[1. Функции ФС. Низкоуровневое и высокоуровневое форматирование. Разделы. Логический раздел, логический диск, кластеры. 2](#_Toc346722598)

[2. Системные вызовы: назначение, реализация в различных ОС. Синхронные и асинхронные вызовы. 3](#_Toc346722599)

[3. Функции генерации сигналов (POSIX): названия, аргументы, принцип работы. 4](#_Toc346722600)

[4. Программные ресурсы, реентерабельные и повторно входимые модули, мотивация их применения при проектировании ОС и её компонентов 5](#_Toc346722601)

[5. Обращение к файлам из процесса. Этапы, системные таблицы, идентификации файлов и процесса. 6](#_Toc346722602)

[6. Организация именованных/неименованных каналов. Ограничесния, функции, примеры применения 8](#_Toc346722603)

[7. Анализ архитектуры и моделей функционирования ОС линейки Windows. 10](#_Toc346722604)

[8. Взаимодействие обработчиков событий в ОС:при наличии аппаратных событий, внутренних прерываниях, исключениях 11](#_Toc346722605)

[9. Резидентные программы. Пример реализации. 13](#_Toc346722606)

[10. Структура ядра ОС с иерархической архитектурой. Алгоритмы выполнения запросов прикладного уровня ……………………………………………………………………………………………………………….14](#_Toc346722607)

[11. Синхронный и асинхронный ввод/вывод в NT, формат IRP, проекционный ввод/вывод. 15](#_Toc346722608)

[12. Причины генерации сигналов.Примеры.Диспозиция. Пример кода C++ запуска.деление на 0. 16](#_Toc346722609)

[13. Многозадачность (мультипрограммирование). Реализация в различных ОС. 17](#_Toc346722610)

[14. Трудоемкость сигналов 18](#_Toc346722611)

[15. Реализация сообщений в различных ОС 19](#_Toc346722612)

[16. Функцион комп-ты сетев. ОС. Их взаимод-ие и способы реализации. 20](#_Toc346722613)

[17. взаимосвязь ф-ий exec(). Алгоритм загр-ки исполняемого файла 21](#_Toc346722614)

[18. Способы низкоуровнего отключ-ия прерываний. 22](#_Toc346722615)

[19. Задача планирования и диспетчиризации.Типы планировщиков.Планирование в системах реального времени.Разделение времени. 23](#_Toc346722616)

[20. убить тред с клавиатуры (программа) 24](#_Toc346722617)

[21. Совместимость и множественные прикладные среды.Реализация множественных прикладных сред. ……………………………………………………………………………………………………………….25](#_Toc346722618)

[22. Критерии показателей ОС различных классов.(1 лекция) 26](#_Toc346722619)

[23. Структуры данных ФС UNIX подобных ОС(чето там про vnod)Назначение и основной состав. 27](#_Toc346722620)

[24. Реализация сервисов по запросу от приложений в ОС с различными архитектурами. Два подхода к реализации клиент-серверной модели 28](#_Toc346722621)

[25. Индексный дескриптор, идентификатор файла, ссылки.... назначение... 29](#_Toc346722622)

[26. Потоки в NT, сервисы, приоритеты, атрибуты.... 30](#_Toc346722623)

[27. Драйвер, структура (набор процедур). Многослойный, простейший, с прерываниями... 31](#_Toc346722624)

[28. записи каталогов различных файловых систем 32](#_Toc346722625)

[29. структура сист. ввода-вывода NT, функции драйвера и диспетчера 33](#_Toc346722626)

[30. вызов подпрограмм и задач.(через шлюз) 34](#_Toc346722627)

[31. Совместимость ОС. Прикладные среды, реализация (напр. WinNT). **Совместимость** 40](#_Toc346722628)

[32. Жизненный цикл сигнала (unix). 40](#_Toc346722629)

[33. файловая система FAT и HPFS 40](#_Toc346722630)

1. Функции ФС. Низкоуровневое и высокоуровневое форматирование. Разделы. Логический раздел, логический диск, кластеры.

Основные функции любой файловой системы нацелены на решение следующих задач:

* именование файлов;
* программный интерфейс работы с файлами для приложений;
* отображения логической модели файловой системы на физическую организацию хранилища данных;
* организация устойчивости файловой системы к сбоям питания, ошибкам аппаратных и программных средств;
* содержание параметров файла, необходимых для правильного его взаимодействия с другими объектами системы (ядро, приложения и пр.).

В многопользовательских системах появляется ещё одна задача: защита файлов одного пользователя от несанкционированного доступа другого пользователя, а также обеспечение совместной работы с файлами, к примеру, при открытии файла одним из пользователей, для других этот же файл временно будет доступен в режиме «только чтение».

**Низкоуровневое форматирование** (*Low level format*) — операция, в процессе которой на магнитную поверхность диска наносятся т. н. сервометки — служебная информация, которая используется для позиционирования головок жёсткого диска. Выполняется в процессе изготовления жёсткого диска, на специальном оборудовании, называемом серворайтером.

Низкоуровневое форматирование — это процесс нанесения информации о позиции треков и секторов, а также запись служебной информации для сервосистемы. Этот процесс иногда называется "настоящим" форматированием, потому что он создает физический формат, который определяет дальнейшее расположение данных. Когда в первый раз запускается процесс низкоуровневого форматирования винчестера, пластины жесткого диска пусты, т.е. не содержат абсолютно никакой информации о секторах, треках и т.п. Это последний момент, когда у жесткого диска абсолютно пустые пластины. Информация, записанная во время этого процесса, больше никогда не будет переписана.

Высокоуровневое форматирование — процесс, который заключается в создании главной загрузочной записи с таблицей разделов и (или) структур пустой файловой системы, установке загрузочного сектора и т. п. В процессе форматирования также проверяется целостность носителя для блокировки дефектных секторов. Известен также способ без проверки носителя, который называется «быстрое форматирование». После завершения процесса низкоуровневого форматирования винчестера, мы получаем диск с треками и секторами, но содержимое секторов будет заполнено случайной информацией. Высокоуровневое форматирование — это процесс записи структуры файловой системы на диск, которая позволяет использовать диск в операционной системе для хранения программ и данных.

**Раздел**  — часть долговременной памяти жёсткого диска, выделенная для удобства работы, и состоящая из смежных блоков. Всего разделов на физическом диске может быть не более 4-х. Из них не более одного расширенного(дополнительного) раздела

Виды логических разделов:

Первичный (основной) раздел

В ранних версиях Microsoft Windows Первичный раздел (primary partition) обязательно должен был присутствовать на физическом диске первым. Соответственно, эти операционные системы могли быть установлены только на первичный раздел. Этот раздел всегда содержит только одну файловую систему. При использовании MBR, на физическом диске может быть до четырёх первичных разделов.

Расширенный (дополнительный) раздел

Основная таблица разделов может содержать не более 4 первичных разделов, поэтому был изобретён Расширенный раздел (extended partition). Это первичный раздел, который не содержит собственной файловой системы, а содержит другие логические разделы.

Кластер (cluster) — в некоторых типах файловых систем логическая единица хранения данных в таблице размещения файлов, объединяющая группу секторов

1. Системные вызовы: назначение, реализация в различных ОС. Синхронные и асинхронные вызовы.

Системные вызовы – возможность обращения к кодовым сегментам ОС из приложений. В результате обеспечивается возможность перехода в привилегированный режим.

Достоинства:

1. Высокая скорость вызова процедур ОС;
2. Универсальность обращения (одинаковые обращения для всех аппаратных платформ);
3. Контроль со стороны ОС за корректностью выполнения системных вызовов;
4. Легкая масштабируемость системных вызовов (22-32 для большинства ОС).

2 способа передачи управления при использовании программного прерывания:

- децентрализованный – каждому системному вызову соответствует свой вектор прерывания;

- централизованный – в большинстве ОС основан на использовании диспетчера системных вызовов.

Децентрализованный способ является более быстрым, но менее гибким (вектор прерывания указывается в запросе). Централизованный способ – всегда используется один и тот же вектор прерывания.

Пред выполнением программы прерывания приложение передает в ОС номер системного вызова. Номер является индексом в таблице системных вызовов (табличный способ реализации системных вызовов). Аргументы системных вызовов в таком случае передаются через стек. При выполнении системного вызова производится копирование пользовательского стека. Далее производится чтение аргументов из системного стека. Такой механизм используется в большинстве ОС. Достоинство такой реализации – легко расширяется набор системных вызовов.

Системные вызовы могут быть организованы как синхронно, так и асинхронно, в некоторых ОС – это как режим.

