Оглавление

Операционная система. Функции операционных систем.

Операцио́нная систе́ма - базовый комплекс **компьютерных программ**, обеспечивающий управление аппаратными средствами **компьютера**, работу с **файлами**, ввод и вывод данных, а также выполнение **прикладных программ** и **утилит**.

Функции:

1) запуск и исполнение программ
2) управление памятью
3) работа с периферийными устройствами
4) поддержка пользовательского интерфейса

(1)-(4) функции присущи дисковым ОС. Универсальные ОС обладают еще следующими функциями, связанными с многозадачностью и многопользовательностью:

5) параллельное выполнение нескольких задач
6) распределение ресурсов компьютера между задачами
7) организация взаимодействия задач друг с другом
8) взаимодействие пользовательских программ с нестандартными внешними устройствами
9) организация межмашинного взаимодействия и разделения ресурсов
10) защита системных ресурсов, данных и программ пользователя

Виды операционных систем

ДОС (Дисковые ОС)

Это системы, берущие на себя выполнение только первых четырех функций. Как правило, это просто некий резидентный набор подпрограмм, не более того. Он загружает пользовательскую программу в память и передает ей управление, после чего программа делает с системой все, что ей заблагорассудится. Считается желательным, чтобы после завершения программы машина оставалась в таком состоянии, чтобы ДОС могла продолжить работу. Если же программа приводит машину в какое-то другое состояние, ДОС ничем ей в этом не может помешать. Характерный пример - различные загрузочные мониторы для машин класса *Spectrum*. Как правило, такие системы работают одновременно только с одной программой.

Дисковая операционная система *MS DOS* для *IBM PC*-совместимых машин является прямым наследником одного из таких резидентных мониторов.

Существование систем такого класса обусловлено их простотой и тем, что они потребляют мало ресурсов.

Примеры: MS DOS, CP/M, Win 95/98

Универсальные ОС

К этому классу относятся системы, берущие на себя выполнение всех вышеперечисленных функций. Разделение на ОС и ДОС идет, по-видимому, от систем IBM DOS/360 и OS/360 для больших компьютеров этой фирмы, клоны которых известны у нас в стране под названием ЕС ЭВМ серии 10XX.

Здесь под ОС мы будем подразумевать системы ``общего назначения'', то есть рассчитанные на интерактивную работу одного

или нескольких пользователей в режиме разделения времени, при не очень жестких требованиях на время реакции системы на внешние события. Как правило, в таких системах уделяется большое внимание защите самой системы, программного обеспечения и пользовательских данных от ошибочных и злонамеренных программ и пользователей. Обычно такие системы используют встроенные в архитектуру процессора средства защиты и виртуализации памяти. К этому классу относятся такие широко распространенные системы, как VAX/VMS, системы семейства Unix и OS/2, хотя последняя не обеспечивает одновременной работы нескольких пользователей и защиты пользователей друг от друга.

Системы реального времени

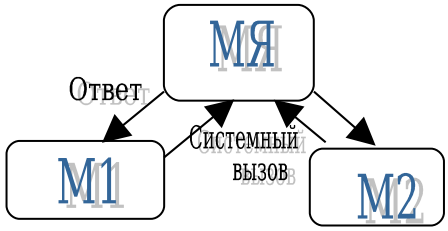
Это системы, предназначенные для облегчения разработки так называемых приложений реального времени. Это программы, управляющие некомпьютерным по природе оборудованием, часто с очень жесткими ограничениями по времени. Такие системы обязаны поддерживать многопроцессорность, гарантированное время реакции на внешнее событие, простой доступ к таймеру и внешним устройствам. Такие системы могут по другим признакам относиться как к классу ДОС (RT-11), так и к ОС (OS-9, QNX)

Микроядерные ОС

В отличие от традиционной архитектуры, согласно которой операционная система представляет собой монолитное ядро, реализующие основные функции по управлению аппаратными ресурсами и организующее среду для выполнения пользовательских процессов, микроядерная архитектура распределяет функции ОС между микроядром и входящими в состав ОС системными сервисами, реализованными в виде процессов, равноправных с пользовательскими приложениями.

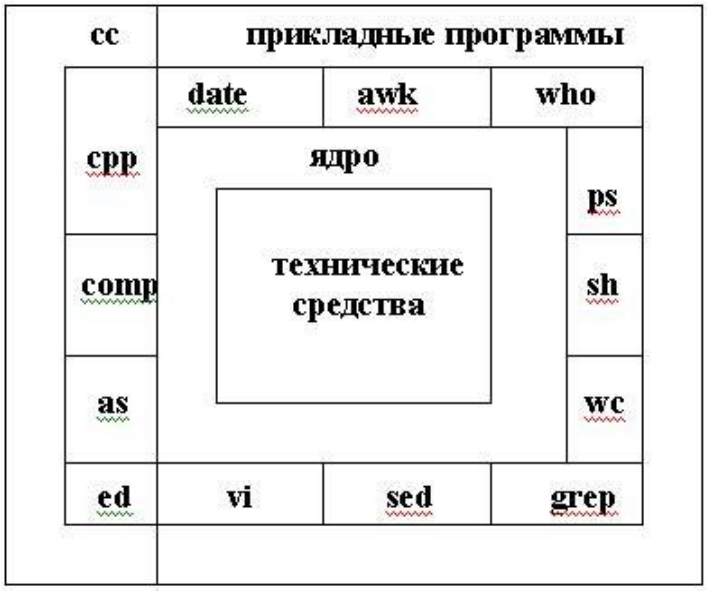
В основе архитектуры микроядерных ОС лежат следующие базовые концепции:

- минимизация набора функций, поддерживаемых микроядром, и реализация традиционных функций ОС (файловая система, сетевая поддержка) вне микроядра;
- организация синхронного и асинхронного взаимодействия между процессами исключительно через механизм обмена сообщениями;
- все отношения между компонентами строятся на основе модели клиент/сервер;
- применение объектно-ориентированного подхода при разработке архитектуры и программировании системы.



Архитектура UNIX.

На Рисунке изображена архитектура верхнего уровня системы UNIX.



Технические средства, показанные в центре диаграммы, выполняют функции, обеспечивающие функционирование операционной системы. Операционная система взаимодействует с аппаратурой непосредственно обеспечивая обслуживание программ и их независимость от деталей аппаратной конфигурации. Если представить систему состоящей из пластов, в ней можно выделить системное ядро, изолированное от пользовательских программ. Поскольку программы не зависят от аппаратуры, их легко переносить из одной системы UNIX в другую, функционирующую на другом комплексе технических средств, если только в этих программах не подразумевается работа с конкретным оборудованием. Например, программы, рассчитанные на определенный размер машинного слова, гораздо труднее переводить на другие машины по сравнению с программами, не требующими подобных установлений.

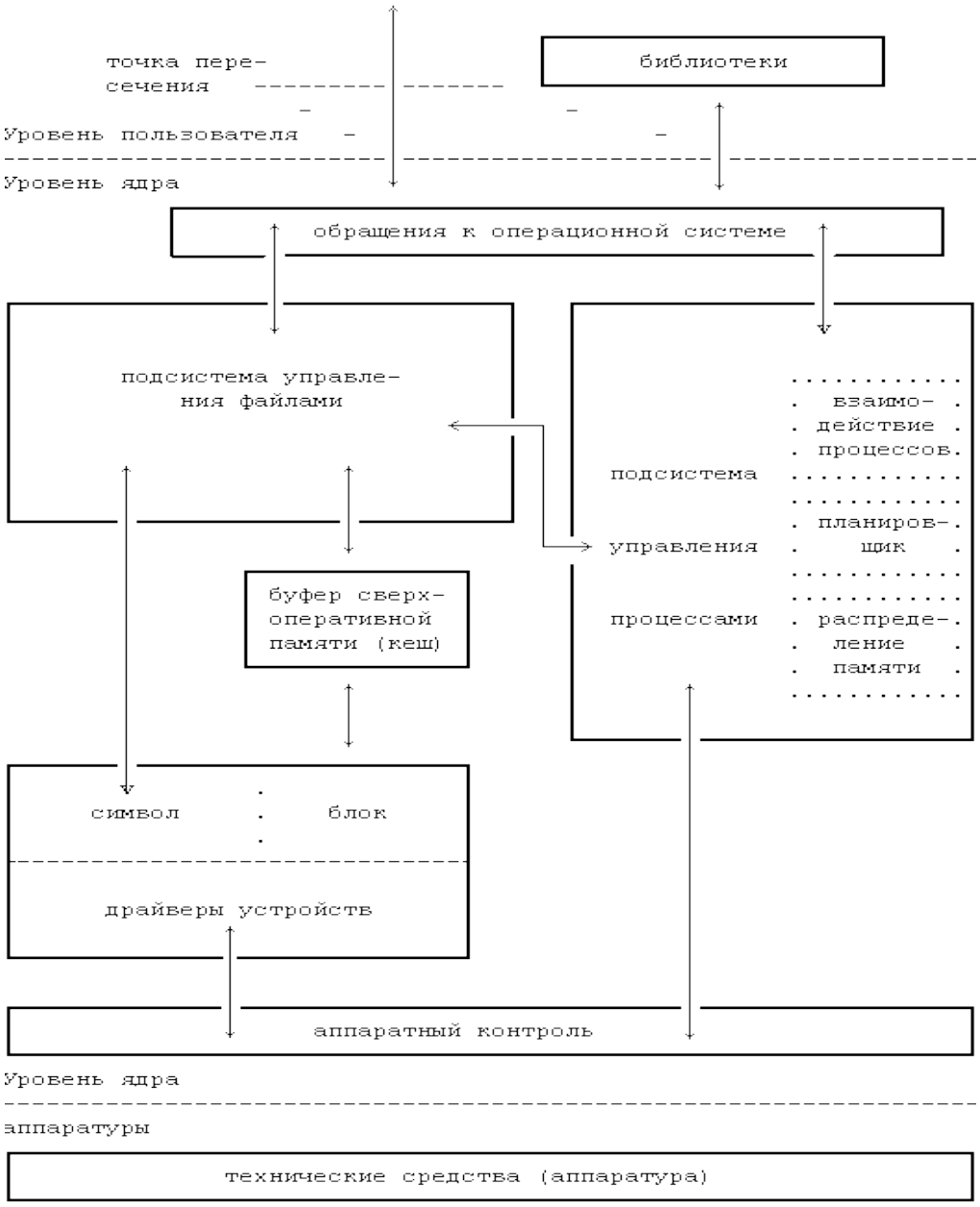
Программы, подобные командному процессору shell и редакторам (ed и vi) и показанные на внешнем по отношению к ядру слое, взаимодействуют с ядром при помощи хорошо определенного набора обращений к операционной системе. Обращения к операционной системе понуждают ядро к выполнению различных операций, которых требует вызывающая программа, и обеспечивают обмен данными между ядром и программой.

Некоторые из программ, приведенных на рисунке, в стандартных конфигурациях системы известны как команды, однако на одном уровне с ними могут располагаться и доступные пользователю программы, такие как программа a.out, стандартное имя для исполняемого файла, созданного компилятором с языка Си. Другие прикладные программы располагаются выше указанных программ, на верхнем уровне, как это показано на рисунке. Например, стандартный компилятор с языка Си ('cc') располагается на самом внешнем слое: он вызывает препроцессор для Си, ассемблер и загрузчик (компоновщик), т.е. отдельные программы предыдущего уровня.

Хотя на рисунке приведена двухуровневая иерархия прикладных программ, пользователь может расширить иерархическую структуру на столько уровней, сколько необходимо. В самом деле, стиль программирования, принятый в системе UNIX, допускает разработку комбинации программ, выполняющих одну и ту же, общую задачу.

Многие прикладные подсистемы и программы, составляющие верхний уровень системы, такие как командный процессор shell, редакторы, SCCS(система обработки исходных текстов программ) и пакеты программ подготовки документации, постепенно становятся синонимом понятия 'система UNIX'. Однако все они пользуются услугами программ нижних уровней и в конечном счете ядра с помощью набора обращений к операционной системе. Они имеют несложные параметры, что облегчает их использование, предоставляя при этом большие возможности пользователю. Набор обращений к операционной системе вместе с реализующими их внутренними алгоритмами составляют 'тело' ядра.

Блок – схема ядра



Ядро - внутренние алгоритмы и структуры, составляющие основу операционной системы. Ядро реализует функции, на которых основывается выполнение всех прикладных программ в операционной системе, и им же определяются эти функции.

Функции ядра :

1. управление и выполнение процессов

2. планирование очередности, предоставление времени ЦП

3. выделение оперативной памяти процессу

4. работа с внешней памятью

5. управление доступом к периферии

Файловые системы

TAR

Имя файла	Данные	Имя файла	Данные	Имя файла	Данные	…

Недостаток: для поиска нужных данных необходимо просматривать весь архив.

RT 11

Имя файла	Адрес 1	Имя файла	Адрес 2	Имя файла	Адрес 3	…

Данные 1	Данные 2	Данные 3

Недостатки:

- низкая скорость доступа
- не может просматривать все ячейки
- не подразумевает фрагментации

FAT

Файловая система FAT не может контролировать отдельно каждый сектор, поэтому она объединяет смежные сектора в кластеры. Таким образом, уменьшается общее количество единиц хранения, за которыми должна следить файловая система. Размер кластера в FAT является степенью двойки и определяется размером тома при форматировании диска. Кластер представляет собой минимальное пространство, которое может занимать файл. Это приводит к тому, что часть пространства диска расходуется впустую.

Номера блоков диска		
1		
2	10	
3	11	Начало файла F_2
4		
5	EOF	
6	2	Начало файла F_1
7	EOF	
8		
9		
10	7	
11	5	

FAT - плюсы:

- Для эффективной работы требуется немного оперативной памяти.
- Быстрая работа с малыми и средними каталогами.
- Диск совершает в среднем меньшее количество движений головок (в сравнении с ntfs).
- Эффективная работа на медленных дисках.
- При выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла

FAT - минусы:

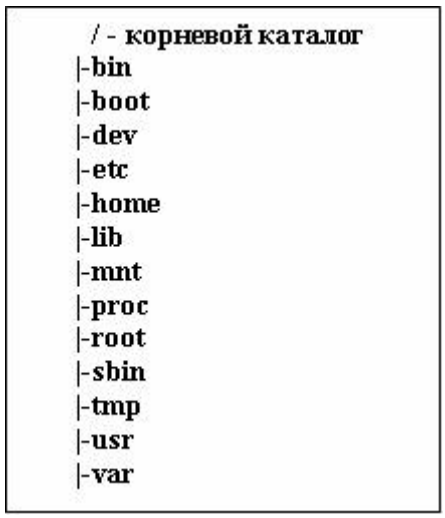
- Катастрофическая потеря быстродействия с увеличением фрагментации, особенно для больших дисков (только FAT32).
- Сложности с произвольным доступом к большим (скажем, 10% и более от размера диска) файлам.
- Очень медленная работа с каталогами, содержащими большое количество файлов.
- Необходимость хранения в памяти этой довольно большой таблицы.

Файловая система S5FS

(файловая система Unix System V)
Файловая система характеризуется:
- иерархической структурой
- согласованной обработкой массивов данных
- возможностью создания и удаления файлов
- динамическим расширением файлов
- защитой информации в файлах
- трактовкой периферийных устройств(таких, как терминалы и ленточные устройства) как файлов.

Файловая система организована в виде дерева с одной исходной вершиной, которая называется корнем (записывается: '/'); каждая вершина в древовидной структуре файловой системы, кроме листьев, является каталогом файлов.

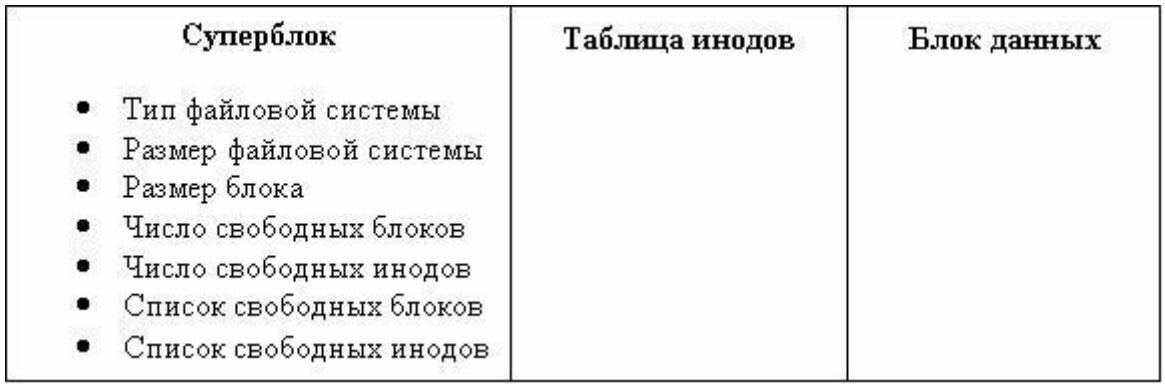
Корневой каталог имеет вид:



bin	Двоичные коды наиболее важных команд
boot	Статические файлы загрузчика boot
dev	Файлы устройств
etc	Файлы настройки конфигурации системы
home	Домашние каталоги пользователей
lib	Разделяемые библиотеки
mnt	Точка монтирования временно подключаемых систем
proc	Псевдо-файловая система с информацией о процессах
root	Домашний каталог (пользователя) root
sbin	Наиболее важные системные двоичные коды
tmp	Временные файлы
usr	Вторая главная иерархия
var	Переменные данные

Каждый из этих, а также расположенных на других уровнях, каталогов имеет строго определенное назначение, что обеспечивает удобство работы с файловой системой.

