало бы рабочий набор некоторого процесса). Любой процесс, затребовавший страницу не из своего текущего рабочего набора, становится кандидатом на своппинг. Ему больше не предоставляются ресурсы процессора, и описатель процесса ставится в очередь к системному процессу-swapper. Конечно, в этой очереди может находиться несколько процессов. Процесс-swapper по очереди осуществляет полный своппинг этих процессов (т.е. откачку всех страниц их виртуальной памяти, которые присутствуют в основной памяти), помещая соответствующие описатели физических страниц в список свободных страниц, до тех пор, пока количество страниц в этом списке не достигнет установленного в системе верхнего предела. После завершения полного своппинга каждого процесса одному из процессов из очереди к процессу-swapper дается возможность попытаться продолжить свое выполнение (в расчете на то, что свободной памяти уже может быть достаточно).

В последней "фактически стандартной" версии ОС UNIX (System V Release 4) используется более упрощенный алгоритм. Это глобальный алгоритм, в котором вероятность thrashing погашается за счет своппинга. Используемый алгоритм называется NRU (Not Recently Used) или clock. Смысл алгоритма состоит в том, что процесс-stealer периодически очищает признаки обращения всех страниц основной памяти, входящих в виртуальную память процессов (отсюда название "clock"). Если возникает потребность в откачке (т.е. достигнут нижний предел размера

списка описателей свободных страниц), то stealer выбирает в качестве кандидатов на откачку прежде всего те страницы, к которым не было обращений по записи после последней "очистки" и у которых нет признака модификации (т.е. те, которые можно дешевле освободить). Во вторую очередь выбираются страницы, которые действительно нужно откачивать. Параллельно с этим работает описанный выше алгоритм своппинга, т.е. если возникает требование страницы, а свободных страниц нет, то соответствующий процесс становится кандидатом на своппинг.

Распределение памяти

Ядро постоянно располагается в оперативной памяти, наряду с выполняющимся в данный момент процессом (или частью его, по меньшей мере). В процессе компиляции программа-компилятор генерирует последовательность адресов, являющихся адресами переменных и информационных структур, а также адресами инструкций и функций. Компилятор генерирует адреса для виртуальной машины так, словно на физической машине не будет выполняться параллельно с транслируемой ни одна другая программа.

Когда программа запускается на выполнение, ядро выделяет для нее место в оперативной памяти, при этом совпадение виртуальных адресов, сгенерированных компилятором, с физическими адресами совсем необязательно. Ядро, взаимодействуя с аппаратными средствами, транслирует виртуальные адреса в физические, т.е. отображает адреса, сгенерированные компилятором, в физические, машинные адреса. Такое отображение опирается на возможности аппаратных средств, поэтому компоненты системы UNIX, занимающиеся им, являются машинно-зависимыми. Например, отдельные вычислительные машины имеют специальное оборудование для подкачки выгруженных страниц памяти.

Упражнения к лекции 1.

1. Принципы защиты ОС Unix.
2. Управление устройствами ОС Unix.
3. Драйверы устройств ОС Unix.
4. Внешний и внутренний интерфейсы устройств ОС Unix.
5. Программные гнезда (Sockets) ОС Unix.
6. Вызовы удаленных процедур (RPC) ОС Unix.
7. Из каких компонентов состоит ОС Unix.
8. Основные функции ОС Unix.
9. Основные функции shell ОС Unix.
10. Основные функции команд ОС Unix.
11. Основные функции файловой системы ОС Unix.
12. За счет чего достигается мобильность ОС Unix?
13. Что включает в себя машинно-зависимая часть ядра ОС UNIX?
14. Какие основные функции ядра ОС Unix?
15. Какие отличия между пользовательским режимом и режимом ядра в ОС Unix?
16. Принципы взаимодействия с ядром. Что такое системные вызовы?

Литература к лекции

- 3.1 А.В.Гордеев, А.Ю.Молчанов. Системное программное обеспечение. — "Питер", 2002. — 736с.
- 3.2 Кристиан К. Введение в операционную систему Unix: пер. с англ. — М. Финансы и статистика, 1985. — 360с.
- 3.3 Робачевский А.М., Немнюгин С.А., Стесик О.Л. Операционная система Unix. 2-е изд.— СПб.: БХВ — Петербург, 2005. — 635с.