Синхронный вызов – более простой способ: процесс приостанавливается и ожидает окончания системного вызова (блокируется до завершения системной процедуры). После завершения системного вызова планировщик переводит процесс в очередь готовых процессов. Как только возобновится выполнение процесса, он сможет воспользоваться результатами системного вызова.

Асинхронный вызов – блокировки процесса не происходит, выполняется определенный ряд действий, а затем передается управление тому процессу, который выполнялся (такой способ удобен при обращении к ПУ – сохраняется возможность продолжения вычислений исходным процессом).

При синхронном вызове не нужно отслеживать момент завершения системной процедуры – эту функцию берет на себя планировщик, но в этом случае менее экономно расходуются ресурсы.

В универсальных ОС используется блокирующие (синхронные) системные вызовы. В микроядерных – асинхронные системные вызовы – это тенденция.

1. Функции генерации сигналов (POSIX): названия, аргументы, принцип работы.

Функции генерации сигналов. Для генерации сигналов в Unix предусмотрены две функции – kill(2) и raise(3). Первая функция предназначена для передачи сигналов любым процессам, к которым владелец данного процесса имеет доступ, а с помощью второй функции процесс может передать сигнал самому себе. Как это обычно принято в мире Unix, семантика вызова функции kill()совпадает с семантикой одноименной команды ОС. У функции kill()два аргумента –PID процесса-приемника и номер передаваемого сигнала. С помощью функции kill()как и с помощью одноименной команды можно передавать сообщения не только конкретному процессу, но и группе процессов.

Таблица 2 демонстрирует поведение функции kill()в зависимости от значения PID:

|  |  |
| --- | --- |
| PID > 1 | Сигнал посылается процессу с соответствующим PID. |
| PID == 0 | Сигнал посылается всем процессам из той же группы что и процесс-источник. |
| PID < 0 | Сигнал посылается всем процессам, чей идентификатор группы равен абсолютному значению PID. |
| PID == 1 | Сигнал посылается всем процессам системы. |

1. Программные ресурсы, реентерабельные и повторно входимые модули, мотивация их применения при проектировании ОС и её компонентов

программные модули, т.к. именно они могут рассматриваться как программные ресурсы. Программные модули могут быть однократно и многократно или повторно используемыми. Однократно используемыми называют такие программные модули, которые могут быть правильно выполнены только один раз. Это означает, что в процессе выполнения они могут испортить себя: либо повреждается часть кода, либо – исходные данные, от которых зависит ход вычислений. Однократно используемые программные модули являются неделимым ресурсом. Обычно их не распределяют как ресурс системы, они используются только на этапе загрузки ОС. При этом собственно двоичные файлы, которые хранятся на системном диске и в которых записаны эти модули, не портятся, и могут быть повторно использованы при следующем запуске ОС.  
Повторно используемые программные модули могут быть непривилегированными, привилегированными и реентерабельными.  
Привилегированные программные модули работают в так называемом привилегированном режиме, т.е. при отключенной системе прерываний, так, что никакие внешние события не могут нарушить порядок вычислений. В результате программный модуль выполняется до своего конца, после чего он может быть вновь вызван на исполнение из другой задачи (другого вычислительного процесса). Со стороны по отношению к вычислительным процессам, которые попеременно в течение срока жизни вызывают некоторый привилегированный программный модуль, такой модуль будет выступать попеременно разделяемым ресурсом.  
Непривилегированные программные модули – это обычные программные модули, которые могут быть прерваны во время работы. В общем случае их нельзя считать разделяемыми, потому что если после прерывания выполнения такого модуля, исполняемого в рамках одного вычислительного процесса, запустить его еще раз по требованию другого вычислительного процесса, то промежуточные результаты для прерванных вычислений могут быть потеряны.  
В противоположность этому, реентерабельные программные модули допускают повторное многократное прерывания своего исполнения и повторный их запуск из других задач. Для этого реентерабельные программные модули должны быть созданы таким образом, чтобы было обеспечено сохранение промежуточных вычислений для прерываемых вычислений и возврат к ним, когда вычислительный процесс возобновляется с прерванной ранее точки. Это может быть реализовано двумя способами: с помощью статических и динамических методов выделения памяти под сохраняемые значения.  
Кроме реентерабельных программных модулей существуют еще повторно входимые. Это программные модули, которые допускают многократное параллельное использование, но в отличие от реентерабельных их нельзя прерывать.

1. Обращение к файлам из процесса. Этапы, системные таблицы, идентификации файлов и процесса.

Взаимосвязь системных таблиц иллюстрирует логическая схема управления параллельной обработкой 2-х файлов в 2-х процессах, показанная на следующем рисунке.

>

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **u\_ofile** | | ----------> | |  | | |  | | --- | | 0 | |  | |  | | --- | | Структуры | | **struct file** | |  | |  | |  | | --- | | **struct inode**      файла F1 | |
|  | |  | | --- | | 1 | |  | |  | | --- | | ::::::::: | |  |
| |  | | --- | | Контекст | | процесса | | I | | |  | | --- | | 2 | |  | |  | | --- | | ::::::::: | |  |
|  | |  | | --- | | 3 | | |  | | --- | | open F1 | | ---------> | |  | | |  | | --- | | **O\_RDONLY** | | |  | | --- | |  | | ---------> | |  | |
|  | |  | | --- | | 4 | | |  | | --- | | open F2 | | ---------> | |  | | |  | | --- | | **O\_RDONLY** | | |  | | --- | |  | | ---------> | |  | | |  | | --- | | **struct inode**      файла F2 | |
|  | |  | |  | | --- | | ::::::::: | |  |
|  | |  | | --- | | 4 | | |  | | --- | | open F2 | | ---------> | |  | | |  | | --- | | **O\_RDONLY** | | |  | | --- | |  | | ---------> | |  | |
|  | |  | | --- | | 3 | | |  | | --- | | open F2 | | ---------> | |  | | |  | | --- | | **O\_WRONLY** | | |  | | --- | |  | | ---------> | |  | |
| |  | | --- | | Контекст | | процесса | | II | | |  | | --- | | 2 | |  | |  | | --- | | ::::::::: | |  | |  | | --- | | **struct inode**      файла ... | |  |
|  | |  | | --- | | 1 | |  | |  | | --- | | ::::::::: | |  |
| |  | | --- | | **u\_ofile** | | ----------> | |  | | |  | | --- | | 0 | |  | |  | | --- | | ::::::::: | |  |
|  | |  | |  | | --- | | ::::::::: | |  |  |
| Рис. Схема управления файлами | | | | | |

Процесс I обрабатывает файлы *F2* и *F1*, которые открыты в нем для чтения. Их пользовательские дескрипторы равны **3** и **4** в контексте процесса I. Им соответствуют разные записи в*системной таблице файлов* и различные индексные дескрипторы в резидентной *таблице индексных дескрипторов*. Процесс II обрабатывает только файл *F2*, который открыт в нем отдельно по чтению и записи. Эти варианты доступа индексируются пользовательскими дескрипторами **3** и **4** в контексте процесса II. Им соответствуют разные записи в *системной таблице файлов*, но общий индексный дескриптор, причем тот же, что для файла *F2*, открытого процессом I.

Системная таблица файлов состоит из записей - по одной на каждое открытие файла в одном или различных процессах. Каждая запись системной таблицы файлов формально отражается структурой **struct file**, которая имеет следующие поля:

|  |  |  |
| --- | --- | --- |
| **f\_next, f\_prev** | - | указатели на следующую и предыдущую запись в системной таблице файлов; |
| **f\_flag** | - | режим открытия файла (запись, дополнение, чтение и т.д.); |
| **f\_offset** | - | смещение указателей чтение-записи при работе с файлом; |
| **f\_count** | - | число ссылок на данную запись из таблицы открытых файлов в контекстах процессов; |
| **f\_inode** | - | ссылка на соответствующий индексный дескриптор в резидентной таблице индексных дескрипторов. |

Расширение системной таблицы файлов - результат открытия или создания файлов в пользовательских процессах. При завершении обработки любого файла значение поля **f\_count**структуры **struct file** в соответствующей записи системной таблицы файлов уменьшается на 1. Запись с нулевым значением поля **f\_count** удаляется из системной таблицы файлов и вместе с ней исчезает ссылка (**f\_inode**) на соответствующий образ индексного дескриптора в таблице **inode**, уменьшая на 1 значение поля **i\_count** в его структуре **struct inode**.