Устройство файловой системы:



В суперблоке содержится информация о системе в целом, т.е. тип, размер, размер блока (512, 1024, 2048), число свободных блоков, число свободных инодов, список свободных блоков, список свободных инодов.

В списке свободных инодов находится ограниченное число инодов. В этом списке содержатся ссылки на иноды в таблице инодов. По мере захвата инодов можно перечитать таблицу инодов с целью её

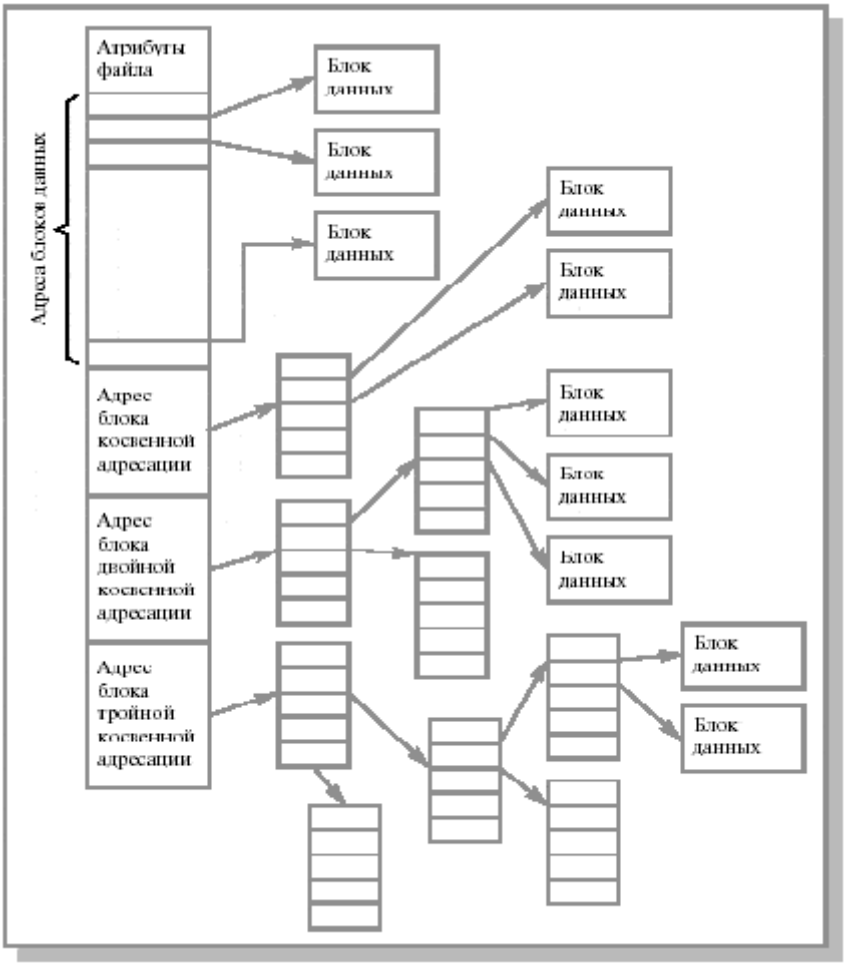
заполнения. Когда список инодов пустеет, тогда ОС просматривает таблицу инодов и пополняет его.

В списке свободных блоков содержатся ссылки на все свободные блоки данных.

Инод – запись содержащая информацию о файле и его размещении.

Инод содержит информацию о: типе файла, времени доступа и последней модификации файла, атрибутах(чтение, запись, исполнение), идентификаторах пользователя и группы(UID, GID), размере. Там также содержатся прямые и косвенные ссылки на блоки данных. Прямой ссылкой можно адресовать файл в 1кб. Косвенная ссылка содержит ссылку на блок, содержащийся в области блоков данных, который содержит ссылки на блоки ей можно адресовать 256 кб. Ссылка двойной косвенности содержит ссылку на блок, который содержит ссылки на блоки, которые содержат ссылки на блоки. Адресовать можно около 50 мб (256*256).

В иноде указывается число ссылок на этот инод. Т.е. сколько имен у этого файла. Сами же имена файлов содержатся с специализированных файлах (называемых директории или каталоги). В директории 14 кб отводится под имя файла и 2 кб на ссылку на номер инода.



Плюсы:

При фиксированном, относительно небольшом размере индексного узла поддерживать работу с файлами, размер которых может меняться от нескольких байтов до нескольких гигабайтов. Для маленьких файлов используется только прямая адресация, обеспечивающая максимальную производительность.

Минусы:

Уязвимость суперблока.
Список свободных блоков сделан неудобно.
Имя файла имеет фиксированную длину.

Файловая система ext2

Основные компоненты файловой системы ext2

Как и в любой файловой системе UNIX, в составе файловой системы ext2 можно выделить следующие составляющие:
- блоки и группы блоков;
- информационный узел (information node);
- суперблок (superblock);

Блоки и группы блоков
Все пространство раздела диска разбивается на блоки фиксированного размера, кратные размеру сектора - 1024, 2048 и 4096 байт. Размер блока указывается при создании файловой системы на разделе диска. Меньший размер блока позволяет экономить место на жестком диске, но также ограничивает максимальный размер файловой системы. Все блоки имеют порядковые номера.
Информационный узел
Базовым понятием файловой системы является информационный узел или inode. Это специальная структура, которая содержит информацию об атрибутах и физическом расположении файла. Атрибутами файла являются его тип (обычный файл, каталог и т.д.), права доступа к нему, идентификатор владельца, размер, время создания.
Суперблок

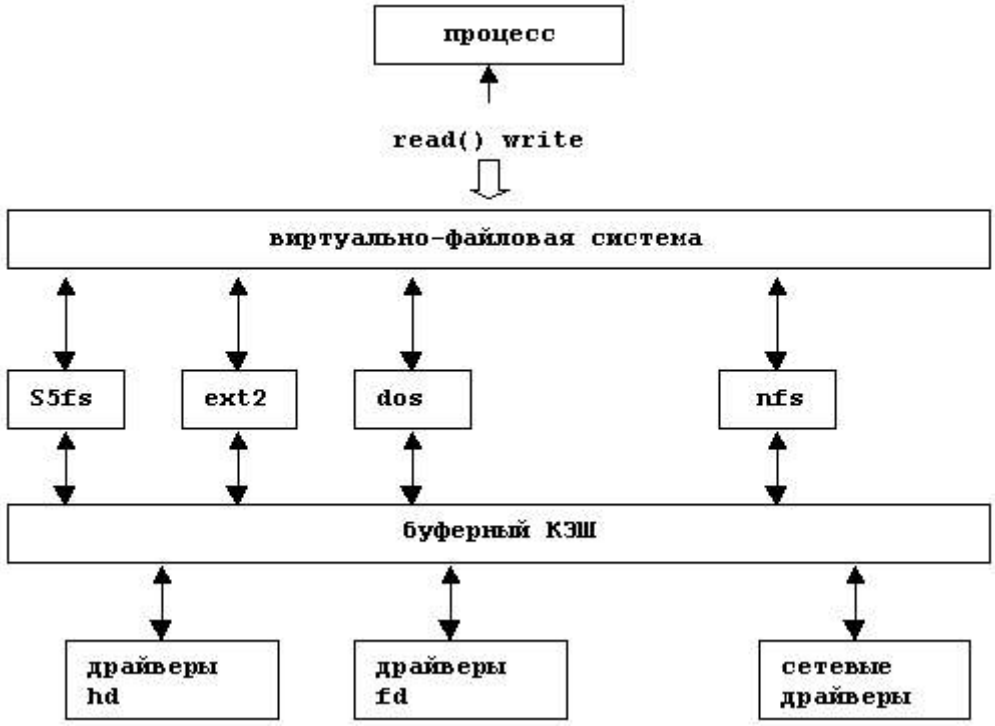
Суперблок - основной элемент файловой системы ext2. Он содержит следующую информацию о файловой системе (список не полный):
- общее число блоков и inode-ов в файловой системе
- число свободных блоков и inode-ов в файловой системе
- размер блока файловой системы
- количество блоков и inode-ов в группе
- размер inode-а

- идентификатор файловой системы
- номер первого блока данных. Другими словами, это номер блока, содержащего суперблок. Этот номер всегда равен 0, если размер блока файловой системы больше 1024 байт, и 1, если размер блока равен 1024 байт

Файловая система VFS

Виртуальная файловая система.

VFS содержит набор функций, которые должна поддерживать любая файловая система. Этот интерфейс состоит из ряда операций, которые оперируют тремя типами объектов: файловые системы, индексные дескрипторы и открытые файлы.



VFS содержит информацию о всех типах поддерживаемых файловых систем. Здесь используется таблица, которая создается во время компиляции ядра. Каждая запись в такой таблице содержит тип файловой системы: она включает в себя наименование типа и указатель на функцию, вызываемую во время монтирования этой файловой системы. При монтировании файловой системы вызывается соответствующая функция монтирования. Эта функция используется для считывания суперблока, установки внутренних переменных и возврата дескриптора смонтированной системы в VFS. После того, как система смонтирована, функции VFS используют этот дескриптор для доступа к процедурам используемой файловой системы.

Дескриптор смонтированной файловой системы содержит в себе некоторую информацию: информация, которая одинакова для каждого типа файловой системы, указатели на функции, используемые для выполнения операций данной файловой системы и некоторые данные, используемые этой системой. Указатели на функции, расположенные в дескрипторе файловой системы, позволяют VFS получить доступ к внутренним функциям файловой системы.

В VFS используются еще два типа дескрипторов: это inode и дескриптор открытого файла. Каждый из них содержит информацию, связанную с используемыми файлами и набором операций, используемых кодом файловой системы. В то время как дескриптор inode содержит указатели к функциям, используемым по отношению к любому файлу (например, create или unlink), то дескриптор файлов содержит указатели к функциям, оперирующим только с открытыми файлами (например, read или write).

Журналируемая файловая система

Для минимизации проблем связанных с целостностью и минимизации времени перезапуска системы, журналируемая файловая система хранит список изменений, которые она будут проводить с файловой системой перед фактической записью изменений. Эти записи хранятся в отдельной части файловой системы, называемой «журналом. Как только эти записи журнала безопасно записаны, журналируемая файловая система вносит эти изменения в файловую систему и затем удаляет эти записи из журнала регистраций.

Журналируемая файловая система увеличивает вероятность целостности, потому что записи журнал ведутся до проведения изменений файловой системы, и потому что файловая система хранит эти записи до тех пор, пока они не будут целиком и безопасно применены к файловой системе. При перезагрузке компьютера, который использует журналируемую файловую систему, программа монтирования может гарантировать целостность файловой системы простой проверкой журнала на наличие ожидаемых, но не произведенных изменений и записью их в файловую систему. В большинстве случаев, системе не нужно проводить проверку целостности файловой системы, а это означает, что компьютер использующий журналируемую файловую систему будет доступен для работы практически сразу после перезагрузки. Соответственно шансы потери данных в связи с проблемами в файловой системе значительно снижаются.

Примеры:

1) XFS, журналируемая файловая система разработанная Silicon Graphics, но сейчас выпущенная открытым кодом (open source);

2) RaiserFS, журналируемая файловая система разработанная специально для Linux;

3) JFS, журналируемая файловая система первоначально разработанная IBM, но сейчас выпущенная как открытый код;

4) ext3 – файловая система разработанная доктором Стефаном Твиди (Stephan Tweedie) в Red Hat, и несколько других систем.

Буфер сверхоперативной памяти (кеш)

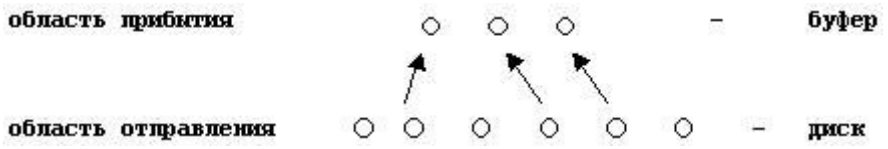
Определение

Ядро старается свести к минимуму частоту обращений к диску, заведя специальную область внутренних информационных буферов, именуемую буферным кешем. Буферный кеш представляет собой программную структуру, которую не следует путать с аппаратными кешами, ускоряющими косвенную адресацию памяти.

Назначение

Буфер хранит содержимое блоков диска, к которым перед этим производились обращения. Перед чтением информации с диска ядро пытается считать что-нибудь из буфера кеша. Если в этом буфере отсутствует информация, ядро читает данные с диска и заносит их в буфер. Аналогично, информация, записываемая на диск, заносится в буфер для того, чтобы находиться там, если ядро позднее попытается считать ее.

Каждому блоку соответствует один буфер.



Свойства:

- функциональность

- инъективность

- не всюду определено

- сюръективный

Каждый буфер состоит из двух частей: области памяти, в которой хранится информация, считываемая с диска, и заголовка буфера, который идентифицирует буфер.

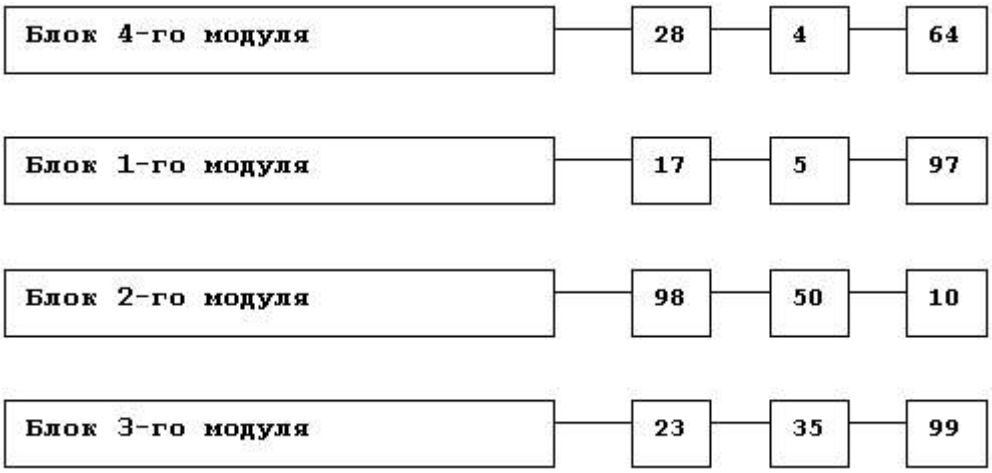
Состояние буфера представляет собой комбинацию из следующих условий:

- буфер заблокирован (термины "заблокирован (недоступен)" и "занят" равнозначны, так же, как и понятия "свободен" и "доступен"),
- буфер содержит правильную информацию,
- ядро должно переписать содержимое буфера на диск перед тем, как переназначить буфер; это условие известно, как "задержка, вызванная записью",
- ядро читает или записывает содержимое буфера на диск,
- процесс ждет освобождения буфера.

Заголовок буфера:

№ устройства
№ блока, который в текущий момент времени загружен в буфере
указатель на предыдущий блок в ХЭШ-очереди
указатель на следующий блок в ХЭШ-очереди
указатель на следующий буфер в списке свободных
указатель на предыдущий буфер в списке свободных
состояние буфера

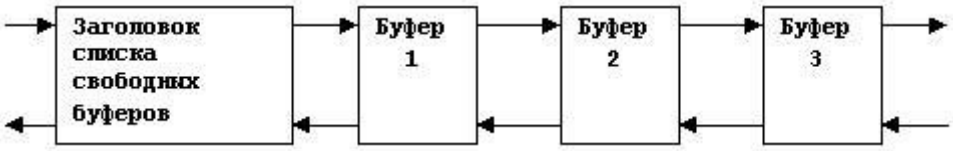
Структура области буферов:



Буферы организуются в ХЭШ очереди.

В зависимости от номера блока генерируется ХЭШ ключ, и буфер помещается в соответствующую этому ключу очередь.

Буфер, с которым не производится чтение и запись или операции дискового ввода вывода помещается в список свободных буферов.



ХЭШ- функция ≈ это остаток от деления на 4.

Буферы, с которыми не производятся чтение/ запись процессом или операции дискового ввода/ вывода помещаются в список свободных буферов.

Если нам нужен буфер, то он выделяется из начала списка свободных буферов. В него заносится нужная информация с диска, и буфер помещается в конец этого списка.

Любой буфер всегда находится в хеш-очереди, но его положение в очереди не имеет значения. Ядро просматривает хеш-очередь, если ему нужно найти определенный буфер, и выбирает буфер из списка свободных буферов, если ему нужен любой свободный буфер. Еще раз напомним: буфер всегда находится в хеш-очереди, а в списке свободных буферов может быть, но может и отсутствовать.