Что касается идентификации процесса (process identification), то почти во всех операционных системах каждому процессу присваивается числовой идентификатор, который может быть просто индексом в первичной таблице процессов (рис. 3.8). В любом случае должно иметься некоторое отображение, позволяющее операционной системе найти по идентификатору процесса соответствующие ему таблицы. Идентификаторы могут использоваться в разных ситуациях. В частности, они используются для реализации перекрестных ссылок на таблицы процессов из других таблиц, находящихся под управлением операционной системы. Например, таблицы памяти могут предоставлять информацию об основной памяти с указанием всех областей, выделенных каждому из процессов, указываемому посредством его идентификатора. Аналогичные ссылки могут быть и в таблицах ввода-вывода или таблицах файлов. Если процессы обмениваются между собой информацией, их идентификаторы указывают операционной системе участников такого обмена. При создании нового процесса идентификаторы указывают родительский и дочерние процессы.

1. Организация именованных/неименованных каналов. Ограничесния, функции, примеры применения

**Каналы (pipe).**

Обмен данными между процессами порождает программный канал, обеспечивающий симплексную передачу между двумя процессами (задачами).

Системный вызов: int pipe(int \*filedes);

filedes[0] – указатель на запись;

filedes[1] – указатель на чтение;

По сути системный вызов вызывает 2 файловых дескриптора, которые позволяют открыть файл на запись и файл на чтение.

Эти каналы могут использоваться только родственными процессами.

#include <stdio.h> <unistd.h>

int main (void)

{

int fd [2];

char str[256];

char pipe\_str[256];

int err;

if((err = pipe(fd)) < 0) ОШИБКА СОЗДАНИЯ

if (!fork()) СЫН

{

printf("Entering SON function\n");

int input = fopen("input.txt","r");

if (input <=0) Ошибка

else {

printf("opened file: %s\n","input.txt");

fscanf(input,"%s",str);

write(fd[1],str,strlen(str));

printf("wrote data to pipe\n",str);

close(input);

close(fd[1]);

}

return 0;

}

else

{

printf("Entering FATHER function\n");

read (fd[0],pipe\_str,30);

close(fd[0]);

printf("read from pipe: %s\n",pipe\_str);

}

return 0;

}

**FIFO.**

В отличие от неименованных каналов (pipe) – возможен обмен данными не только между родственными процессами, так как буферизация происходит в рамках файловой системы с именованием.

Системный вызов: mknode(char \*pathname, int mode, int dev);

Правила открытия канала на запись и чтение для независимых процессов:

1) при чтении n байт из N (n<N) системный вызов возвращает n байт, а остальное сохраняет до следующего чтения;

2) при чтении n байт из N (n<N) (n>N) возвращается N и происходит обработка прерывания;

3) если канал пуст, то возвращается 0 байт, если канал открыт на запись, то процесс будет заблокирован до того момента, пока не появится хотя бы 1 байт;

4) если в канал пишут несколько процессов, то информация не смешивается (это достигается путем использования tag’ов);

5) при записи, когда FIFO полон, процесс блокируется до тех пор, пока не появится место для записи.

При использовании FIFO не гарантируется атомарность операций.

При записи FIFO работает как клиент-серверное приложение.

Функции сервера:

- создание FIFO (mknode());

- открытие FIFO на чтение;

- чтение сообщений;

- закрытие FIFO.

Функции клиента:

- открытие FIFO на запись;

- запись сообщений для сервера в FIFO;

- удаление FIFO.

#include <stdio.h>, <stdlib.h>, <unistd.h>, <sys/types.h>, <sys/stat.h>, <fcntl.h>

void server(int input, int output){

char buf[256]; int num=10; int count=0;

char \* FileName = "input.txt";

if (write(output,FileName,strlen(FileName))<0)

ОШИБКА ЗАПИСИ В FIFO

num = read(input,buf,256);

if( num <0) ОШИБКА ЧТЕНИЯ

printf("SERVER has read %d bytes\n",num);

printf("Server has recieved data: %s\n",buf);

}

int main (void){

int res = mknod ("output",S\_IFIFO | 0666,0);

if (res<0) ОШИБКА СОЗДАНИЯ FIFO

res = mknod ("input",S\_IFIFO | 0666,0);

if (res<0) ОШИБКА СОЗДАНИЯ FIFO

int output,input;

output = open("output",O\_WRONLY);

if (output) OK else НЕ ОК

sleep(1);

input = open("input",O\_RDONLY);

if (input) ОК else НЕ ОК

server(input,output);

close(output); close(input);

if (unlink("input")<0) ОШИБКА

if (unlink("output")<0) ОШИБКА

}

void client(int input, int output){

char buf[256]; char FileName[64];

int num = 10; int FileID;

num = read (input,FileName,64);

printf("recieved file name: %s\n",FileName);

FileID = fopen(FileName,"r");

if (FileID<0) ОШИБКА

int res = fscanf(FileID,"%s",buf);

fclose(FileID);

write(output,buf,strlen(buf));

}

int main (void){

int input, output;

if ((input = open("output",O\_RDONLY))<0) ОШИБКА

if ((output = open("input",O\_WRONLY))<0) ОШИБКА

client(input,output);

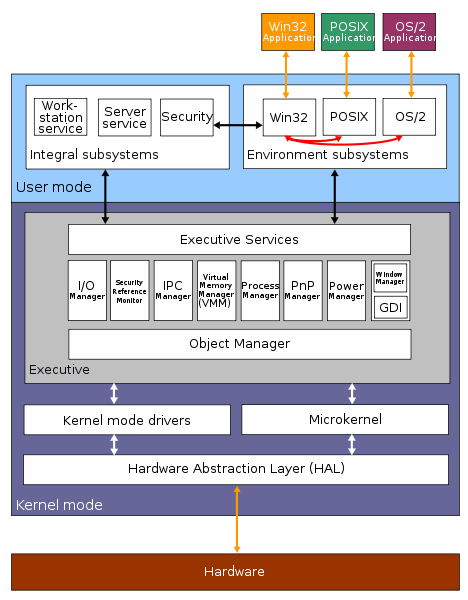
close(input); close(output); return 0;

}

1. Анализ архитектуры и моделей функционирования ОС линейки Windows.

**Архитектура Windows NT** имеет модульную структуру и состоит из двух основных уровней — компоненты, работающие в режиме пользователя и компоненты режима ядра. Программы и подсистемы, работающие в режиме пользователя имееют ограничения на доступ к системным ресурсам. Режим ядра имеет неограниченный доступ к системной памяти и внешним устройствам. Ядро системы NT называют гибридным ядром или макроядром. Архитектура включает в себя само ядро, уровень аппаратных абстракций (HAL), драйверы и ряд служб (Executives), которые работают в режиме ядра (Kernel-mode drivers) или в пользовательском режиме (User-mode drivers).

Пользовательский режим Windows NT состоит из подсистем, передающих запросы ввода/вывода соответствующему драйверу режима ядра посредством менеджера ввода/вывода. Есть две подсистемы на уровне пользователя: подсистема окружения (запускает приложения, написанные для разных операционных систем) и интегрированная подсистема (управляет особыми системными функциями от имени подсистемы окружения). Режим ядра имеет полный доступ к аппаратной части и системным ресурсам компьютера. И также предотвращает доступ к критическим зонам системы со стороны пользовательских служб и приложений.

[](http://upload.wikimedia.org/wikipedia/commons/5/5d/Windows_2000_architecture.svg)

1. Взаимодействие обработчиков событий в ОС:при наличии аппаратных событий, внутренних прерываниях, исключениях

Обработка прерываний. Последов-ть прог и апп ср-в для обработки.

1Прерывание от таймера – вызывает смену задач в мультизадачной ОС;

2Управление вводом/выводом, потоками данных;

3Системные вызовы обеспечиваются прерываниями и позволяют менять пользовательский режим на привилегированный.

Прерывания:

- внутренние – генерируются процессором при возникновении аварий в результате выполнения некоторых инструкций, ошибки четности;

- аппаратные (внешние) – прерывания от ПУ, контроллера прерываний:

\*Маскируемые–генерируются контроллером прерываний по запросу ПУ;

\*Немаскируемые (ошибка обращения к памяти, ошибка сети и т.д.) - формируются системной платой.

- программные.

Классификация прерываний зависит не только от процессора, но и от другой аппаратуры (наличие/отсутствие контроллера прерываний).

Программные прерывания оптимизированные процедуры ОС, вызываемые пользователем или системными программами. Эти модули используются много и часто. Для обработки прерывания необходимо сохранение в стеке значения регистра состояния процессора, а после выхода из прерывания – его восстановление. Программное прерывание не является истинным прерыванием и возникает при вызове соответствующей команды (int – Intel, trap – Motorola…). Команда прерывания зарезервирована в системе команд процессора.

Идея использования программных прерываний:

- использование компактного кода;

- переход из пользовательского режима в привилегированный.

Программные прерывания используются для выполнения ограниченного числа вызовов функций ядра ОС, которые называются системными вызовами.

Приоритеты: относительный и абсолютный.

Аппаратные прерывания обрабатываются драйверами соответствующих устройств.

Исключения (внутренние прерывания) обрабатываются специальными модулями ядра.

Программные прерывания обрабатываются модулями ОС и соответствующими системными вызовами.

Аппаратная поддержка прерываний.

2 способа реализации прерываний на шине:

- векторный;

- голосование - динамический приоритет.

В обоих случая передается информация о приоритетах. С каждым уровнем прерываний может быть связано несколько устройств и несколько обработчиков. При векторном способе передаются еще и адреса начала программ обработки прерываний. При опросе производится вызов всех обработчиков прерываний данного уровня приоритета, после чего производится ожидание подтверждения тем обработчиком, который соответствует реальному устройству.

Характеристики

Быстродействие: векторный способ работает быстрее при наличии на шине нескольких устройств.

Гибкость: в зависимости от аппаратных платформ.

В реальности используется смешанный способ: соблюдается векторный способ по отношению к процессору и способ голосования по отношению к шине.

Для физической реализуемости такой структуры необходим посредник, который хранил бы адреса подпрограмм обработки прерываний – это контроллер прерываний, может также присутствовать и контроллер ПДП. Каскадное соединение позволяет использовать много устройств. Шины, использующие опрашивание – ISA, PCI и др. Векторная шина VME – позволяет процессору общаться с ПУ без посредников.

Программная поддержка – прерывания позволяют процессору быстро реагировать на асинхронные по отношению к вычислительному процессу события.

Для упорядочения событий в некоторых ОС применяется механизм приоритетных очередей – в соответствии с приоритетами источников. Для обеспечения работы данной структуры используется специальный программный модуль – диспетчер прерываний. При возникновении прерывания диспетчер прерываний вызывается первым, временно запрещаются все прерывания, диспетчер выясняет источник возникновения прерывания, сравнивает его приоритет с приоритетом текущего прерывания и принимает решение о прерывании или ожидании его завершения работы текущего прерывания.

По отношению к любому потоку прерывания всегда обладают более высоким приоритетом. Таким образом, используется двухуровневая система прерываний:

1-ый уровень – уровень диспетчера прерываний;

2-ой уровень – уровень диспетчера потоков.

В качестве стратегии планирования используется алгоритм квантования. Также используются различные схемы приоритетов и различные схемы квантования.

1. Резидентные программы. Пример реализации.

Резидентнтная программа — это программа, которая завершает выполнение, не освобождая занятую память полностью. Этот термин используется применительно к ОС DOS. Обычно в памяти оставляют фрагмент программы, содержащей процедурю обработки прерываний; в нерезидентной части программы выполняется настройка векторов на процедуры обработки из резидентной части программы.

После установки векторов выполнение программы как самостоятельного процесса заканчиваетяся — но таким образом, что код и данные остаются в памяти. Отсюда — принятое в англоязычной технической литературе название Terminate but stay Resident. Или, сокращенно TSR. Оставленая в памяти процкдура обработки прерывания в момент ее вызова считается частью *текущего* , т.е. прерванного процесса. В результате, обращение к функции DOS 04c из резидентной процедуры приводит к завершению *прерванной* программы.

Завершение программы с сохранением ее фрагмента в памяти выполняется вызовом int 027h , вместо обычного int 020h. В регистре dx при вызове int 027 должен быть задан адрес нижней границы области освобождаемой памяти. Фрагмент от смещения 0 до смещения, указанного в dx , сохраняется.

Код программы для обработки исключения 4:

jmp start:

msg db 13,10,'?-Trap (into)$'

new4:

push ax,dx,ds

mov ax,cs

mov ds,ax

lea dx,msg

mov ah,9

int 021

#if stop

mov ax,04c01

int 021

#else

pop ds,dx,ax

iret

#endif

start:

mov ax, 02504

lea dx, new4

int 021

mov dx, start

int 027

1. Структура ядра ОС с иерархической архитектурой. Алгоритмы выполнения запросов прикладного уровня

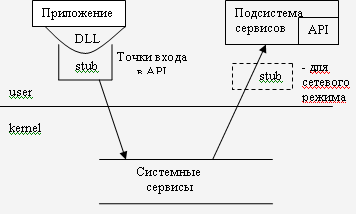
В монолитных и иерархических системах механизм реализации сервиса: запрос системного сервиса приводит к переходу в режим ядра, выполняется в режиме ядра, а затем выход обратно.

В случае клиент-серверной модели:

1) Совместное использование сегментов dll (Windows X, OS/2).

Различные приложения разделяют АП различных динамических библиотек, при этом в Windows делается однократное копирование библиотек в разделяемую память (доступ от всех), а в OS/2 – многократное копирование библиотек в каждый процесс отдельно (в АП процесса). Каждый процесс может модифицировать данные, используемые всеми приложениями. Обращение производится за счет обычных вызовов подпрограмм.

2) с защитой памяти подсистем (Windows NT).



stub – приводит запрос к формату для передачи в подсистему.

Модель защиты (клиент-серверная):

«+» 1. Защита глобальных структур данных

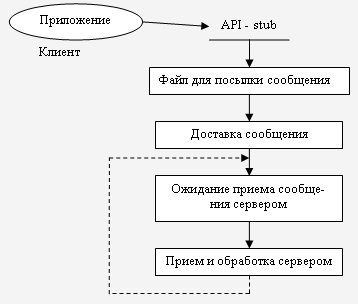
2. Автономность сервисов

3. Сервера для приложения работают в пользовательском режиме, следовательно, не могут влиять на ядро и не могут вызывать внутренние функции ОС. Единственный доступ к ядру – вызов системных сервисов.

4. Обеспечивает разделение сервера – ОС.

5. Возможность одновременной работы любого количества серверов поддерживающих средства API. Повышение степени параллелизма.

1. Потеря производительности.

Для получения и обработки сообщения сервером необходимо: 1)переключить контекст 2)сохранить предыдущий 3)выбрать для исполнения поток сервера 4)загрузить его контекст 5)выполнить функции API (используя поток сервера) 6)сохранить контекст потока-сервера 7)загрузить обратно поток клиентского потока 8)обработка результата API вызова

1. Синхронный и асинхронный ввод/вывод в NT, формат IRP, проекционный ввод/вывод.

Требования, которые предъявляются для разработчиков драйверов под NT:

* Драйверы пишутся на языках высокого уровня для обеспечения переносимости.
* Операция В/В управляется IRP-пакетами. IRP используются многократно по слоям. Система динамически назначает драйверы для управления дополнительными новыми.
* Драйверы должны синхронизировать доступ к глобальным данным
* Должны позволять вытеснять потоками более высокого приоритета
* Должны быть прерываемы другими потоками
* Код драйвера должен быть способен выполняться на нескольких процессорах.
* Драйверы должны восстанавливаться после сбоя и выполнять нереализованные операции (в NT 4.0 не реализовано)

Драйверы имеют унифицированную модель интерфейса => диспетчер В/В не видит их структуру и детали реализации. При взаимодействии различных драйверов диспетчер играет роль посредника.

Драйверы могут быть:

* Однослойные (для последовательного порта и др.)
* Многослойные (для ЗУ большой емкости)

Пример многослойного драйвера – при работе через SCSI-интерфейс драйвер дисков передает запрос на драйвер SCSI, который реализует обработку. Запрос к дискам реализует драйвер SCSI.