Механизм поиска буфера

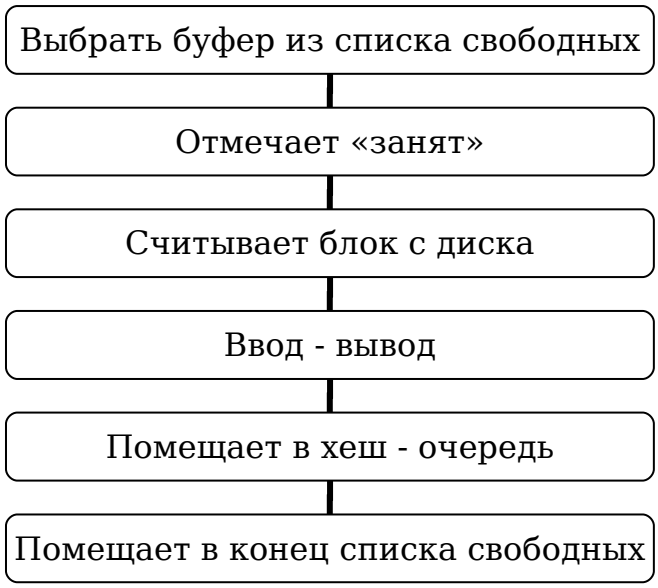
Рассмотрим пять возможных случаев:.

1. Ядро обнаруживает блок в хеш-очереди, соответствующий ему буфер свободен.
2. Ядро не может обнаружить блок в хеш-очереди, поэтому оно выделяет буфер из списка свободных буферов.
3. Ядро не может обнаружить блок в хеш-очереди и, пытаясь выделить буфер из списка свободных буферов (как в случае 2), обнаруживает в списке буфер, который помечен как "занят на время записи". Ядро должно переписать этот буфер на диск и выделить другой буфер.
4. Ядро не может обнаружить блок в хеш-очереди, а список свободных буферов пуст.
5. Ядро обнаруживает блок в хеш-очереди, но его буфер в настоящий момент занят.

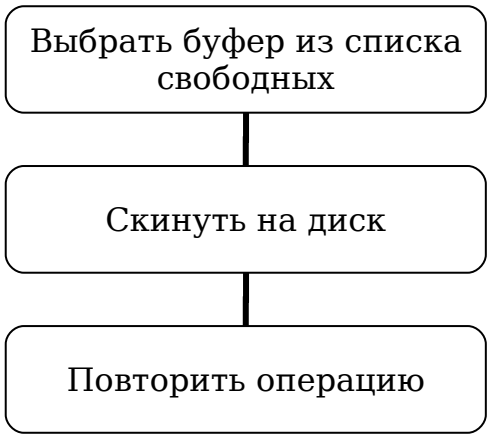
Рассмотрим каждый случай более подробно.

1. Ядро обнаруживает блок в хеш-очереди, соответствующий ему буфер свободен.

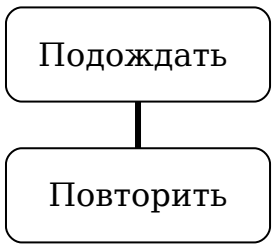
2. Ядро не может обнаружить блок в хеш-очереди, поэтому оно выделяет буфер из списка свободных буферов.



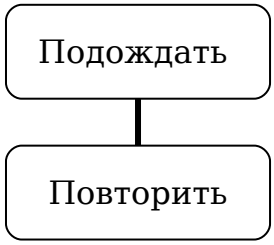
3. Ядро не может обнаружить блок в хеш-очереди и, пытаясь выделить буфер из списка свободных буферов (как в случае 2), обнаруживает в списке буфер, который помечен как "занят на время записи". Ядро должно переписать этот буфер на диск и выделить другой буфер.



4. Ядро не может обнаружить блок в хеш-очереди, а список свободных буферов пуст.



6. Ядро обнаруживает блок в хеш-очереди, но его буфер в настоящий момент занят.



Преимущества и неудобства буферного кеша

- Использование буферов позволяет внести единообразие в процедуру обращения к диску, поскольку ядру нет необходимости знать причину ввода-вывода. Вместо этого, ядро копирует данные в буфер и из буфера, невзирая на то, являются ли данные частью файла, индекса или суперблока.
- Система не накладывает никаких ограничений на выравнивание информации пользовательскими процессами, выполняющими ввод-вывод.
- Благодаря использованию буферного кеша, сокращается объем дискового трафика и время реакции и повышается общая производительность системы.
- Алгоритмы буферизации помогают поддерживать целостность файловой системы, так как они сохраняют общий, первоначальный и единственный образ дисковых блоков,

содержащихся в кеше. Если два процесса одновременно попытаются обратиться к одному и тому же дисковому блоку, алгоритмы буферизации параллельный доступ преобразуют в последовательный, предотвращая разрушение данных.

- Сокращение дискового трафика является важным преимуществом с точки зрения обеспечения хорошей производительности или быстрой реакции системы, однако стратегия кеширования также имеет некоторые неудобства. Так как ядро в случае отложенной записи не переписывает данные на диск немедленно, такая система уязвима для сбоев, которые оставляют дисковые данные в некорректном виде.
- Использование буферного кеша требует дополнительного копирования информации при ее считывании и записи пользовательскими процессами. Процесс, записывающий данные, передает их ядру и ядро копирует данные на диск; процесс, считывающий данные, получает их от ядра, которое читает данные с диска. При передаче большого количества данных дополнительное копирование отрицательным образом отражается на производительности системы.

Состояние процессов

Предположим, что программа была заранее собрана в некий единый самодостаточный объект, называемый *загрузочным модулем*. В ряде ОС программа собирается в момент загрузки из большого числа отдельных модулей. *Программа* – та часть загрузочного модуля, которая содержит исполняемый код. Результат загрузки программы в память называется *образом процесса или процесс*. К образу процесса относят не только код и данные процесса, но и системные структуры данных, связанные с этим процессом. В старой литературе процесс часто называют *задачей*.

В системах с виртуальной памятью каждому процессу обычно выделяется свое адресное пространство, поэтому можно употреблять термин *процесс* и в этом смысле. Во многих системах значительная часть адресных пространств перекрываются – это используется для реализации разделяемого кода и данных.

В рамках одного процесса может выполняться один или несколько *потоков* или *нитей*.

Некоторые системы представляют и более крупные структурные единицы, чем процесс. Например, в Unix существуют группы процессов, которые используются для реализации логического объединения процессов в *задания*.

Создание процессов в Unix

Процесс создается системным вызовом fork. Этот вызов создает два процесса, образы которых в первый момент идентичны, у них различается только возвращаемое значение.

При этом каждый из процессов имеет свою копию всех локальных и статических переменных. На процессорах со страничным диспетчером памяти физического копирования не происходит. Изначально оба процесса используют одни и те же страницы памяти, а дублируются только те из них, которые были изменены.

Если мы хотим запустить другую программу, то мы должны исполнить системный вызов из семейства exel. Вызовы этого семейства различаются только способом передачи параметров. Все они прекращают исполнение текущего образа процесса и создают новый процесс с новым виртуальным адресным пространством, но с тем же идентификатором процесса. При этом у нового процесса будет тот же приоритет, будут открыты те же файлы и т. д.

Процесс - это последовательность операций при выполнении программы, которая представляет собой наборы байтов интерпретируемые ЦП, как машинные инструкции. Он состоит из текста, данных и стека.
Под текстом понимаются код, машинные инструкции.

Время жизни процесса можно теоретически разбить на несколько состояний, описывающих процесс. Полный набор состояний процесса содержится в следующем перечне:

1. Процесс выполняется в режиме задачи.
2. Процесс выполняется в режиме ядра.
3. Процесс не выполняется, но готов к запуску под управлением ядра.
4. Процесс приостановлен и находится в оперативной памяти.
5. Процесс готов к запуску, но программа подкачки (нулевой процесс) должна еще загрузить процесс в оперативную память, прежде чем он будет запущен под управлением ядра.
6. Процесс приостановлен и программа подкачки выгрузила его во внешнюю память, чтобы в оперативной памяти освободить место для других процессов.
7. Процесс возвращен из привилегированного режима (режима ядра) в непривилегированный (режим задачи), ядро резервирует его и переключает контекст на другой процесс. Об отличии этого состояния от состояния 3 (готовность к запуску) пойдет речь ниже.
8. Процесс вновь создан и находится в переходном состоянии; процесс существует, но не готов к выполнению, хотя и не приостановлен. Это состояние является начальным состоянием всех процессов, кроме нулевого.
9. Процесс вызывает системную функцию exit и прекращает существование. Однако, после него осталась запись, содержащая код выхода, и некоторая хронометрическая статистика, собираемая родительским процессом. Это состояние является последним состоянием процесса.

Рисунок представляет собой полную диаграмму переходов процесса из состояния в состояние. Рассмотрим с помощью модели переходов типичное поведение процесса. Ситуации, которые будут обсуждаться, несколько искусственны и процессы не всегда имеют дело с ними, но эти ситуации вполне применимы для иллюстрации различных переходов. Начальным состоянием модели является создание процесса родительским процессом с помощью системной функции fork; из этого состояния процесс неминуемо переходит в состояние готовности к

запуску (3 или 5). Для простоты предположим, что процесс перешел в состояние "готовности к запуску в памяти" (3). Планировщик процессов в конечном счете выберет процесс для выполнения и процесс перейдет в состояние "выполнения в режиме ядра", где доиграет до конца роль, отведенную ему функцией fork.

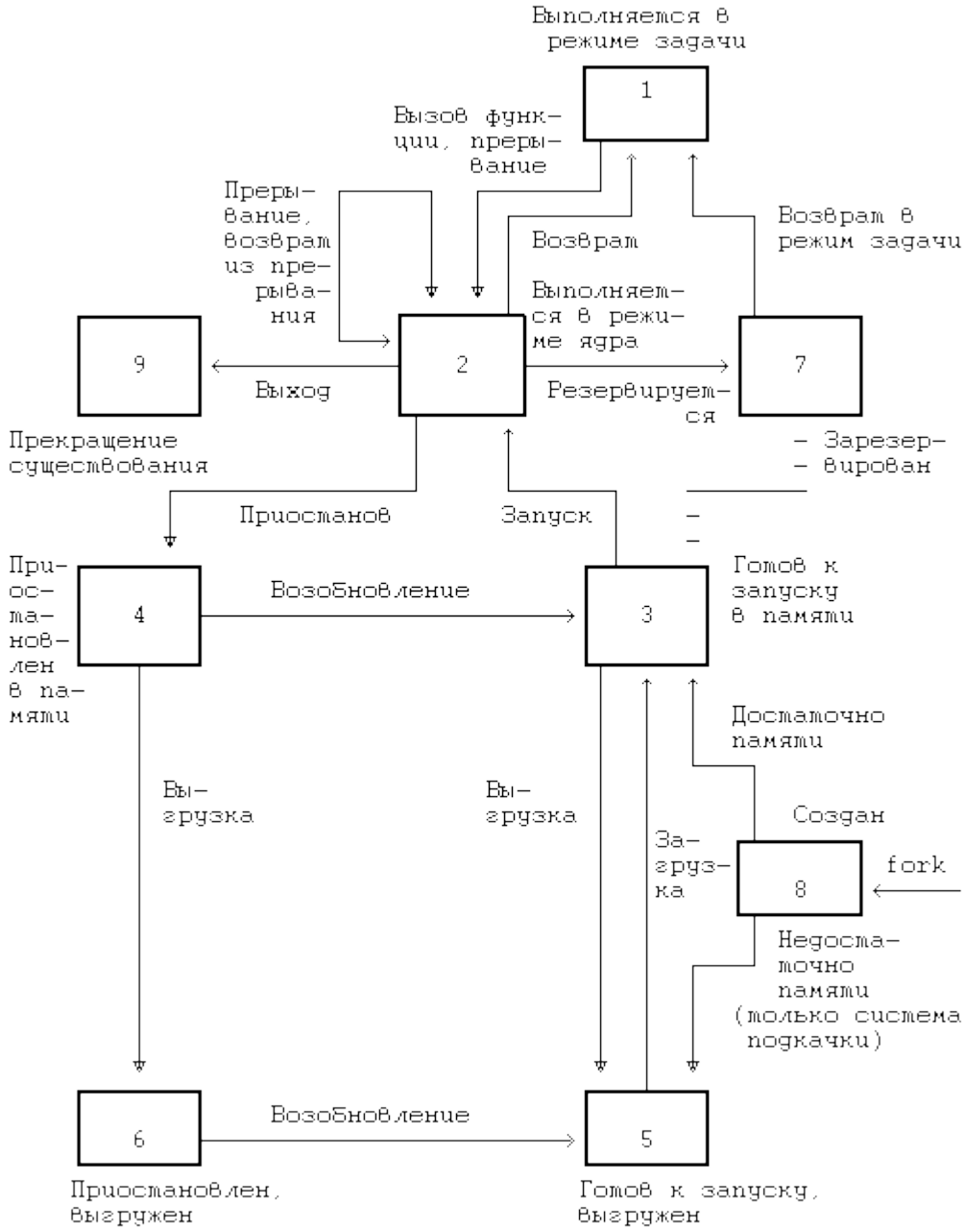


Диаграмма переходов процесса из состояния в состояние

После всего этого процесс может перейти в состояние "выполнения в режиме задачи". По прохождении определенного периода времени может произойти прерывание работы процессора по таймеру и процесс снова перейдет в состояние "выполнения в режиме ядра". Как только программа обработки прерывания закончит работу, ядру может понадобиться подготовить к запуску другой процесс, поэтому первый процесс перейдет в состояние "резервирования", уступив дорогу второму процессу. Состояние "резервирования" в действительности не отличается от состояния "готовности к запуску в памяти" (пунктирная

линия на рисунке, соединяющая между собой оба состояния, подчеркивает их эквивалентность), но они выделяются в отдельные состояния, чтобы подчеркнуть, что процесс, выполняющийся в режиме ядра, может быть зарезервирован только в том случае, если он собирается вернуться в режим задачи. Следовательно, ядро может при необходимости подкачивать процесс из состояния "резервирования". При известных условиях планировщик выберет процесс для исполнения и тот снова вернется в состояние "выполнения в режиме задачи".

Когда процесс выполняет вызов системной функции, он из состояния "выполнения в режиме задачи" переходит в состояние "выполнения в режиме ядра". Предположим, что системной функции требуется ввод-вывод с диска и поэтому процесс вынужден дожидаться завершения ввода-вывода. Он переходит в состояние "приостановка в памяти", в котором будет находиться до тех пор, пока не получит извещения об окончании ввода-вывода. Когда ввод-вывод завершится, произойдет аппаратное прерывание работы центрального процессора и программа обработки прерывания возобновит выполнение процесса, в результате чего он перейдет в состояние "готовности к запуску в памяти".

Предположим, что система выполняет множество процессов, которые одновременно никак не могут поместиться в оперативной памяти, и программа подкачки (нулевой процесс) выгружает один процесс, чтобы освободить место для другого процесса, находящегося в состоянии "готов к запуску, но выгружен". Первый процесс, выгруженный из оперативной памяти, переходит в то же состояние. Когда программа подкачки выбирает наиболее подходящий процесс для загрузки в оперативную память, этот процесс переходит в состояние "готовности к запуску в памяти". Планировщик выбирает процесс для исполнения и он переходит в состояние "выполнения в режиме ядра". Когда процесс завершается, он исполняет системную функцию exit, последовательно переходя в состояния "выполнения в режиме ядра" и, наконец, в состояние "прекращения существования".

Процесс может управлять некоторыми из переходов на уровне задачи:

1) Во-первых, один процесс может создать другой процесс. Тем не менее, в какое из состояний процесс перейдет после создания (т.е. в состояние "готов к выполнению, находясь в памяти" или в состояние "готов к выполнению, но выгружен") зависит уже от ядра. Процессу эти состояния не подконтрольны.

2) Во-вторых, процесс может обратиться к различным системным функциям, чтобы перейти из состояния "выполнения в режиме задачи" в состояние "выполнения в режиме ядра", а также перейти в режим ядра по своей собственной воле. Тем не менее, момент возвращения из режима ядра от процесса уже не зависит; в результате каких-то событий он может никогда не вернуться из этого режима и из него перейдет в состояние "прекращения существования".

3) Наконец, процесс может завершиться с помощью функции exit по своей собственной воле, но как указывалось ранее, внешние события могут потребовать завершения процесса без явного обращения к функции exit. Все остальные переходы относятся к жестко закрепленной части модели, закодированной в ядре, и являются результатом определенных событий, реагируя на них в соответствии с правилами, сформулированными в этой и последующих главах. Некоторые из правил уже упоминались: например, то, что процесс может выгрузить другой процесс, выполняющийся в ядре.

Две принадлежащие ядру структуры данных описывают процесс: запись в таблице процессов и пространство процесса. Таблица процессов содержит поля, которые должны быть всегда доступны ядру, а пространство процесса - поля, необходимость в которых возникает только у выполняющегося процесса. Поэтому ядро выделяет место для пространства процесса только при создании процесса: в нем нет необходимости, если записи в таблице процессов не соответствует конкретный процесс.