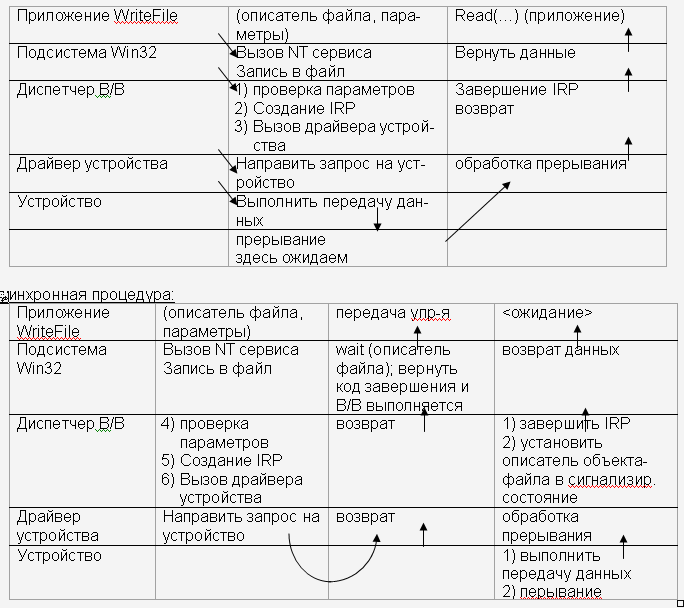
Передача может быть:

* Асинхронной – более быстродействующий вариант и экономичный в плане ресурсов
* Синхронный – проще в плане реализации

Win32 автоматически выбирает синхронный/асинхронный вариант взаимодействия.

Поскольку асинхронный способ позволяет существенно увеличить производительность пользовательских приложений, то по умолчанию для 1/3 сервисов установлен асинхронный режим. Тем не менее поток может сам решать каким способом передавать данные.

Синхронная процедура:



**Формат пакета IRP**



Каждому потоку соответствует своя очередь IRP-пакетов. Это позволяет идентифицировать IRP-пакеты. При завершении потока очередь удаляется. Управление очередью в соответствии с составом потока

1. Причины генерации сигналов.Примеры.Диспозиция. Пример кода C++ запуска.деление на 0.

si\_code – содержит код причины генерации сигнала. Для большинства сигналов si\_code принимает следующие значения.

* SI\_KERNEL – сигнал сгенерирован ядром.
* SI\_QUEUE – пользовательское приложение вызвало системный вызов sigqueue() для отправки сигнала.
* SI\_TKILL – пользовательское приложение использовало системный вызов tkill().
* SI\_USER – пользовательское приложение вызвало системный вызов kill() для отправки сигнала.

Обработчик.

#include <stdio.h>  
#include <fenv.h>  
#include <signal.h>  
#include <stdlib.h>  
  
void fpehandler(int sig\_num)  
{  
signal(SIGFPE, fpehandler);  
printf("SIGFPE: floating point exception occured, exiting.\n");  
exit(1);  
}  
  
int main()  
{  
double x, y;  
  
int feenableexcept();  
feenableexcept(FE\_ALL\_EXCEPT);  
signal(SIGFPE, fpehandler);  
  
x= 0.0;  
y= 0.0;  
  
x= x/y;  
}

1. Многозадачность (мультипрограммирование). Реализация в различных ОС.

**Многозадачность в ОС:**

Вытесняющая – управление полностью централизованное, им занимается только ОС, централизовано. (Большинство современных ОС: Windows NT, UNIX, OS/2).

Невытесняющая – управлением могут заниматься и программы, т.е. в приложении заложена возможность передачи управления (ОС для определенных задач, например файловый сервер NetWare, Windows 3.x).

1. Трудоемкость сигналов

Ресурсоёмко потому что требуется много переключений из пользовательского режима в режим ядра из-за системных вызовов (issig, sendsig).

Доставка сигнала происходит после того, как ядро от имени процесса вызывает системную процедуру issig() которая проверяет, существуют лиожидающие доставки сигналы, адресованные данному процессу. Функция issig вызывается ядром в трех случаях:

1. Непосредственно перед возвращением из режима ядра в режим задачи после обработки системного вызова или прерывания.

2. Непосредственно перед переходом процесса в состояние сна с приори!

тетом, допускающим прерывание сигналом.

3. Сразу же после пробуждения после сна с приоритетом, допускающим

прерывание сигналом.

Если процедура issig() обнаруживает ожидающие доставки сигналы, ядро вызывает функцию доставки сигнала, которая выполняет действия по умолчанию или вызывает специальную функцию sendsig() запускающую обработчик сигнала, зарегистрированный процессом. Функция

возвращает процесс в режим задачи, передает управление обработчику сигнала, а затем восстанавливает контекст процесса для продолжения прерванного сигналом выполнения.

В качестве бонуса можно упоминуть про ненадежность (по временному критерию) сигналов. Сигнал будет доставлен только после выбора процесса планировщиком и выделения ему ресурсов, что нивилирует асинхронный характер сигнала.

1. Реализация сообщений в различных ОС

**Сообщения.**

Сообщения обслуживаются ОС, являются разделяемым системным ресурсом, являются частью ядра ОС, образуют очереди, которые образуют список. Сама очередь имеет уникальное имя – идентификатор.

Процессы могут читать сообщения из различных очередей.

Атрибуты сообщений:

- тип сообщения (мультиплексный/немультиплексный);

- длина сообщений (байт);

- сами данные (м.б. структурированы).

Хранится очередь в виде однонаправленного односвязного списка. Для каждой очереди ядро создает свою структуру.

msgid\_ds ----------------------------------------🡪 ipc\_perm

msg\_perm (permission – права доступа)

msg\_cbytes (число байт)

msg\_num (число сообщений в очереди)

msg\_first (первый объект в списке)

msg\_last (последний в списке)

Функции:

msg\_snd() – отправка сообщения;

msg\_rcv() – прием сообщения;

msg\_get()– прием сообщения;

msg\_ctl()– прием сообщения.

|  |
| --- |
|  |

|  |
| --- |
| 1. Функцион комп-ты сетев. ОС. Их взаимод-ие и способы реализации.   На рис. 2.1 показаны основные функциональные компоненты сетевой ОС:   * средства управления локальными ресурсами компьютера реализуют все функции ОС автономного компьютера (распределение оперативной памяти между процессами, планирование и диспетчеризацию процессов, управление процессорами в мультипроцессорных машинах, управление внешней памятью, интерфейс с пользователем и т. д.); * сетевые средства, в свою очередь, можно разделить на три компонента: * средства предоставления локальных ресурсов и услуг в общее пользование — серверная часть ОС; * средства запроса доступа к удаленным ресурсам и услугам — клиентская часть ОС; * транспортные средства ОС, которые совместно с коммуникационной системой обеспечивают передачу сообщений между компьютерами сети.   http://education.aspu.ru/olifer_img/gl2-1.jpg  Рис. 2.1. Функциональные компоненты сетевой ОС  Упрощенно работа сетевой .ОС происходит следующим образом. Предположим, что' пользователь компьютера А решил разместить свой файл на диске другого компьютера сети — компьютера В. Для этого он набирает на клавиатуре соответствующую команду и нажимает клавишу Enter. Программный модуль ОС, отвечающий за интерфейс с пользователем, принимает эту команду и передает ее клиентской части ОС компьютера А.  Клиентская часть ОС не может получить непосредственный доступ к ресурсам другого компьютера — в данном случае к дискам и файлам компьютера В. Она может только «попросить» об этом серверную часть ОС, работающую на том компьютере, которому принадлежат эти ресурсы. Эти «просьбы» выражаются в виде сообщений, передаваемых по сети. Сообщения могут содержать не только команды на выполнение некоторых действий, но и собственно данные, например содержимое некоторого файла.  Управляют передачей сообщений между клиентской и серверными частями по коммуникационной системе сети транспортные средства ОС. Эти средства выполняют такие функции, как формирование сообщений, разбиение сообщения на части (пакеты, кадры), преобразование имен компьютеров в числовые адреса, организацию надежной доставки сообщений, определение маршрута в сложной сети и т. д. и т. п. Правила взаимодействия компьютеров при передаче сообщений по сети фиксируются в коммуникационных протоколах, таких как Ethernet, Token Ring, IP, IPX и пр. Чтобы два компьютера смогли обмениваться сообщениями по сети, транспортные средства их ОС должны поддерживать некоторый общий набор коммуникационных протоколов. Коммуникационные протоколы переносят сообщения клиентских и серверных частей ОС по сети, не вникая в их содержание.  На стороне компьютера В, на диске которого пользователь хочет разместить свой файл, должна работать серверная часть ОС, постоянно ожидающая прихода запросов из сети на удаленный доступ к ресурсам этого компьютера. Серверная часть, приняв запрос из сети, обращается к локальному диску и записывает в один из его каталогов указанный файл. Конечно, для выполнения этих действий требуется не одно, а целая серия сообщений, переносящих между компьютерами команды ОС и части передаваемого файла.  Очень удобной и полезной функцией клиентской части ОС является способность отличить запрос к удаленному файлу от запроса к локальному файлу. Если клиентская часть ОС умеет это делать, то приложения не должны заботиться о том, с локальным или удаленным файлом они работают, — клиентская программа сама распознает и перенаправляет (redirect) запрос к удаленной машине. Отсюда и название, часто используемое для клиентской части сетевой ОС, — редиректор. Иногда функции распознавания выделяются в отдельный программный модуль, в этом случае редиректором называют не всю клиентскую часть, а только этот модуль.  Клиентские части сетевых ОС выполняют также преобразование форматов запросов к ресурсам. Они принимают запросы от приложений на доступ к сетевым ресурсам в локальной форме, то есть в форме, принятой в локальной части ОС. В сеть же запрос передается клиентской частью в другой форме, соответствующей требованиям серверной части ОС, работающей на компьютере, где расположен требуемый ресурс. Клиентская часть также осуществляет прием ответов от серверной части и преобразование их в локальный формат, так что для приложения выполнение локальных и удаленных запросов неразличимо.   1. взаимосвязь ф-ий exec(). Алгоритм загр-ки исполняемого файла   Этап exec()  выполняет 2 функции:  1.Заполняет АП из исполняемого файла 2.Запуск на исполнение  С точки зрения полноты параметров наиболее близка функция execve().  Наследуются: PID , PPID родителя идентификатор пользователя и группы , эффективный идентификатор пользователя и группы (EUID , EGID) без флага SUID , ограничения на процесс , текущий и корневой каталого , маска создаваемых файлов , упр. терминал , файлов. дескриптор.  Сист. вызов обычно содержит след. аргументы exec():  1.Имя исполняемого файла (программы)  2.Набор аргументов  3.Список переменных окружения  В большинстве случаев не созд. АП , а происходит замена АП как есть после вызова fork().(зависит от версии UNIX)  Все возвращаемые значения типа int  В большинстве обрабатываемых системных вызовах , если возвращается -1 то это ошибка.   1. Способы низкоуровнего отключ-ия прерываний. |
|  |

Для отключения прерываний на критических участках кода можно использовать несколько возможностей: 1) замаскировать необходимые прерывания в регистре iMrx контроллера прерываний

2)снять флаг разрешения прерываний процессора(флаг I).

1.Для отключения отдельного устройства в регистр масок контроллера прерываний необходимо записать значения, позволяющее замаскировать порт к которуму будет подключено соответствующее устройство. Для этого нужно обратиться к порту PORTy контроллера прерываний и записать в него в разряд iMrx единицу для маскирования соответствующего прерывания от устройства.

Mov dx, porty//port

mov ax ,08h//маска

out dx, ax//посылаем

2.Для снятия/установки флага I в ассемблере существуют команды cli/sti. Они позволяют заблокировать все внешние прерывания от устройств. Но запрос через дополнительный вход процессора NMI(Non-Maskable Interrupt) не блокируется и обслуживается сразу независимо от флага I. Ко входу NMI обычно подключены сигналы фатальных системных ошибок, например сгнал ошибки четности или обращения к памяти.

1. Задача планирования и диспетчиризации.Типы планировщиков.Планирование в системах реального времени.Разделение времени.

Переход от одного проц. к другому выполняется в результате планирования потоков. Фактически осуществляется планирование потоков, а не процессов.

При планировании решается 2 задачи:

1) определение момента t смены актив потока

2) выбор потока из очереди готовых (как правило тех что в ОЗУ)

Обе задачи решается программно 2 основ классами алгоритмов:

- на основе квантования

- на основу приоритетов

Сущ стат и динам планирование. В больш-ве ОС использ динам планирование – online – планирование, те планир исходя из текущего сост системы. Такое планир сложнее но они более гибки. В стат планировках предполагается наличие расписания. Такое планир повышает быстродействие. Исп для real-time ОС, спец системой стат планир offline. Реализация планирования – диспетчеризация. Вып как в стат так и в динам режиме. В отличие от планирования диспетчеризация реализуется в большинстве случаев аппаратно.

**Алгоритмы планирования.**

1. На осн квантования – передача происходит либо по завершении процесса либо при переходе процесса в ожидание, либо при завершении времени квантования, либо при выходе с ошибкой. Квантование исп во всех система раздел времени. Величина кванта м.б. фикс, либо динамич. Измен времени кванта может зависеть от стадии в жизнен цикле процесса и связ с оптимиз

Приоритеты : абсолютные, относительные

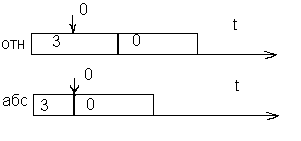


Схема выбора всегда одинаковая – выбир процесс с наивысшим приоритетом. Но если во время вып процесс приходит процесс с более высоким приоритетом то при реализации абсолютного приоритета текущий процесс прерывается и начинается вып нового; в относит –

выполнение пришедшего процесса начинается только после заверш текущего.

Вытесняющая и невытесняющая многозадачность

-Невытесн – способ планирования при кот процесс выполняется до тех пор пока собственной инициативе не передаст управление планировщику ОС

-Вытесн – способ при кот решение о переключении процесса принадлежит планировщику , а не активной задаче.

Онсовные различия – степень централизации механизма планирования. При невытесн схеме механизм планир распред между прикладной задачей и ОС. Такой механизм более сложный. При вытесняющей – реализация планирования полностью является прерогативой ОС.

1. убить тред с клавиатуры (программа)

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#include <signal.h>

pthread\_t t1, t2;

void int\_handler(int signum)

{

printf("Caught SIGUSR1\n");

pthread\_exit(NULL);

}

void \* thread\_function1(void \* arg)

{

char c[100];

int time = 0;

while (1)

{

fgets (c, sizeof(c),stdin);

if (c[0]=='q')

{

printf("Killing t2\n");

pthread\_kill(t2, SIGUSR1);

}

}

}

void \* thread\_function2(void \* arg)

{

int time = 0;

while (1)

{

printf("Thread 2: %d\n", time++);

sleep(1);

}

}

int main()

{

signal(SIGUSR1, int\_handler);

pthread\_create(&t1, NULL, thread\_function1, NULL);

pthread\_create(&t2, NULL, thread\_function2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

return 0;

}

1. Совместимость и множественные прикладные среды.Реализация множественных прикладных сред.

**Совместимость**

Совместимость ОС – свойство, позволяющее выполнять приложения, написанные для других ОС.

* Двоичная совместимость – исполняемый файл. Определяется архитектурой процессора, совпадение API. Внутренняя структура файла должна соответствовать структуре, используемой в данной ОС. При несовпадении структуры необходима эмуляция двоичного кода. Эмуляция используется в системах кросс-разработки.
* На уровне исходных кодов – совместимость на уровне компиляторов, совместимость библиотек, системных вызовов.

Эмуляция двоичного кода используется в системах кросс-разработки – системах, предназначенных для разработки программ в 2х машинной конфигурации.

Состав системы кросс-разработки:

* Средства редактирования
* Средства компиляции
* Средства отладки

Все это находится на инструментальной машине, а в готовом виде передается на целевую. Использование кросс-средств: системы программирования МК (Intel, Atmel и др.). ОС WinCE, PalmOS. Такие ОС включают в себя набор компиляторов и ассемблеров на инструментальной машине под ее ОС, библиотеки, выполняющие большую часть функций целевой ОС, средства отладки.

**Прикладные среды. Способы реализации**

Альтернатива эмуляции – прикладные среды. В состав каждой прикладной среды входит набор функций интерфейса API. Прикладные среды имитируют обращение к библиотечным функциям инородной среды. На самом деле обращение происходит к внутренним библиотекам. Это называется трансляцией библиотек. Чтобы программа, написанная под одной ОС работала под другой, необходимо обеспечить бесконфликтное взаимодействие способов управления процессами в разных ОС.

Способы реализации прикладных сред

1. В данном случае прикладная программная среда реализована как верхний слой ядра родной ОС.

2.



Отдельный слой (API). Прикладная программная среда на основе нескольких равноправных API.

3. Реализуется по микроядерному принципу (Windows NT).

1. Процесс сосстояния жизненный цикл многозадачных ОС.Утилиты статуса,информационной структуры.
2. Критерии показателей ОС различных классов.(1 лекция)

**Классификация ОС.**

1. С точки зрения алгоритмов управления процессором.

Однопользовательские/многопользовательские; Однозадачные/многозадачные; С поддержкой многопоточности / без поддержки многопоточности; Однопроцессорные/многопроцессорные.

2. С точки зрения максимальной производительности.

Критерии оценки:

Если мы рассматриваем в качестве критерия максимальное количество задач в единицу времени, имеем в виду алгоритмические системы с пакетной обработкой.