Таблица процессов состоит из следующих полей:

- Поле состояния, которое идентифицирует состояние процесса.
- Несколько пользовательских идентификаторов (UID), устанавливающих различные привилегии процесса.
- Идентификаторы процесса (PID), указывающие взаимосвязь между процессами. Значения полей PID задаются при переходе процесса в состояние "создан" во время выполнения функции fork.
- Дескриптор события (устанавливается тогда, когда процесс приостановлен).
- Параметры планирования, позволяющие ядру устанавливать порядок перехода процессов из состояния "выполнения в режиме ядра" в состояние "выполнения в режиме задачи".
- Поле сигналов, в котором перечисляются сигналы, посланные процессу, но еще не обработанные.
- Различные таймеры, описывающие время выполнения процесса и использование ресурсов ядра и позволяющие осуществлять слежение за выполнением и вычислять приоритет планирования процесса.

Поля пространства процесса:

- Указатель на таблицу процессов, который идентифицирует запись, соответствующую процессу.
- Пользовательские идентификаторы, устанавливающие различные привилегии процесса, в частности, права доступа к файлу.
- Поля таймеров, хранящие время выполнения процесса (и его потомков) в режиме задачи и в режиме ядра.
- Вектор, описывающий реакцию процесса на сигналы.
- Поле операторского терминала, идентифицирующее "регистрационный терминал", который связан с процессом.

- Поле ошибок, в которое записываются ошибки, имевшие место при выполнении системной функции.
- Поле возвращенного значения, хранящее результат выполнения системной функции.
- Параметры ввода-вывода: объем передаваемых данных, адрес источника (или приемника) данных в пространстве задачи, смещения в файле (которыми пользуются операции ввода-вывода) и т.д.
- Имена текущего каталога и текущего корня, описывающие файловую систему, в которой выполняется процесс.
- Таблица пользовательских дескрипторов файла, которая описывает файлы, открытые процессом.
- Поля границ, накладывающие ограничения на размерные характеристики процесса и на размер файла, в который процесс может вести запись.
- Поле прав доступа, хранящее двоичную маску установок прав доступа к файлам, которые создаются процессом.

Fork

Единственная возможность породить процесс. Создает копию. Не наследует PID.

Алгоритм:

1) Проверяет доступность ресурсов ядра

2) Получает место в таблице

3) Заполняет таблицу. Копирование записи родителя.

4) Увеличивает счетчик ссылок файла

5) Копирует контекст.

если *выполняется родитель*

перевести порожд. процесс в (3), (5)

возвратить PID порожд. процесса — *родитель*

иначе

инициализация

возвратить 0; — *потомок*

Exec

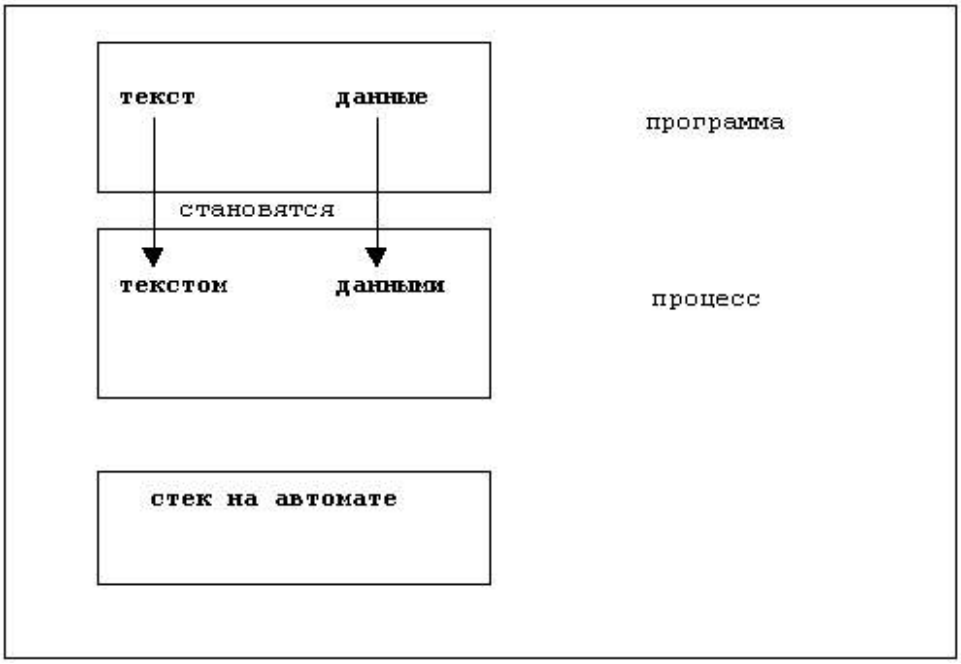
1) загружает текст

2) загружает инициализ. данные

3) выделяет память под неинициализ. данные

4) создает стек

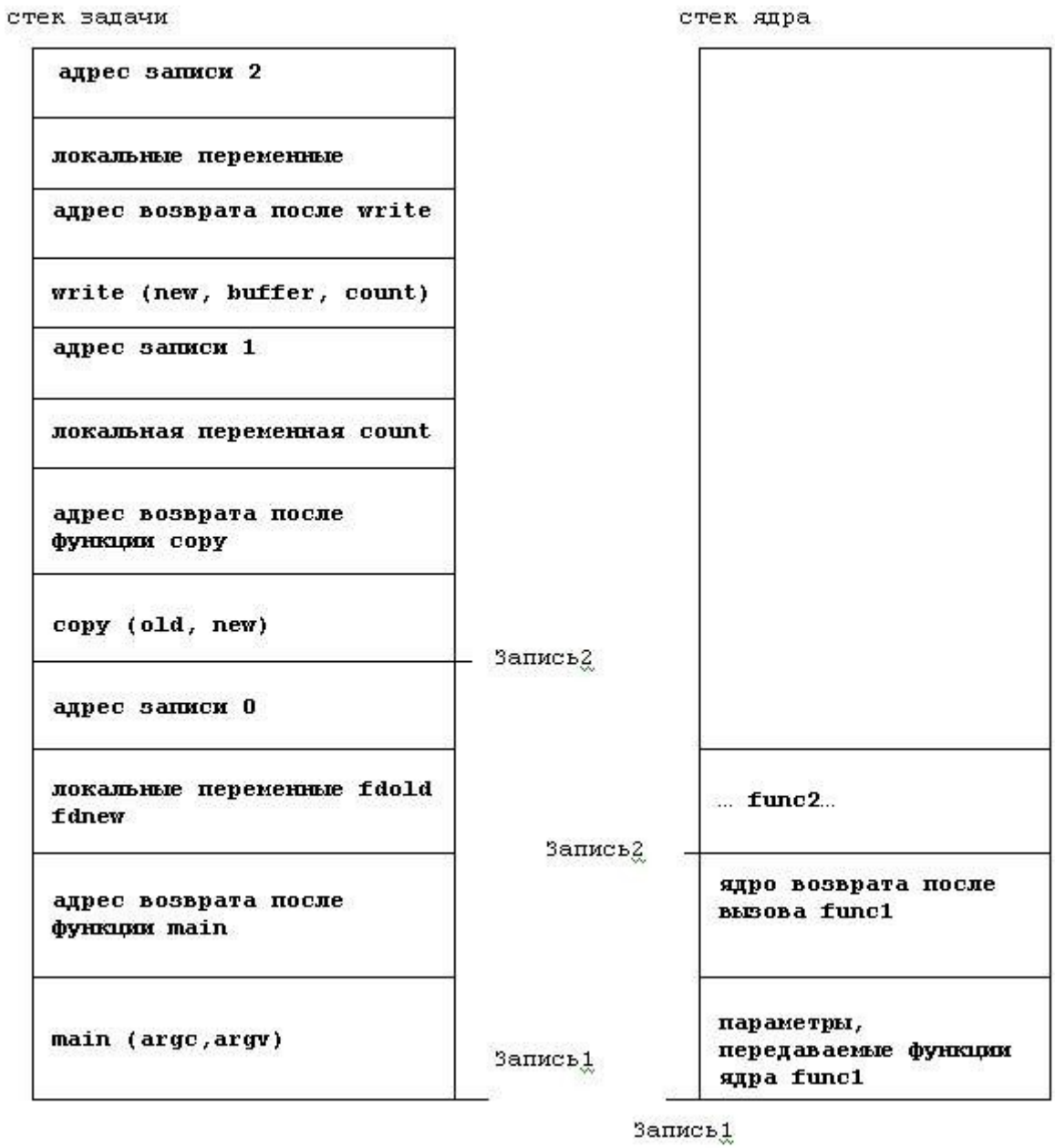
Когда программа лежит на диске в ней содержатся:



Стеки:

· задачи

· ядра



Каждая запись состоит из параметров функции, ее локальных переменных и данных для восстановления последней активации.

Пример:

```
main (arg c, arg v)
    int arg c;
    char *arg v[];
{ if (fork() == 0)
    execl (“copy”, “copy”, arg v[1], arg v[2], 0);
    wait ((int *)0);
```

printf (“copydone/n”);

}

Exit

Системная функция, завершает работу процесса

Int exit (int c);

Ничего не возвращает, т. к. при завершении процесса функция продолжает работу.

Состояние процесса «зомби», если родитель завершает существование, а порожденный продолжает.

1) если процесс возглавляет группу, связанную с терминалом. Посылается сообщение всей группе – завершение.

2) закрыть все открытые файлы, освободить текущий каталог

3) освободить область памяти, связанную с процессом

4) перевести процесс в состояние завершения

5) назначить всем потомкам в качестве родителя – класс Unit, если кто-то из потомков перестал существовать init посылается сообщение о гибели потомка.

6) переключение контекста

Формат памяти системы

Адресное пространство процесса состоит из 3 сегментов:

- текстового сегмента (команды);
- сегмент данных;
- сегмент стека.

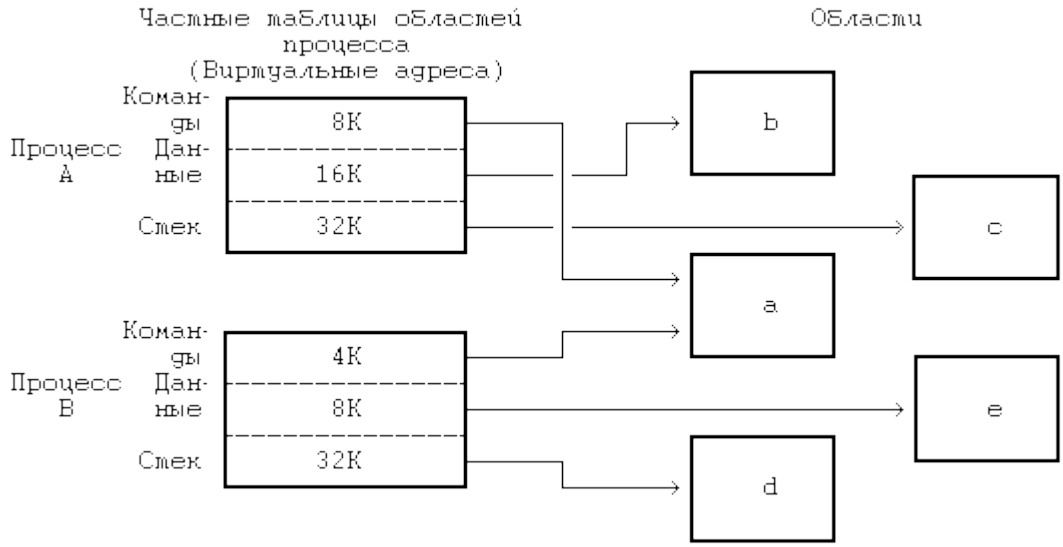
Если адреса в сгенерированном коде трактовать как адреса в физической памяти, два процесса не смогут параллельно выполняться, если их адреса перекрываются.

Поэтому компилятор генерирует адреса для виртуального адресного пространства заданного диапазона, а устройство управления памятью, называемое диспетчером памяти, транслирует виртуальные адреса, сгенерированные компилятором, в адреса ячеек, расположенных в физической памяти. Компилятору нет необходимости знать, в какое место в памяти ядро потом загрузит выполняемую программу.

Виртуальная адресация

Ядро в версии V делит виртуальное адресное пространство процесса на совокупность логических областей. Область - это непрерывная зона виртуального адресного пространства процесса, рассматриваемая в качестве отдельного объекта для совместного использования и защиты. Таким образом, команды, данные и стек обычно образуют автономные области, принадлежащие процессу

На рисунке изображены два процесса, A и B, показаны их области, частные таблицы областей и виртуальные адреса, в которых эти области соединяются. Процессы разделяют область команд 'a' с виртуальными адресами 8K и 4K соответственно. Если процесс A читает ячейку памяти с адресом 8K, а процесс B читает ячейку с адресом 4K, то они читают одну и ту же ячейку в области 'a'. Область данных и область стека у каждого процесса свои.



Процессы и области

Физическая адресация

Центральный Процессор при выполнении процесса переводит виртуальные адреса, используемые в нем в физические, используя таблицы страниц и областей.

Для удобства подобных трансляций память делится на участки фиксированного раздела (страницы) или варьирующегося (сегменты). В дальнейшем будем рассматривать страничную организацию памяти.

Таким образом, адрес делится на две составляющие - номер страницы и смещение относительно ее начала.

Для простоты взяв размер страницы 1 Кб. и 32 битную архитектуру получим:

1) возможный объем виртуальной памяти 2^{32} b = 2^{22} Kb = 2^{12} Mb = 4 Gb,

2) диапазон смещения 1 Kb = 2^{10} b,

3) число страниц 2^{22} шт.

Для того чтобы разделить адрес на номер страницы и смещение надо "отрезать" от этого адреса нижние 10 бит - это будет смещение, то, что останется - номер страницы. Или, что то же самое, поделить нацело на 1024 (2^{10}). Остаток будет смещением, а частное - номером страницы.

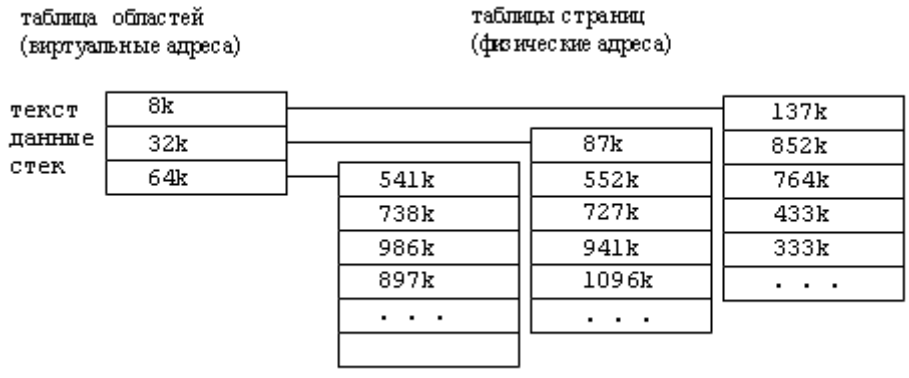
Рассмотрим для примера адрес 0x58432. Мы взяли число в шестнадцатиричном коде, т. к. он удобен при переводе в двоичный - каждая цифра переводится в четыре двоичных.

HEX	5	8	4	3	2
BIN	0101	1000	0100	0011	0010
BIN	01	0110	0001	00	0011 0010
HEX	1	6	1	3	2

Номер страницы получился 0x161, а смещение 0x32

Каждая из составляющих процесса (текст, данные, стек) содержится в отдельной области - непрерывного пространства виртуальных адресов. Рассмотрим, как центральный процессор производит перевод виртуального адреса в физический, используя таблицы страниц и областей.

Возьмем некоторый процесс, виртуальные адреса которого начинаются с 8k (8*1024 = 8192), а его физическое расположение фрагментировано и номера физических страниц указаны в таблице страниц.

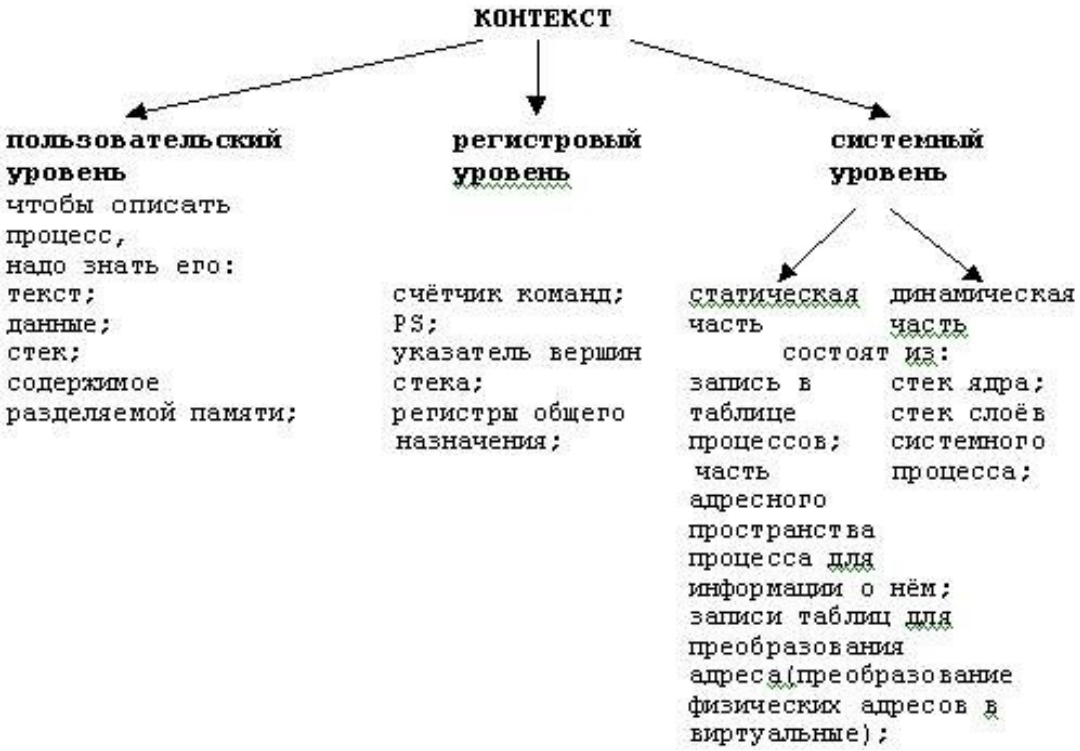


По этому рисунку: обращение к виртуальному адресу 68432 (в десятичном коде на этот раз). Этот адрес, как видно (> 64 * 1024), относится к области стека. Вычислим смещение от начала стека: 68432-64*1024=68432-65536=2896. Определим номер страницы в стеке: 2896 div 1024 = 2. По рисунку видно, что второй виртуальной странице стека соответствует физическая страница по адресу 986k (считаем от 0).