Мультипрограммная смесь составляется таким образом, чтобы задачи были разнородными: вычислительные, ввода/вывода (чтобы обеспечить равномерную загрузки всех средств).

3. Другие критерии:

- Удобство работы пользователя – возможность пользователя работать сразу с несколькими программами одновременно. Используется системное разделение времени.

- Реактивность – время, прошедшее от запуска программы до появления управляющего воздействия (для систем реального времени).

4. Мультипроцессированные ОС.

ОС может быть однопроцессорной или многопроцессорной.

Симметричные ВС:

В ВС входит набор однородных процессоров (одинаковая конструкция). Такую систему можно наращивать по вертикали до 4…8 процессоров.

Ассиметричные ВС:

Также в ВС может быть набор неоднородных процессоров – каждый со своими свойствами. Для вычислительных систем это лучше, но для ОС сложнее.

Симметричные ОС – только для симметричной аппаратуры. Все функции ОС распределены между процессорами. Идеальный вариант – динамическое распределение. Такие системы из-за своей сложности часто ненадежны.

Ассиметричные ОС – для любой аппаратуры. Один из вариантов ОС, ассиметричной по мультипроцессированию – когда один процессор главный, и он распределяет задачи между остальными. Это надежно.

5. Многозадачность в ОС:

Вытесняющая – управление полностью централизованное, им занимается только ОС, централизовано. (Большинство современных ОС: Windows NT, UNIX, OS/2).

Невытесняющая – управлением могут заниматься и программы, т.е. в приложении заложена возможность передачи управления (ОС для определенных задач, например файловый сервер NetWare, Windows 3.x).

1. Структуры данных ФС UNIX подобных ОС(чето там про vnod)Назначение и основной состав.

1) 2 типа объектов для использования(во всех ОС):

- файлы (все данные - файлы)

- процессы 2) в UNIX доступ к ПУ через файлы 3) при запуске программы ядро запускает соответствующий файл создает образ процесса и передает процессу управление 4) взаимодействие между файлами и процессами возникающие при функционировании процессов

5) функциональность ОС определяется выполнением соответствующих процессов Программа – совокупность файлов, подвергнутых компоновке и выполнению. Чтобы запустить программу нужно создать среду выполнения программы (окружение), это: ресурсы памяти, доступ к системе к ПУ и тд. Это окружение процесс. Процесс – совокупность данных ядра системы, необходимых для описания образа программы в памяти и для управления ее выполнением.

При запуске исполняемого файла может быть создано несколько процессов, но минимум один Типы процессов в UNIX системах:

1) системные процессы (основные) 2) демоны 3) прикладные процессы

1)– часть ядра всегда в ОЗУ не имея исполняемых файлов, запускаются при инициализации ядра и выполняют инструкции ядра; доступ фактически ко всем данным (диспетчеры кеша, памяти ядра и др)

init – прародитель всех остальных процессов, запускается (в etc/init ) первым (имеет свой исполнительный файл) (далее - loginshell)

init не является частью ядра, запуск из исполнительного файла, функционирует до выгрузки системы

2)– не интерактивные процессы, запускаемые обычным способом (загрузка кода в ОЗУ исполняемого файла), как правило, выполняются в фоновом режиме. Поддерживает основные системы (терминальные и тд)

не связанная ни с одним пользовательским сеансом, пользователем не изменяется, работают в режиме ожидания (работа с подсистемами) Все остальное - (3).В рамках пользовательского сеанса; фоновый режим; время жизни ограничено временем пользовательского сеанса

Атрибуты процессов: - PID – идентификатор процесса. Уникальный номер, по которому ядро различает тот или иной процесс. Присваивается по возрастанию и по мере освобождения поцессов

- идентификатор родитель – PPID - NiceNumber – приоритет процесса; - относительный и - приоритет выполнения(текущий) -TTY – соответствие используемому терминалу - RID – реальный идентификатор пользователя, запустившего данный процесс. Сущ эффективный идентификатор (SUID – расширенные права, направлено на запуск чужих программ).

1. Реализация сервисов по запросу от приложений в ОС с различными архитектурами. Два подхода к реализации клиент-серверной модели

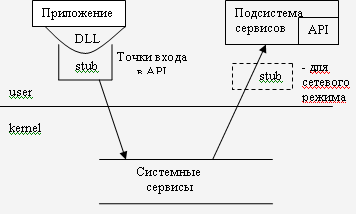
В монолитных и иерархических системах механизм реализации сервиса: запрос системного сервиса приводит к переходу в режим ядра, выполняется в режиме ядра, а затем выход обратно.

В случае клиент-серверной модели:

1) Совместное использование сегментов dll (Windows X, OS/2).

Различные приложения разделяют АП различных динамических библиотек, при этом в Windows делается однократное копирование библиотек в разделяемую память (доступ от всех), а в OS/2 – многократное копирование библиотек в каждый процесс отдельно (в АП процесса). Каждый процесс может модифицировать данные, используемые всеми приложениями. Обращение производится за счет обычных вызовов подпрограмм.

2) с защитой памяти подсистем (Windows NT).



stub – приводит запрос к формату для передачи в подсистему.

Модель защиты (клиент-серверная):

«+» 1. Защита глобальных структур данных

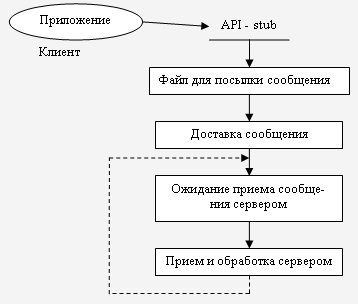
2. Автономность сервисов

3. Сервера для приложения работают в пользовательском режиме, следовательно, не могут влиять на ядро и не могут вызывать внутренние функции ОС. Единственный доступ к ядру – вызов системных сервисов.

4. Обеспечивает разделение сервера – ОС.

5. Возможность одновременной работы любого количества серверов поддерживающих средства API. Повышение степени параллелизма.

«-» 1. Потеря производительности.

Для получения и обработки сообщения сервером необходимо: 1)переключить контекст 2)сохранить предыдущий 3)выбрать для исполнения поток сервера 4)загрузить его контекст 5)выполнить функции API (используя поток сервера) 6)сохранить контекст потока-сервера 7)загрузить обратно поток клиентского потока 8)обработка результата API вызова

1. Индексный дескриптор, идентификатор файла, ссылки.... назначение...

индексный дескриптор — inode. Содержит информацию о файле , необходимую для обработки данных, т.е. метаданные файла. Используется файловой системой.

файловый дескриптор — индекс в таблице файловых дескрипторов процесса. Представляет собой неотрицательное число, возвращаемое системными вызовами, такими как open(), pipe(). Файловый дискриптор связан сполями u\_ofile и u\_pofile структуры user и таким образом обеспечивает доступ к соответсвующему элементу файловой таблицы(структуре данных file).

Файловые ссылки — 2 вида. Жесткие ссылки и символьные. Жесткие ссылаются на vnode. Символьные — это путь к жесткой ссылке. Для связи vnode и содержимого каталогов ФС.Связь имя файла ->inode.

1. Потоки в NT, сервисы, приоритеты, атрибуты....

В разных ОС процессы реализуются по-разному. Эти различия заключаются в том, какими структурами данных представлены процессы, как они именуются, какими способами защищены друг от друга и какие отношения существуют между ними. Процессы Windows NT имеют следующие характерные свойства:

Процессы Windows NT реализованы в форме объектов, и доступ к ним осуществляется посредством службы объектов.

Процесс Windows NT имеет многонитевую организацию.

Как объекты-процессы, так и объекты-нити имеют встроенные средства синхронизации.

Менеджер процессов Windows NT не поддерживает между процессами отношений типа "родитель-потомок".

В любой системе понятие "процесс" включает следующее:

исполняемый код,

собственное адресное пространство, которое представляет собой совокупность виртуальных адресов, которые может использовать процесс,

ресурсы системы, такие как файлы, семафоры и т.п., которые назначены процессу операционной системой.

хотя бы одну выполняемую нить.

Адресное пространство каждого процесса защищено от вмешательства в него любого другого процесса. Это обеспечивается механизмами виртуальной памяти. Операционная система, конечно, тоже защищена от прикладных процессов. Чтобы выполнить какую-либо процедуру ОС или прочитать что-либо из ее области памяти, нить должна выполняться в режиме ядра. Пользовательские процессы получают доступ к функциям ядра посредством системных вызовов. В пользовательском режиме выполняются не только прикладные программы, но и защищенные подсистемы Windows NT.