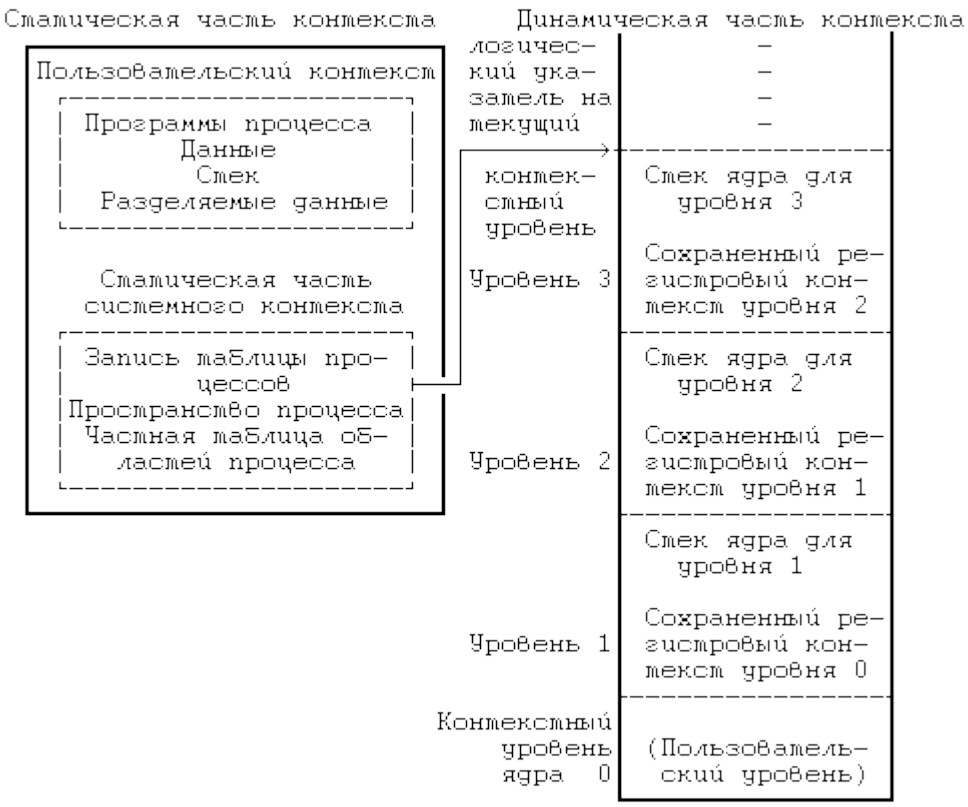
Смещение относительно начала страницы 2896 mod 1024 = 848. Таким образом, мы получили, что виртуальному адресу 68432 соответствует физический 986*1024+848.

Все эти действия выполняет центральный процессор, а задача операционной системы только составить эти таблицы и сообщить процессору, где они находятся.

Контекст процесса. Уровни и слои контекста



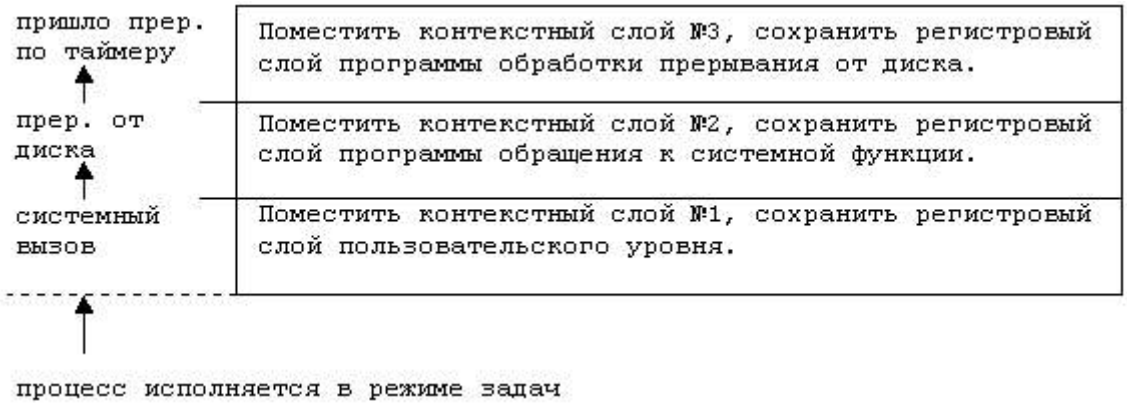
Компоненты контекста процесса



Слои контекста

Ядро помещает контекстный слой, когда возникает прерывание или программа делает системный вызов.

Ядро выталкивает контекстный слой, когда происходит завершение обработки прерывания либо возврат в режим задачи, т.е. в состояние 1.



Переключение контекста

· процесс переходит в состояние сна;

· процесс делает системный вызов exit и завершает работу;

· процесс переходит в режим задачи после обработки прерывания;

· процесс переходит в режим задачи после системного вызова.

Алгоритм переключения контекста:

1. принять решение о необходимости и допустимости переключения контекста;

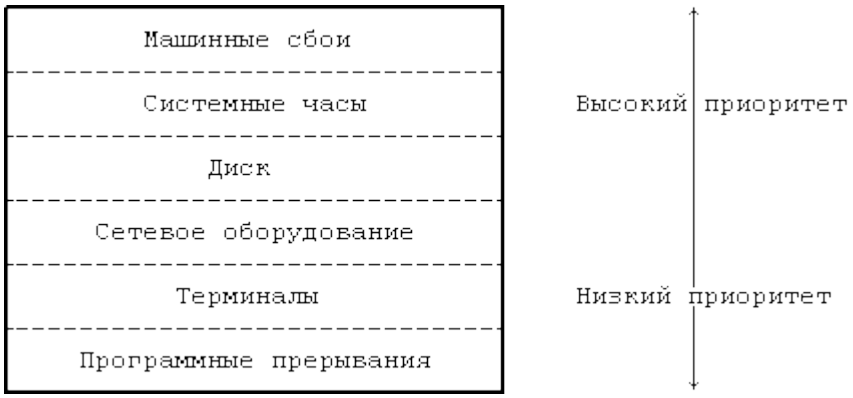
2. сохранить контекст;

3. выбрать подходящий процесс;

4. восстановить его контекст.

Прерывания

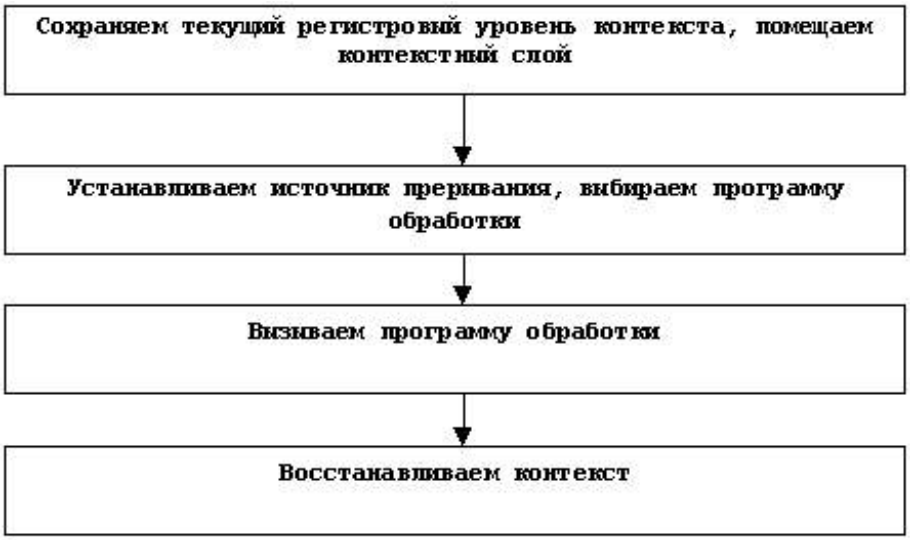
6 уровней прерывания:



По получении сигнала прерывания ядро операционной системы сохраняет свой текущий контекст (застывший образ выполняемого процесса), устанавливает причину прерывания и обрабатывает прерывание. После того, как прерывание будет обработано ядром, прерванный контекст восстановится и работа продолжится так, как будто ничего не случилось.

Устройствам обычно приписываются приоритеты в соответствии с очередностью обработки прерываний. В процессе обработки прерываний ядро учитывает их приоритеты и блокирует обслуживание прерывания с низким приоритетом на время обработки прерывания с более высоким приоритетом.

Обработка прерываний:



Системный вызов

Системный вызов можно рассматривать как внутреннее прерывание операционной системы.

Алгоритм системного вызова:

1. найти запись в таблице системных функций;

2. определить число параметров и скопировать их из адресного пространства задачи;

3. сохранение контекста;

4. запуск системного вызова;

5. номер ошибки записывается в регистр общего назначения;

6. возврат

Планирование процессов

Алгоритмы планирования

1) First Come, First Served (первым пришел, первым обслужен)

Очередь FIFO (First In, First Out (первым вошел, первым вышел)).

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

Недостатки:

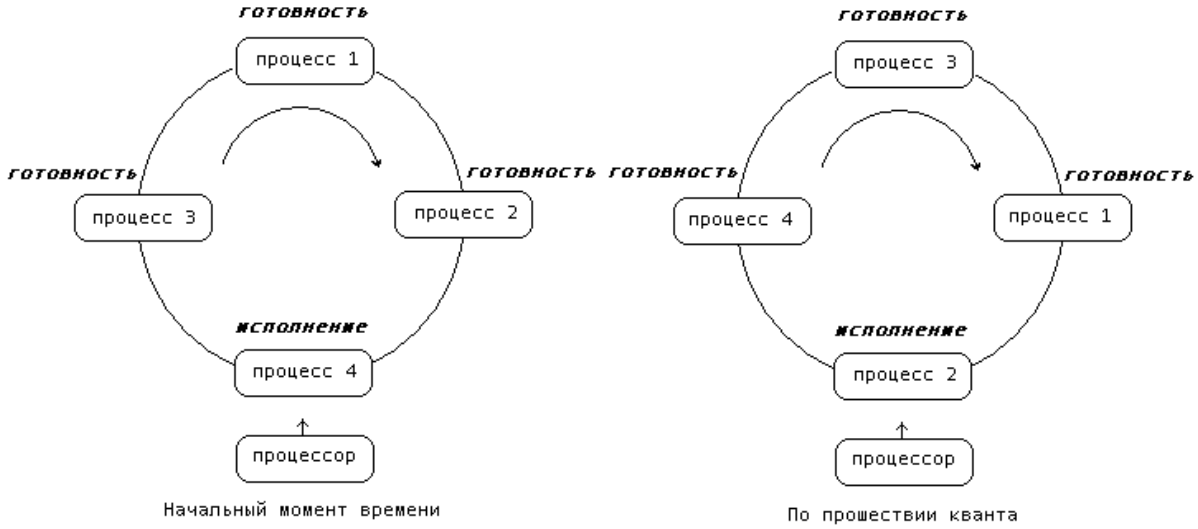
- если процесс с длительным CPU burst, то короткие процессы, перешедшие в состояние «готовность» после длительного процесса, будут очень долго ждать начала своего выполнения

- алгоритм FCFS практически неприменим для систем разделения времени

- слишком большое среднее время отклика в интерактивных процессах

2) Round Robin (RR)

(Round Robin – это вид детской карусели в США)



При выполнении процесса возможны два варианта:

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

Недостатки:

- при слишком малой величине кванта времени и, соответственно, слишком частом переключении контекста, накладные расходы на переключение резко снижают производительность системы

3) Shortest-Job-First (SJF)

(“кратчайшая работа первой” или *Shortest Job First* (*SJF*))

Квантование времени не применяется.

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

4) Приоритетное планирование

При приоритетном планировании каждому процессу присваивается определенное числовое значение — приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJF в качестве такого приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс.

5) Многоуровневые очереди

Для систем, в которых процессы могут быть легко рассортированы на разные группы, был разработан другой класс алгоритмов планирования. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии готовность (см. рисунок). Этим очередям приписываются фиксированные приоритеты. Приоритет очереди процессов, запущенных студентами, — ниже, чем для очереди процессов, запущенных преподавателями. Это значит, что ни один пользовательский процесс не будет выбран для исполнения, пока есть хоть один готовый системный процесс, и ни один студенческий процесс не получит в свое распоряжение процессор, если есть процессы преподавателей, готовые к исполнению. Внутри этих очередей для планирования могут применяться самые разные алгоритмы. Так, например, для больших счетных процессов может использоваться алгоритм FCFS, а для интерактивных процессов – алгоритм RR. Подобный подход, получивший название многоуровневых очередей, повышает гибкость планирования: для процессов с различными характеристиками применяется наиболее подходящий им алгоритм.

Диспетчеризация процессов

Ядро предоставляет процессу ресурсы центрального процессора на интервал времени квант, по истечению которого выгружает этот процесс и запускает другой, а также периодически меняет порядок в очереди процессов.

Ядро пересчитывает приоритеты и переключает контекст на процесс с наивысшим приоритетом.

Время в системе поддерживается с помощью аппаратного таймера, который посылает центральному процессору прерывание с фиксированной частотой.

Планирование и выполнение процессов. Действия планировщика.

1. Выбирается процесс с наивысшим приоритетом из находящихся в состояниях резервирования и готовность к выполнению.

2. Если таких процессов несколько, то выбирается тот, который дольше всех находится в очереди.

3. Если таких процессов нет, то ждем следующего прерывания по таймеру (тика).

4. Удаляем выбранный из очереди готовых к выполнению.

5. Переключение на контекст выбранного процесса.

Работа в режиме реального времени. Таймер.

Когда процессы ждут прерывания, они находятся в режиме сна. Приходит прерывание => процессы переходят в режим готовности к запуску.

Режим реального времени подразумевает возможность обеспечения достаточной скорости при обработке внешних прерываний и выполнении отдельных процессов в темпе, соизмеримом с частотой возникновения вызывающих прерывания событий.

Системные операции, связанные со временем.

stime ≈ устанавливает системное время в секундах, начиная с семидесятого года;

time ≈ выдать время в секундах, начиная с семидесятого года;

times ≈ возвращает суммарное время выполнения процесса и всех его потомков.

alarm ≈ процесс посылает себе сигнал будильника;

(Пример будильника:)

```
main ( )

{

extern wakeup ( );

signal (SIGALARM, wakeup);

while (1);

{

alarm (5);

pause ( );

}

}
```

Опишем wakeup, обрабатывающий сигнал будильника:

```
wakeup ( )

{

printf (“Я проснулся”);

}
```

В цикле заряжаем будильник на пять секунд. Процесс приостанавливается. Через пять секунд приходит сигнал будильника SIGALARM, и мы вызываем wakeup.

Таймер.

Функции программы обработки прерываний по таймеру:

1. Перезапуск часов для выполнения, следующего тика.

2. Вызов на выполнение функций ядра, использующих встроенные часы.

3. Поддержка возможности выполнения процессов.

time prog1

Программа prog1 вызывается на выполнение, и после выполнения выдается, сколько секунд она выполнялась.

4. Сбор статистики о системе и протекающих в ней процессах.

5. Слежение за временем.

6. Посылка процессом сигналов будильника по запросу.

7. Периодическое возобновление процесса подкачки.

8. Управление диспетчеризацией процессов.

Управление памятью

Планирование процессов зависит от алгоритмов управления памятью.

Две стратегии управления памятью:

1) Своппинг (swap)

ОП и устройство выгрузки обмениваются процессами целиком

2) Подкачка по запросу (demand paging)

Память должна быть разбита по страницам

Функции подсистемы управления памятью:

1. Решает, какие процессы следует размещать в памяти.

2. Управляет участниками виртуального пространства.

3. Переписывает процессы во внешнюю память.

4. Помещает данные с устройства выгрузки в основную память.

Свопинг

Рассмотрим три основных функции свопинга:

1. Управление пространством в свопе (на устройстве выгрузки).

2. Выгрузка процессов.

3. Подкачка процессов.

Управление пространством в свопе.

Своп - это устройство блочного типа, чаще всего раздел диска.

Если в файловой системе используются суперблоки, то в свопе используется карта памяти устройства.

Карта состоит из строк, в которых содержится адрес распределяемого ресурса и количество единиц этого ресурса.

Адрес начала ресурса	Число единиц ресурса

Показывает, где и сколько места на устройстве выгрузки.

1	1000

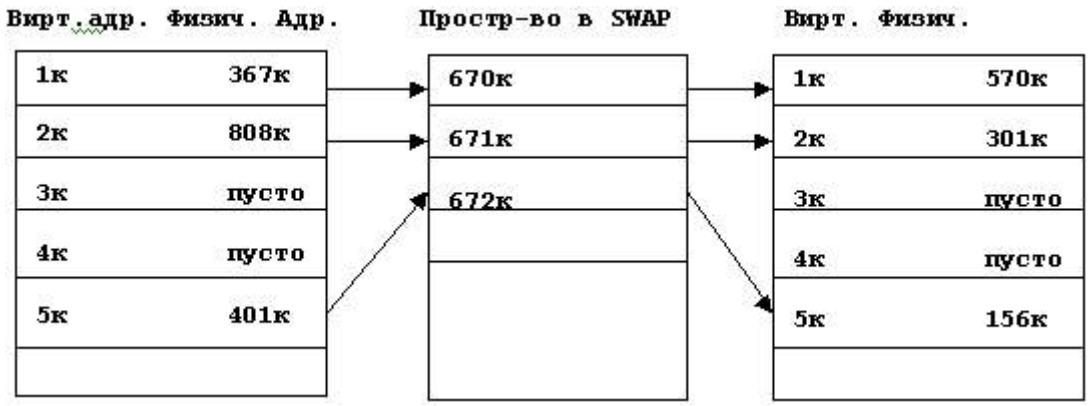
В начальный момент в карте памяти устройства одна строка

Выгрузка процесса в своп происходит, когда:

· возникает системный вызов fork.

· процесс увеличивает свои размеры

· процессу необходима физическая память.



Выгрузкой загрузкой процессов занимается процесс подкачки. Он всегда выполняется в режиме ядра и имеет номер 0.

Алгоритм выгрузки:

1. Процесс подкачки уменьшает счетчик числа ссылок на область на 1.

2. Выделяет место в SWAP.

3. Блокирует процессы в памяти.

4. Выгружает из областей, где счетчик числа ссылок равен 0, непустые страницы через КЭШ.

Загрузка

SWAPPER≈ название алгоритма.

1. Выбирают готовый к выполнению процесс, дольше всех лежащий в свопе (если нет подходящих, то делают приостановку).

2. Загружают его.

Выгрузка

1. Выбирают процесс, который дольше спит (или дольше всего в памяти)

2. Выгружают его

Подкачка по запросу (по обращению)

Demand paging

Особенности:

1) Основная память обменивается с внешней памятью не процессами, а страницами.

2) Этот способ должен иметь аппаратную поддержку: страничную организацию памяти и центральный процессор, имеющий прерываемые команды, должен быть бит упоминания страницы (reference bit), используемый для подсчета возраста страницы.

3) Отсутствуют ограничения на размер процесса, обусловленные объемом физической памяти.

Рабочее множество процесса - это совокупность страниц, использованных процессом в последних n-ссылках. Где n - окно рабочего множества процессов, число страниц, находящихся в ОП.

Последовательность указателей страниц	рабочее множество			
	n = 2	n = 3	n = 4	n = 5
24	24	24		
15	24,15	24,15		
18	15,18	24,15,18		
23	18,23	15,18,23		
24	23,24	18,23,24		
17	24,17	23,24,17		
18	17,18	24,17,18		
24	18,24	17,18,24		
18	24,18	18,24,18		
15	18,15	18,24,15		
число ошибок	9	8	6	5

Рисунок. Рабочее множество процесса

По мере выполнения процесса его рабочее множество видоизменяется в соответствии с используемыми процессом указателями страниц; увеличение размера окна влечет за собой увеличение рабочего множества и, с другой стороны, сокращение числа ошибок в выполнении процесса.

Для поддержки функций управления памятью на машинном (низком) уровне и для реализации механизма замещения страниц ядро использует 4 основные структуры данных: 1) записи таблицы страниц, 2) дескрипторы дисковых блоков, 3) таблицу содержимого страничных

блоков (pfdata) и 4) таблицу использования области подкачки. Место для таблицы pfdata выделяется один раз на все время жизни системы, для других же структур страницы памяти выделяются динамически.

Запись в таблице страниц

Блокир. КЭШа	Обращение reference bit	Изменение	Защита	Присутствие	Физический адрес

В **таблице pfdata** описывается каждая страница физической памяти. Записи таблицы проиндексированы по номеру страницы и состоят из следующих полей:

- Статус страницы
- Количество процессов, ссылающихся на страницу.
- Логический номер устройства (устройства выгрузки или файловой системы)
- Указатели на другие записи таблицы pfdata в соответствии со списком свободных страниц или с хеш-очередью страниц.

Выбирается страница, к которой дольше всего не было обращений. Откачку осуществляет «сборщик страниц». Просматриваются страницы, те, у которых не установлен reference bit (бит упоминания). Увеличивается возраст на 1. Если reference bit установлен, возраст обнуляется и reference bit сбрасывается. Reference bit устанавливается ЦП.

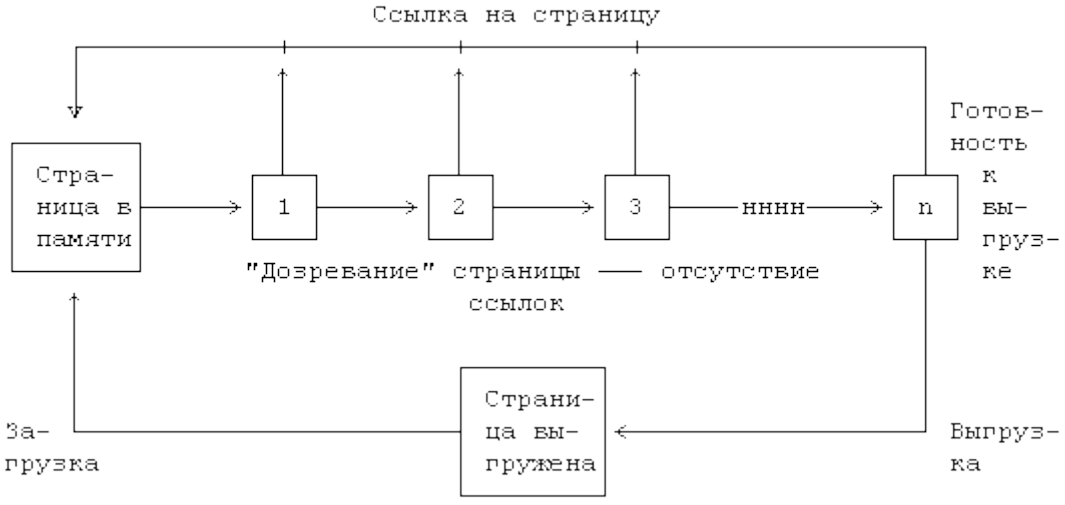


Диаграмма состояний страницы

Когда процесс обращается к странице, отсутствующей в его рабочем множестве, возникает ошибка, при обработке которой ядро корректирует рабочее множество процессов, а в случае необходимости подкачивает страницы с внешнего устройства.

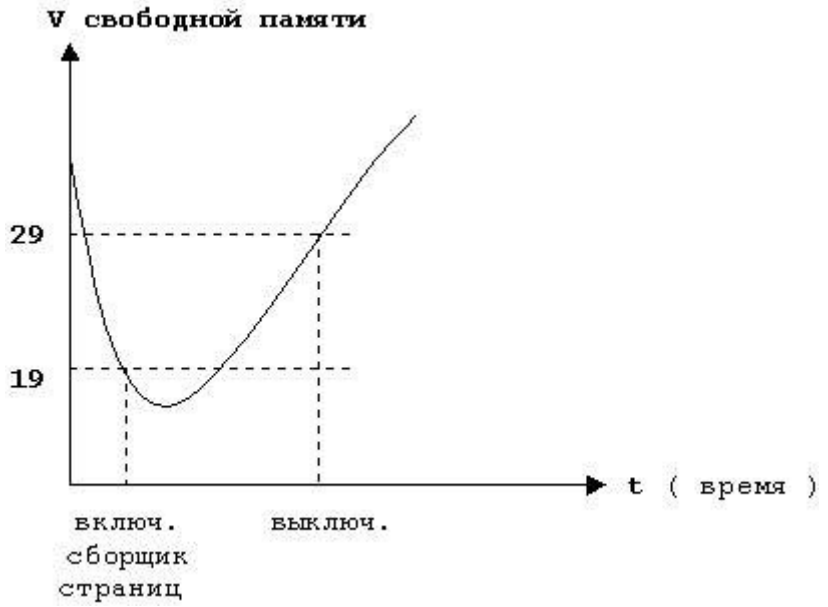
Ошибки - это обращения к несуществующим страницам.

Сборщик страниц - процесс, принадлежащий ядру и выполняющий выгрузку из памяти тех страниц, которые больше не входят в состав рабочего множества пользовательского процесса. Запускается в любой момент, когда в нем возникает необходимость. Он просматривает все активные незаблокированные области и увеличивает значение возраста принадлежащим им страницам.

Состояние	возраст	действие
в памяти	0	
в памяти	1	
в памяти	2	
	----------	обратился пользовательский процесс
в памяти	0	
в памяти	1	
	----------	обратился пользовательский процесс
в памяти	0	
в памяти	1	
в памяти	2	
в памяти	3	
вне памяти		выгрузка страницы

У страницы два состояния: либо она дозревает, либо она готова к выгрузке.

Ядро возобновляет работу сборщика страниц, когда доступная свободная память имеет размер недостающий до нижней допустимой отметки.



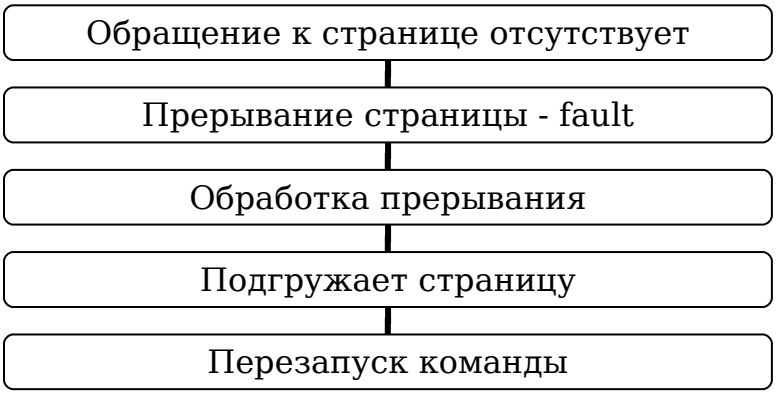
Обработка ошибок

Существует два вида ошибок при обращении к странице:

1. отсутствие данных;

2. отказ системы защиты.

1.



2. Обращение к недопустимым страницам. Например, к адресу, выходящему за рамки виртуального адреса пространства.

Когда процесс пытается обратиться к недоступной странице, возникает прерывание, и процессор запускает программу обработки прерываний по отказу данного типа. Если информация отсутствует в системе, то ядро посылает процессу сигнал нарушения сегментации.

Если ссылка на страницу сделана правильно, то ядро выделяет физическую страницу в памяти и считывает в нее содержимое виртуальной страницы с устройства выгрузки или из исполняемого файла.

Системы смешанного типа со свопингом и подкачкой по запросу. Несмотря на то, что подкачка по запросу отличается гибкостью, возможна ситуация, когда сборщик страниц не сможет достаточно быстро освободить место в памяти из-за того, что все страницы принадлежат рабочему множеству процессов. Выход из этого в том, чтобы комбинировать подкачку по запросу и свопинг.

Когда ядро не может выделить процессу страницы памяти, оно возобновляет работу процесса подкачки и переводит процесс в состояние эквивалентное готовности к запуску будучи зарезервированным.

Управление вводом/ выводом

Подсистема управления вводом/ выводом позволяет процессам поддерживать связь с периферийными устройствами.

Периферийные устройства - это терминалы, принтеры, сети.

Драйверы - модули ядра, которые управляют устройствами.

Каждому устройству соответствует один драйвер. Возможна ситуация, когда для однотипных устройств используются разные драйверы.

Стадии конфигурации драйвера:

1. при подключении модуля

2. для plug-and-play устройств

Bios устанавливает для каждого драйвера свое прерывание:

- при подгруке модуля

- в модуле ядра

Результатом конфигурации является заполнение таблицы ключей. В таблице можно выделить два поля:

тип устройства	адрес драйвера

При обращении к жесткому диску ядро смотрит тип устройства (оно содержится в имени файла).

Существует два вида устройств:

1. символьные - - информация считывается и записывается посимвольно (принтер, сетевые карты, мыши)

2. блочные - информация считывается и записывается по блокам, блоки имеют свой адрес (диски)

К символьным устройствам относят те, к которым возможен последовательный доступ (мышь, модем), к блочным – произвольный доступ (винчестеры, диски).

Соответственно бывают блочные и символьные файлы.

Обращение происходит через буферный Кеш.

/dev/ - специальный каталог, для обращения к устройствам

Для работы с блочными используются команды open, close, read, write.

А для работы с символьными ioctl (для вызова). Создаются файлы командой mknod:

mknod имя_файла тип ст. устр-ва мл. устр-ва

(Пример): для создания файла, который будет отвечать за COM1:

mknod <имя файла><тип> major minor, где

major – номер типа устройства

minor – номер устройства заданного типа

Например, mknod /dev/tty/ S0 C 4 64

Старший номер устройства - это тип устройства, который указывается в таблице ключей, а младший номер - это номер устройства данного типа.

Возникновение прерывания побуждает ядро запускать программу обработки прерывания для данного типа устройств (тип устройства определяется по вектору), передавая ей номер устройства.

ПО ввода/вывода

Ключевые моменты:

1) независимость от устройств. Вид программы не должен зависеть от того, читает ли она данные с гибкого диска или с жесткого диска.

2) обработка ошибок. Ошибки следует обрабатывать как можно ближе к аппаратуре.

3) буферизация. Данные с устройств невозможно сразу записать туда, куда надо.

А) IP – пакет ядро/драйвер скачивает в буфер

Б) формирование звука

В) запись CD/DVD

Для решения поставленных проблем целесообразно разделить программное обеспечение ввода-вывода на четыре слоя (см. рис.)

1) Обработка прерываний
2) Драйверы устройств
3) Независимый от устройств слой операционной системы
4) Пользовательский слой программного обеспечения.

Многоуровневая организация подсистемы ввода-вывода

Способы работы с устройствами I/O

1) Программный (простой)

2) Управляемый прерываниями.

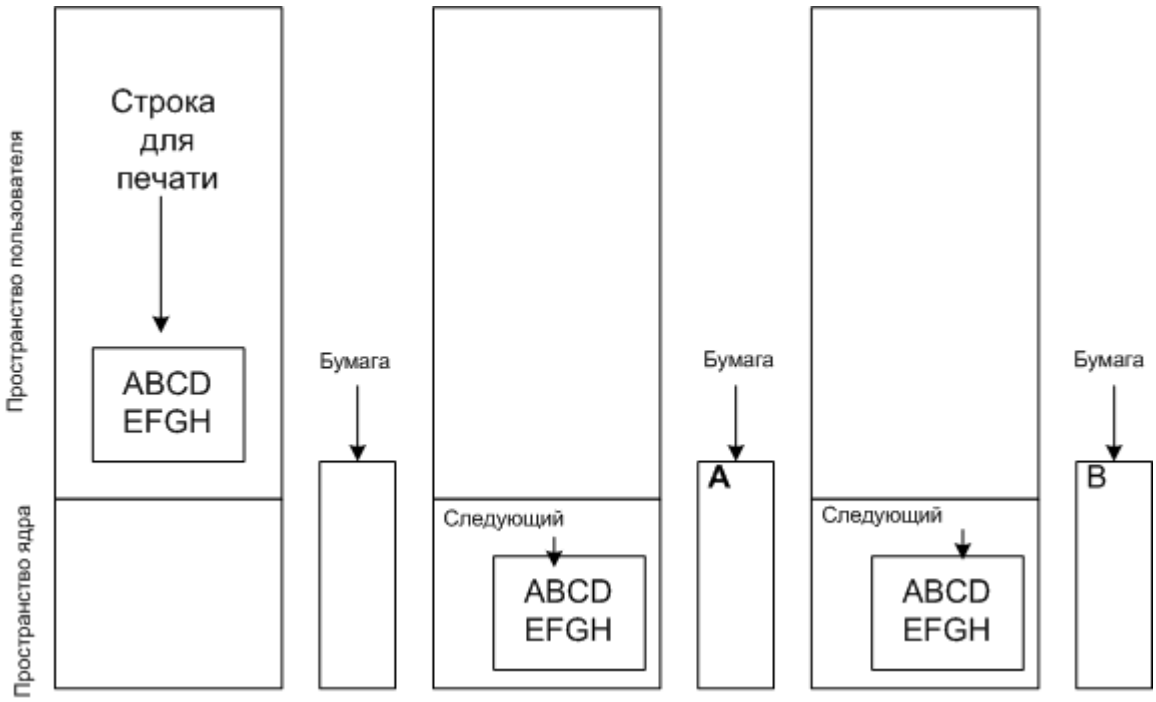
3) С использованием DMA.

Рассмотрим подробнее:

1) Программный ввод-вывод

В этом случае всю работу выполняет центральный процессор.

Рассмотрим процесс печати строки ABCDEFGH этим способом.



Этапы печати строки ABCDEFGH

Алгоритм печати:

1. Строка для печати собирается в пространстве пользователя.
2. Обращаясь к системному вызову, процесс получает принтер.
3. Обращаясь к системному вызову, процесс просит распечатать строку на принтере.
4. Операционная система копирует строку в массив, расположенный в режиме ядра.
5. ОС копирует первый символ в регистр данных принтера, который отображен на памяти.
6. Символ печатается на бумаге.
7. Указатель устанавливается на следующий символ.
8. Процессор ждет, когда бит готовности принтера выставится в готовность.
9. Все повторяется.

При использовании буфера принтера, сначала вся строка копируется в буфер, после этого начинается печать.

Программа:

```
Copy_from_uesr (buf, p, count);

For (i=0; i<count; i++)

{ while (*printer_status_reg)!=READY;          ждем, пока принтер станет доступным

*printer_data_reg=p[i];}
```

2) Управляемый прерываниями ввод-вывод

Если в предыдущем примере буфер не используется, а принтер печатает 100 символов в секунду, то на каждый символ будет уходить 10мс, в это время процессор будет простаивать, ожидая готовности принтера.

Рассмотрим тот же пример, но с небольшим усовершенствованием.

Алгоритм печати:

1. До пункта 8 тоже самое.
2. Процессор не ждет готовности принтера, а вызывает планировщик и переключается на другую задачу. Печатающий процесс блокируется.
3. Когда принтер будет готов, он посылает прерывание процессору.
4. Процессор переключается на печатающий процесс.

Программа:

```
Copy_from_uesr (buf, p, count);

enable_interrupts();          разрешение прерывания

 while (*printer_status_reg)!=READY;

*printer_data_reg=p[0];           записываем 1ый символ

scheduler();          планировщик; переходим к выполнению другой задачи

if (count>0)

{ *printer_data_reg=p[i];

i++; count--;}

else

{unblok_user();}          когда все напечаталось, блокируем
```

3) Ввод-вывод с использованием DMA

Недостаток предыдущего метода в том, что прерывание происходит при печати каждого символа.

Алгоритм не отличается, но всю работу на себя берет контроллер DMA, а не ЦП.

Программа аналогичная, ее выполняет контроллер DMA.

Взаимодействие процессов

Механизмы взаимодействия позволяют процессам обмениваться данными и синхронизировать выполнение.

Способы взаимодействия:

1. использование ptrace (взаимодействует отладчик и отлаживаемая программа)

2. передача сигналов (передаются только сигналы, данные передавать невозможно);

(kill – передаются; signal – установка обработчика)

3. неименованные каналы (pipe. Может взаимодействовать процесс и его потомки);

4. именованные каналы (специальный файл, mkmod - создание);

5. используя текст IPC (межпроцессорное взаимодействие);

6. через систему сокетов (взаимодействие по сети).

Сигналы

За многими сигналами закреплены специальные функции.

signal.h - соответствуетсигналу мнемокоду

#define SIGHUP 15

SIGHUP (Hang Up) опускание трубки телефона , заверш. управл. процесс

SIGINT (Interact) Ctrl+C прерывание с клавиатуры

QUIT- выход

ILL – неверная инструкция

FPE – деление на 0

KILL – нельзя обработать процессом

SEGV – нарушение сегментации

PIPE – возникновение проблем в конвейере

ALRM – сигнал будильника

TERM – один из основных сигналов для завершения процесса

USR1 – не закреплены никакие функции

USR2 – можно определить самому

CHLD – порожденный процесс завершился

STOP – не обрабатывается процессами

CONT - продолжение

PWR – нет напряжения в сети

dd – снимает образы CD/DVD

dd if = /dev/cdrom of = /dev/hda2

ps – номер процесса

dd – обраб. сигн. USR1 USR2

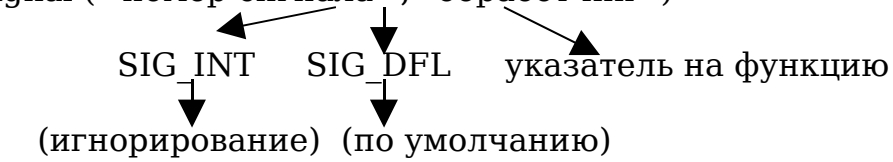
kill посылает сигналы. Соответствует вызову:

kill (<номер процесса> <номер сигнала>)

Если задавать сигнал в форме SIG…, необходимо подключить signal.h

Сигналы обрабатываются асинхронно. Обработчик сигнала устанавливается с помощью вызова:

signal (<номер сигнала>,<обработчик>)



Функция – обработчик:

void <имя> (int <номер сигнала>)

Пример:

#include <signal.h>

#include <stdio.h>

void obr (int n)

```
{   printf (‘%d’,n);
    fflush (stdout);} // чистит буфер вывода, чтобы выводил сразу
main ()
{   signal(SIGUSR2, obr);
    while (1);}
// kill – SIGUSR2 <номер процесса>
// while (1); - грузит ЦП, поэтому
while (1); pause;
// БУДИЛЬНИК
wakeup ()
{   printf (“Я проснулся”);
    fflush (stdout);}
main ()
{   signal (SIGALRM, wakeup);
    while (1)
            {alarm (5); // сигнал будильника с задержкой на 5 секунд
            pause();
}}
// вместо alarm(5) имитация его kill
kill (getpid(),SIGALRM); //возвращает номер процесса
wakeup ()
{   printf (‘умираю’);
    exit(0);
}
wakeup1 ()
{   printf (‘успел’);
```

```
    wait(0);
}
main ()
{   char buf [16];
    int nch, inp; // номер порожд. процесса, дескриптор файла
signal (SIGALRM, wakeup);
signal (SIGCHLD, wakeup1);
if (nch = fork()) // родительский
{   sleep(1);
    kill (nch, SIGALRM);
}
else // потомок
{   inp = open (“/dev/tty”, )_RDOHLY);
    read (inp, buf, sizeof (buf));
}}
```

sigation – лучший вариант, чем signal

sigprocmask – можно задавать номер сигнала, который будут игнорировать

Неименованные каналы

Применяются только при взаимодействии между процессом и его потомком.

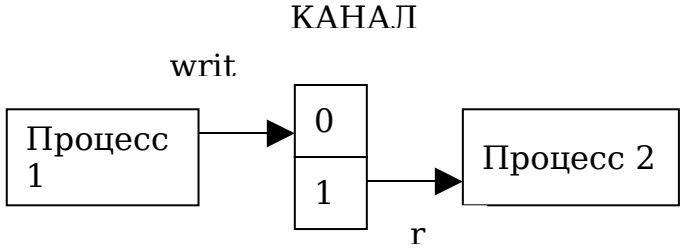
Создается дескриптор, состоящий из двух элементов:

Командой pipe (fdp) он определяется.

pipe - создание неименованного канала

```
int pipe (fildes)
int fildes [2];
```

Системный вызов pipe создает механизм ввода/вывода, называемый каналом, и возвращает два дескриптора файла fildes[0] и fildes[1]. Дескриптор fildes[0] открыт на чтение, дескриптор fildes[1] - на запись.



Канал буферизует до 5120 байт данных; запись в него большего количества информации без считывания приведет к блокированию пишущего процесса. Посредством дескриптора fildes[0] информация читается в том же порядке, в каком она записывалась с помощью дескриптора fildes[1].

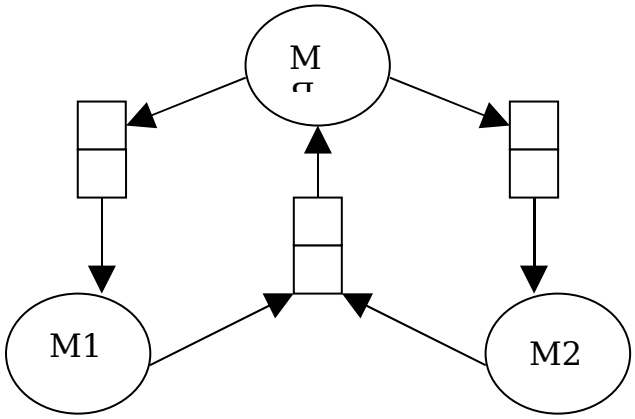
Системный вызов pipe завершается неудачей, если выполнено хотя бы одно из следующих условий:

1) Превышается максимально допустимое количество файлов, открытых одновременно в одном процессе.

2) Переполнена системная таблица файлов.

При успешном завершении результат равен 0; в случае ошибки возвращается -1, а переменной errno присваивается код ошибки.

Использовать один канал для двух сторон обмена неудобно. Взаимодействуют только родственные каналы.



Если используется один канал, сделать запись в него, а потом прочитать то, что только что было в него записано. Для двустороннего взаимодействия можно создать второй канал.

Пример:

Записывать будем с помощью write (fdp [1]);

Читать с помощью read (fdp [0]).

main ()

```
{   char buf [2];
    int to [2], from [2];
    buf [1] = '\0'; //конец строки
    pipe (to); pipe (from);
    if (fork()) //родитель
    {       signal (SIGCHLD, kid);
            while(1)
            {       scanf ('%c', &buf [0]);
                    write (to [1], buf, 2);
                    read (from [0], buf, 2);
                    printf ('%s', buf);
}}
else //порожденный
{   while(1)
    {       read (to [0], buf, 2);
            if (buf [0] == 'e') exit(0);
            buf [0]++;
    write (from [1], buf, 2);
}}}
void kid()
{   wait (0);
    exit (0); //завершение в родит. процесс
}
```

Именованный канал

При работе с именованным каналом создаем специальный файл:

mknod filename p, где p - тип файла.

Работа с именованным каналом производится также как с обычным файлом (теми же системными вызовами), за исключением того, что реально на диске информация не сохраняется, а передается от процесса, вызвавшего запись, процессу, вызвавшему чтение.

Пример:

```
main ()
{   int rd = open (‘имя ф. канала’, O_WRONLY);
    write (fd, “Hello”, 6);
    close (fd);
}
main ()
{   char buf [16];
    int fd = open (‘имя ф. канала’,O_RDONLY);
    read (fd, buf,16);
    printf (“%s”, buf);
    close (fd);
}
```

IPC

В IPC содержится три пакета взаимодействия:

1. механизм сообщений;

2. механизм распределения памяти;

3. семафоры.

	Сообщения	Память	Семафоры
Создание	msgget	shmget	semget
Работа	msgctl	shmctl	semctl
Настройка	msgrcv msgsnd	shmat shmdt	semop

1. Механизм сообщений позволяет принимать и посылать потоки сформированных данных.

За передачу сообщений отвечают четыре системных вызова:

msg get ≈ возвращает дескриптор сообщения;

msg clt ≈ устанавливает параметры сообщений;

msg cnt ≈ передаᵀт сообщение;

msg rcv ≈ принимает сообщение.

2. Механизм распределения памяти позволяет совместно использовать отдельные части виртуального адресного пространства.

shm get ≈ создает новую область;

shm at ≈ логически присоединяет;

shm dt ≈ логически отсоединяет;

shm ctl ≈ работает с параметрами области.

3. Семафоры синхронизацию выполнения параллельных процессов. В любой момент времени над

семафором возможна только одна реализация.

sem get ≈ создание набора семафоров;

sem ctl ≈ управление этим набором;

sem op ≈ работа со значениями.

Семафоры

Используются для синхронизации выполнения приложений и защиты критических секций.

```
#defain SEMKEY  77

union semun //одно из полей буде использоваться

{

int val;

struct semid_ds *bat;
```

```
unsigned short *array;
struct seminfo *buf;
}
main()
{
union semun inisem; //для инициализации
unshort ainisem[1]={1};
int semid;
int i,j,pid;
struct sembuf p,v;
semid=semget(SEMKEY,1,0777|[IPC_CREAT]); //создание, IPC_CREAT - макрос создания, 1 – число символов, 0777 - моды доступа
inisem=ainisem; //команда
semctl(semid,0,SETALL,inisem); //инициализация
p.sem_num=0;
p.sem_op=-1;
p.sem_flg=SEM_UNDO;
v.sem_num=0;
v.sem_op=1;
v.sem_flg=SEM_UNDO;
fork();
pid=getpid(); //определяем номер процесса
for(i=0;;i<10;i++)
{
semop(semid,&p,1);  // сколько операций выполнено
//критическая секция
for(j=0;g<5;g++)
```

```
{
    printf(“%d%d”, pid, j );
    semop(semid,&v,1);
}}
```

Механизм передачи сообщений

Системный вызов msgget()

Системный вызов msgget предназначен для выполнения операции доступа к очереди сообщений и, в случае ее успешного завершения, возвращает дескриптор для этой очереди (целое неотрицательное число).

```
int msgget(key_t key, int msgflg);
```

Описание системного вызова

Параметр key является ключом для очереди сообщений. Параметр msgflg – флаги – играет роль только при создании новой очереди сообщений и определяет права различных пользователей при доступе к очереди.

Возвращаемое значение

Системный вызов возвращает значение дескриптора System V IPC для очереди сообщений при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов msgsnd()

Вызов msgsnd() копирует пользовательское сообщение в очередь сообщений.

```
int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag);
```

Описание системного вызова

Системный вызов копирует сообщение, расположенное по адресу, на который указывает параметр ptr, в очередь сообщений, заданную дескриптором msqid.

Параметр flag может принимать два значения: 0 и IPC_NOWAIT.

Возвращаемое значение

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов msgrcv()

```
int msgrcv(int msqid, struct msgbuf *ptr, int length, long type, int flag);
```

Описание системного вызова

Системный вызов msgrcv предназначен для получения сообщения из очереди сообщений, т. е. является реализацией примитива receive.

Параметр msqid является дескриптором для очереди, из которой должно быть получено сообщение.

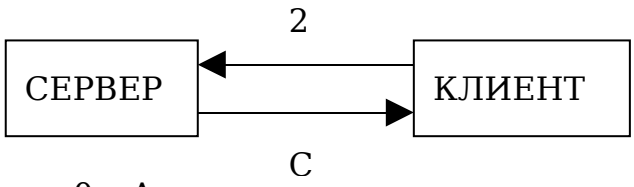
Параметр length должен содержать максимальную длину полезной части информации.

Параметр flag может принимать значение 0 или быть какой-либо комбинацией флагов IPC_NOWAIT и MSG_NOERROR.

Возвращаемое значение

Системный вызов возвращает при нормальном завершении действительную длину полезной части информации (т. е. информации, расположенной в структуре после типа сообщения), скопированной из очереди сообщений, и значение -1 при возникновении ошибки.

Все прочитанные сообщения по умолчанию удаляются из очереди.



0 – А
1 – Б
2 - С

Сервер:

```
#define MSGKEY 81

struct msgform {

long mtype; //идентификатор сообщения

char text [256];  //данные

}

main ()

{char mas [] = {'a','b','c'};
```

```
struct msgform msg;
int msgid, *pint;
char *pchar;
msgid = msgget (MSGKEY, 0777 IPC_CREATE);
pint = (int*) msg.txt;
pchar = (char*) msg.text;
msgrcv (msgid, &msg, sizeof (int), 8, 0) //8 – идентификатор
*pchar = mas [*pint];
msg.mtype = 9;
msgsnd (msgid, &msg, sizeof(char),0); // 0 - флаг
}
```

Клиент:

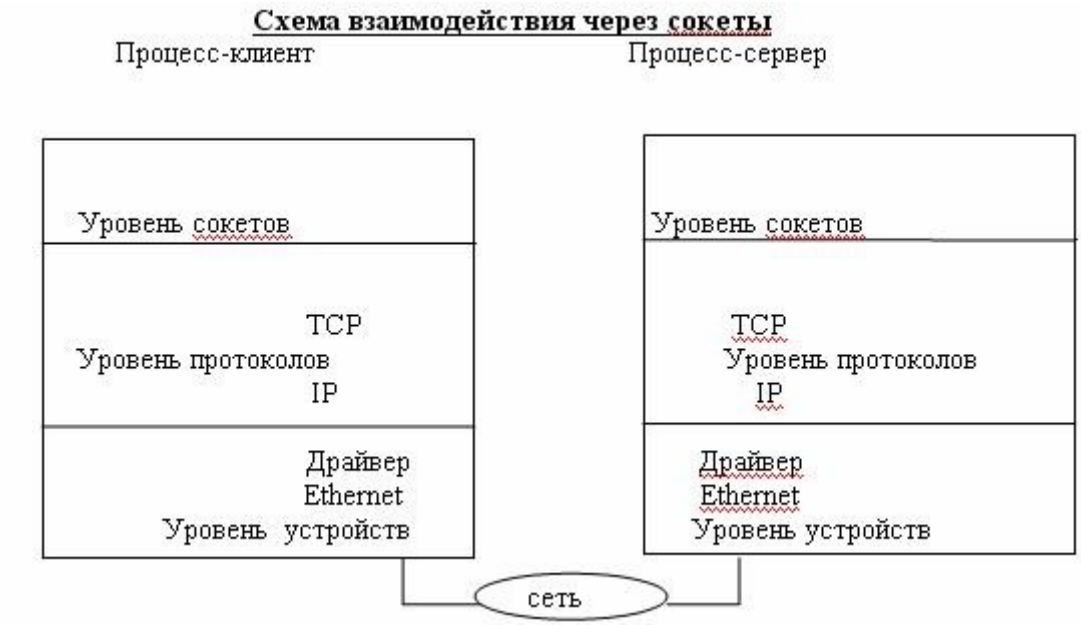
```
#define MSGKEY 81
struct msgform {
long mtype; //идентификатор сообщения
char text [256];  //данные
}
main ()
{char mas [] = {‘a’,’b’,’c’};
struct msgform msg;
int msgid, *pint, g;
char *pchar;
msgid = msgget (MSGKEY, 0777);
pint = (int*) msg.txt;
pchar = (char*) msg.text;
scanf (‘%d’, &g);
```

```
*pint = g;

msg.mtype = 8;

msgsnd (msgid, &msg, sizeof(int),0); // 0 – флаг

msgrcv (msgid, &msg, sizeof(char), g , 0);

printf (‘номер %d у буквы %c’, g , *pchar);

}
```

Сокеты

Сокеты - универсальные методы взаимодействия процессов на основе использования многоуровневых сетевых протоколов. Сокеты предназначены для работы по сети: блокируют драйверы, для удобства выполнения локальных подпрограмм. Сокеты используют для работы по IP – сетям.

Используют клиент-серверный механизм.



Сокеты находятся в областях связи (доменах). Домен сокета - это абстракция, которая определяет структуру адресации и набор протоколов. Сокеты могут соединяться только с сокетами в том же домене. Всего выделено 23 класса сокетов (см. файл `<sys/socket.h>`), из которых обычно используются только UNIX-сокеты и Интернет-сокеты.

Поддерживаются домены:

- "UNIX system" - для взаимодействия процессов внутри одной машины

- "Internet" (межсетевой) - для взаимодействия через сеть с помощью протокола

У сокета 3 атрибута:

1) домен

2) тип

3) протокол

Для создания сокета используется системный вызов socket.

s = socket(domain, type, protocol);

Например, для использования особенностей Internet, значения параметров должны быть следующими:

s = socket(AF_INET, SOCK_STREAM, 0);

Основные типы сокетов:

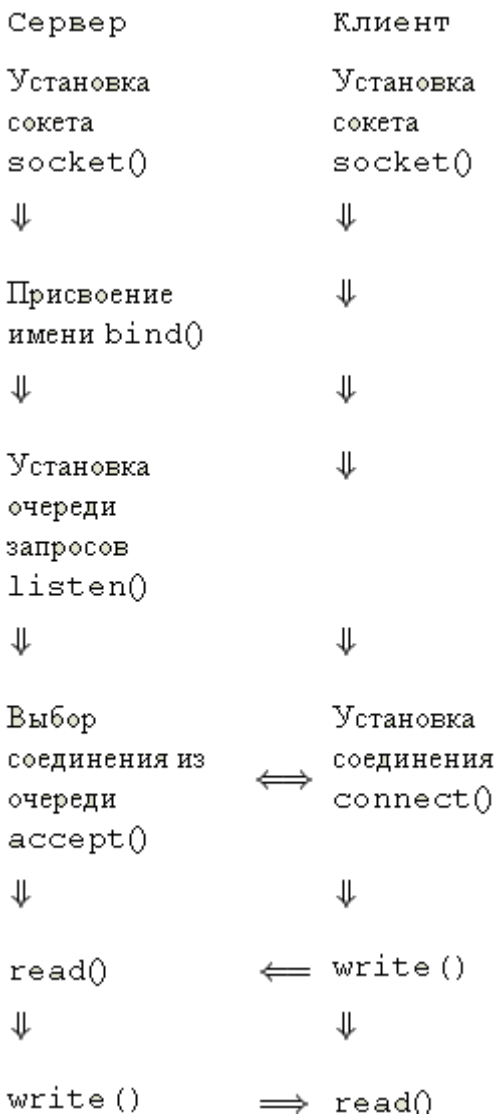
1) Поточный
- обеспечивает двухсторонний, последовательный, надежный, и недублированный поток данных без определенных границ. Тип сокета - SOCK_STREAM, в домене Интернета он использует протокол TCP.

2) Датаграммный
- поддерживает двухсторонний поток сообщений. Приложение, использующее такие сокеты, может получать сообщения в порядке, отличном от последовательности, в которой эти сообщения посылались. Тип сокета - SOCK_DGRAM, в домене Интернета он использует протокол UDP.

3) Сокет
последовательных пакетов - обеспечивает двухсторонний, последовательный, надежный обмен датаграммами фиксированной максимальной длины. Тип сокета - SOCK_SEQPACKET. Для этого типа сокета не существует специального протокола.

4) Простой
сокет - обеспечивает доступ к основным протоколам связи.

Обмен между сокетами происходит по следующей схеме:



Типы передаваемых пакетов:

SOCK_STREAM соответствует потоковым сокетам, реализующим соединения «точка-точка» с надежной передачей данных.

SOCK_DGRAM указывает датаграммный сокет. Датаграммные сокеты осуществляют ненадежные соединения при передаче данных по сети и допускают широковещательную передачу данных.

SOCK_RAVE для низкоуровневого управления пакетами данных

AF_INET сокет для работы по сети

AF_UNIX соответствует сокетам в файловом пространстве имен

Причины успеха сокетов заключаются в их простоте и универсальности. Программы, обменивающиеся данными с помощью сокетов, могут работать в одной системе и в разных, используя для обмена данными как специальные объекты системы, так и сетевой стек. Как и каналы, сокеты используют простой интерфейс, основанный на «файловых» функциях read(2) и write(2) (открывая сокет, программа

Unix получает дескриптор файла, благодаря которому можно работать с сокетами, используя файловые функции), но, в отличие от каналов, сокеты позволяют передавать данные в обоих направлениях, как в синхронном, так и в асинхронном режиме.

Действия с сокетами

Установка связи:

Со стороны клиента связь устанавливается с помощью стандартной функции connect, которая инициирует установление связи на сокете, используя дескриптор сокета s и информацию из структуры *serveraddr*, имеющей тип *sockaddr_in*, которая содержит адрес сервера и номер порта на который надо установить связь:

error = connect(s, serveraddr, serveraddrlen);

Со стороны сервера процесс установления связи сложнее. Для этих целей используется системный вызов listen:

error = listen(s, qlength);

где **s** это дескриптор сокета, а **qlength** это максимальное количество запросов на установление связи, которые могут стоять в очереди.

Передача данных:

Когда связь установлена, с помощью различных функций может начаться процесс передачи данных. При наличии связи, пользователь может посылать и получать сообщения с помощью функций read и write:

write(s, buf, sizeof(buf)); read(s, buf, sizeof(buf));

Вызовы send и recv практически идентичны read и write, за исключением того, что добавляется аргумент флагов.

send(s, buf, sizeof(buf), flags); recv(s, buf, sizeof(buf), flags);

Закрывание сокетов:

Если сокет больше не используется, процесс может закрыть его с помощью функции close, вызвав ее с соответствующим дескриптором сокета:

close(s);

Удаленный вызов процедур

Вызова удаленных процедур *(Remote Procedure Call - RPC)* - механизм передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных

через сеть. Алгоритм: клиент производит вызов процедуры, посылая пакет данных на сервер. Средства удаленного вызова процедур предназначены для облегчения организации распределенных вычислений.

Ключевые моменты (отличия от вызовов локальных процедур):

1) Реализация удаленных вызовов существенно сложнее реализации вызовов локальных процедур. Поскольку вызывающая и вызываемая процедуры выполняются на разных машинах, то они имеют разные адресные пространства, и это создает проблемы при передаче параметров и результатов.

2) Значения параметров должны копироваться с одного компьютера на другой.

3) В реализации RPC участвуют как минимум два процесса - по одному в каждой машине. В случае, если один из них аварийно завершится, могут возникнуть неприятные ситуации.

4) Существует ряд проблем, связанных с неоднородностью языков программирования и операционных сред.

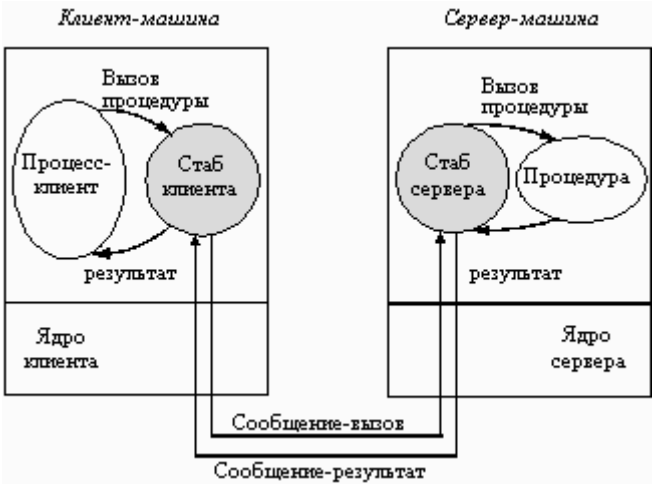
3 уровня RPC

- верхний(highest), совершенно прозрачен для операционной системы машины и сети. Вызов удаленной процедуры производиться так же как обычные Си функции.

- средний(inter medium), настоящий RPC с одной стороны программист сам определяет удаленную процедуру, организует кодировку передаваемых данных. С другой стороны не работает напрямую с сокетами.

- нижний(low), редко используется, позволяет детально программировать взаимодействия по RPC.

Взаимодействие программных компонентов при выполнении удаленного вызова процедуры иллюстрируется рисунком.



Следующий рисунок показывает последовательность команд, которую необходимо выполнить для каждого RPC-вызова

Архитектуры многопроцессорных систем

Многопроцессорная архитектура включает в себя два и более ЦП, совместно использующих общую память и периферийные устройства.

Многопроцессорная конфигурация

Среди архитектур систем параллельной и распределенной обработки известны симметричные многопроцессорные системы -SMP

(Symmetrical Multiprocessing), системы массивно-параллельной обработки MMP (Massively-Parallel Processing), а также кластерные системы (RMC and NUMA). Кластеры с рефлекторной памятью RMC (Reflecting Memory Cluster) являются кластерами с механизмом передачи содержимого памяти между узлами с блокировкой. Системы с несимметричным доступом к памяти NUMA (Non Uniform Memory Access) объединяют узлы с распределенной памятью, к которой обеспечен несимметричный доступ как к общей памяти.

Распределенные вычисления нужны для разбиения программ на части. Выполняются на разных компьютерах.

Используются:

- для повышения производительности

-для повышения надежности решения естественно параллельных задач.

Архитектура распределительных систем

1. Master -slave(именная организация)

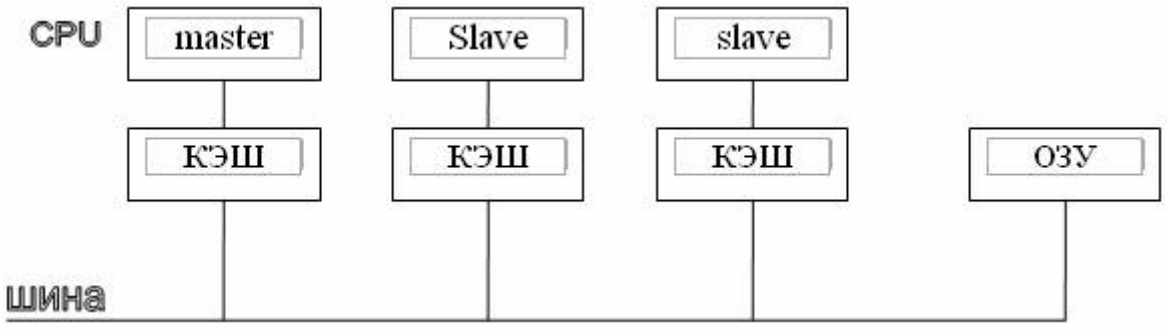
Система с двумя процессорами, один из которых - главный (master) - может работать в режиме ядра, а другой - подчиненный (slave) - только в режиме задачи.

Главный процессор несет ответственность за обработку всех обращений к операционной системе и всех прерываний. Подчиненные процессоры ведают выполнением процессов в режиме задачи и информируют главный процессор о всех производимых обращениях к системным функциям.

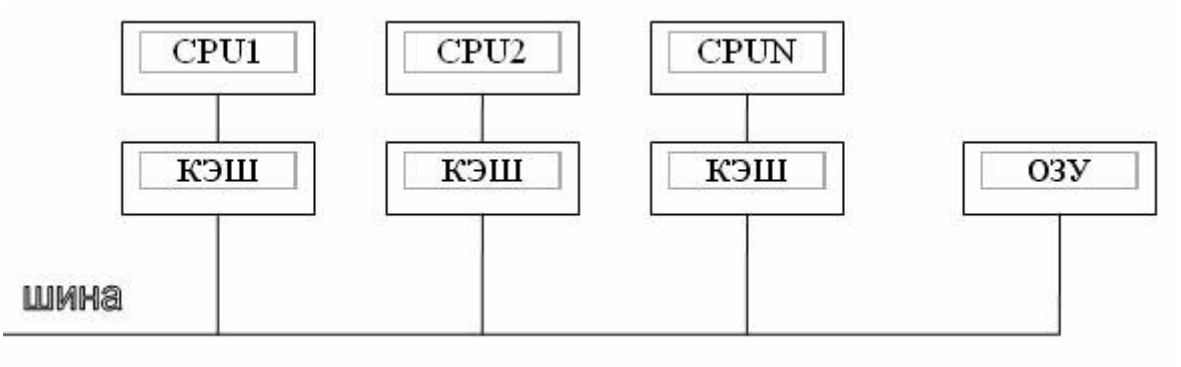
master - в режиме ядра выполняются процессы

slave - процессы выполняются в пользовательском режима

master становиться слабым местом при увеличении slave



2.Симметричная архитектура (шинная организация)

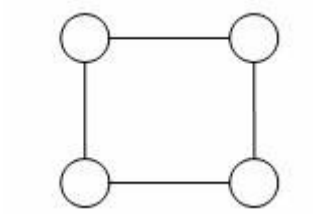


Минус шинной организации - пропускная способность шины ограничена, при увеличении числа CPU шина забивается. Проблема масштабируемости.

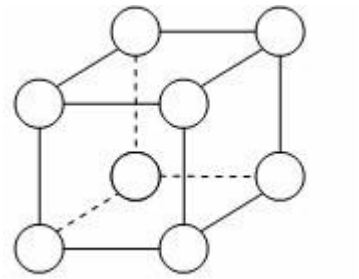
3.Гиперкубическое соединение процессорных модулей

Для каждого процесса выделяется своя память.

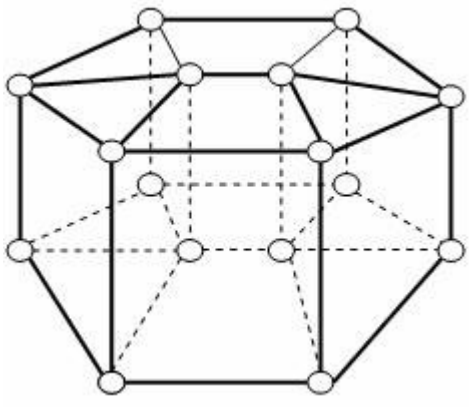
(CPU + ОЗУ + ШИНА + АДАПТЕР МЕЖМОДУЛЬНЫЕ СОЕДИНЕНИЯ)



Элемент, состоящий из памяти, 4-х ЦП и средства коммуникации.

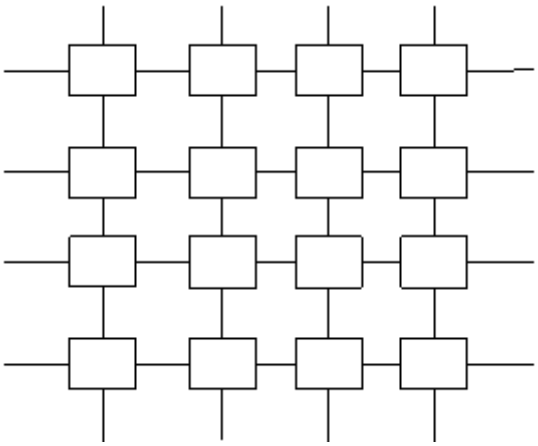


Компьютеры CRAY



Максимальный размер 16 модулей 64 процессора

4.Транспьютер



Программы написаны на параллельных языках

5.Кластер - компьютеры, объединенные быстрой локальной сетью