В Windows NT процесс - это просто объект, создаваемый и уничтожаемый менеджером объектов. Объект-процесс, как и другие объекты, содержит заголовок, который создает и инициализирует менеджер объектов. Менеджер процессов определяет атрибуты, хранимые в теле объекта-процесса, а также обеспечивает системный сервис, который восстанавливает и изменяет эти атрибуты.

В число атрибутов тела объекта-процесса входят:

Идентификатор процесса - уникальное значение, которое идентифицирует процесс в рамках операционной системы.

Токен доступа - исполняемый объект, содержащий информацию о безопасности.

Базовый приоритет - основа для исполнительного приоритета нитей процесса.

Процессорная совместимость - набор процессоров, на которых могут выполняться нити процесса.

Предельные значения квот - максимальное количество страничной и нестраничной системной памяти, дискового пространства, предназначенного для выгрузки страниц, процессорного времени - которые могут быть использованы процессами пользователя.

Время исполнения - общее количество времени, в течение которого выполняются все нити процесса.

Напомним, что нить является выполняемой единицей, которая располагается в адресном пространстве процесса и использует ресурсы, выделенные процессу. Подобно процессу нить в Windows NT реализована в форме объекта и управляется менеджером объектов.

Объект-нить имеет следующие атрибуты тела:

Идентификатор клиента - уникальное значение, которое идентифицирует нить при ее обращении к серверу.

Контекст нити - информация, которая необходима ОС для того, чтобы продолжить выполнение прерванной нити. Контекст нити содержит текущее состояние регистров, стеков и индивидуальной области памяти, которая используется подсистемами и библиотеками.

Динамический приоритет - значение приоритета нити в данный момент.

Базовый приоритет - нижний предел динамического приоритета нити.

Процессорная совместимость нитей - перечень типов процессоров, на которых может выполняться нить.

Время выполнения нити - суммарное время выполнения нити в пользовательском режиме и в режиме ядра, накопленное за период существования нити.

Состояние предупреждения - флаг, который показывает, что нить должна выполнять вызов асинхронной процедуры.

Счетчик приостановок - текущее количество приостановок выполнения нити.

Кроме перечисленных, имеются и некоторые другие атрибуты.

Как видно из перечня, многие атрибуты объекта-нити аналогичны атрибутам объекта-процесса. Весьма сходны и сервисные функции, которые могут быть выполнены над объектами-процессами и объектами-нитями: создание, открытие, завершение, приостановка, запрос и установка информации, запрос и установка контекста и другие функции.

1. Драйвер, структура (набор процедур). Многослойный, простейший, с прерываниями...

Всякий драйвер – независимый элемент ОС, который может быть динамически загружен или удален без изменения конфигурации ОС.

Драйверы имеют унифицированную модель интерфейса => диспетчер В/В не видит их структуру и детали реализации. При взаимодействии различных драйверов диспетчер играет роль посредника.

Драйверы могут быть:

* Однослойные (для последовательного порта и др.)
* Многослойные (для ЗУ большой емкости)
* с прерываниями

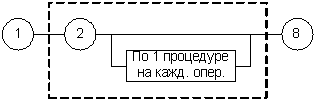
Передача может быть:

* Асинхронной – более быстродействующий вариант и экономичный в плане ресурсов
* Синхронный – проще в плане реализации

**Структура драйвера:**

|  |  |
| --- | --- |
| **Составная часть** | **Назначение процедуры.** |
| Процедура инициализации (1) | Выполняется диспетчером В/В при загрузке ОС. Создаются системные объекты для распознавания устройства и доступа к нему. |
| Набор процедур распределения (2) | Главные функции, предоставляемые драйверами устройствам. Запрос В/В заставляет диспетчер В/В генерировать IRP и обращается к драйверу через процедуру распределения. |
| Процедура запуска В/В  (3) | Драйвер использует эту процедуру для начала передачи данных. Необязательная процедура характерна для физических устройств при передаче больших объемов. |
| ISR  Interrupt  (4) | Прерывание от устройства. На момент выполнения приоритет RPL повышается. Сообразно ISR выполняет часть кода обработчиков. В соответствии с прерыванием эта часть кода может быть с повышенным приоритетом, а часть кода обработки является вторичной и выполняется посредством отлож. вызова DPC (ISR генерир. DPC). |
| Процедура DPC (5) | Процедура обработки прерывания, которая выполняется при более низком, чем у IRQ приоритете. Выполняет большую часть обработки по обслуживанию прерывания. Это инициирует завершение В/В и запуск следующего запроса из очереди на обработку устройствами В/В. |
| Процедура завершения  (6) | Является посредником между драйверами нижнего и верхнего уровня. В ней хранится информация о возможных значениях, сбое или необходимости очистки при инициализации. |
| Процедура отмены В/В (7) | Может быть не одной. Информация об активизации процедуры отмены хранится в IRP. |
| Процедура выгрузки  (8) | Освобождение системных ресурсов, которые используются драйвером. Затем диспетчер В/В удаляет драйвер из памяти. Драйвер может быть загружен/выгружен во время работы. |
| Процедура протоколирования ошибок (9) | Информирование диспетчера В/В об ошибке, который помещает эту информацию в журнал ошибок. |

Минимальный драйвер должен содержать 3 процедуры:



Драйвер устройства:

(1), (2), (3), (4), (5), (6), (8) для многослойного драйвера

1. записи каталогов различных файловых систем

Каталог FFS. Представлен следующей структурой:

d\_ino – номер inode

d\_reclen – длина записи

d\_namlen – длина имени файла

d\_name[] - имя файла

Имя файла имеет переменную длину, дополненную нулями до 4-байтной границы. При удалении имени файла принадлежащая ему запись присоединяется к предыдущей.

В FAT16, как и FAT 12, используются имена 8+3 символа, но в ОС Win32 возможно применение длинных имен, из 255 символов UNICODE (по 2 байта на символ). Первые 8+3 символа хранятся в этом случае также в основной записи каталога а последующие символы – по 13 штук (26 байт) + 6 служебных байт – отдельными порциями за основной записью. Аналогично устроена ФС FAT32.

1. структура сист. ввода-вывода NT, функции драйвера и диспетчера

Часть исполнительной системы, получающая запросы от процессов пользователя либо режима пользователя, либо ядра, и передающая эти запросы устройствам ввода-вывода в преобразованном виде.

Система управляется пакетами. Каждый запрос представляет IRP-пакет (Input Request Package).

IRP позволяет использовать иерарх. настр. системы В/В и драйверов.

IRP – структура данных, управляющая обработкой выполнения операций на каждой стадии их выполнения.

Диспетчер В/В – входит в состав системы В/В, управляет передачей запросов В/В в файловую систему, реализует процедуру общего назначения для разных драйверов.

Функции и этапы работы диспетчера:

* Создание IRP определенного формата
* Передача пакета соответствующему драйверу
* Удаление IRP

Функции драйвера:

* Получение IRP
* Выполнение операции
* Возвращение управления В/В, либо реализация завершения

В NT существуют драйверы файловой системы и драйверы устройств В/В. При любой операции В/В задействованы и драйверы ФС и драйверы устройств. Работают через IRP, контролируются диспетчером. Реализуется общая процедура для различных драйверов. Упрощаются отдельные драйвера и вносится универсальность.



Принцип: виртуальные файлы устройств (как в UNIX) – объектная модель. Работа осуществляется посредством файловых описателей. Описатель ссылается на объект – файл.

Потоки пользовательского режима вызывают базовые сервисы файла объекта (чтение, запись, открытие). Диспетчер динамически преобразует эти запросы в запросы к реальным физическим устройствам (накопителям, сетям и т.д.)

1. вызов подпрограмм и задач.(через шлюз)

Текст программы:

Shluz.c

|  |
| --- |
| #include <stdio.h> |
| main ( argc, argv ) |
| int argc; |
| char \*argv []; |
| { |
| FILE \*fil; |
| int i; |
| char a; |
| sss=fopen(argv[1], “r”); |
| for(i=0;;i++) |
| { |
| If(feof(fil)) |
| { |
| printf(“\n”); |
| break; |
| } |
| fscanf(fil,”%c”,&a); |
| printf(“%c”, a); |
| } |
| fclose(fil); |
| } |

Содержимое файла /root/text:

You couldn’t see this text

Программе-шлюзу и файл, имеют следующие права доступа:

$ ls -l ./root/shluz ./root/text

-rwSrwxr-x 1 root root 4746 Nov 09 22:30 ./root/shluz

-rw------- 1 root root 59 Nov 09 22:35./root/text

Читаем файл программой shluz:

$ ./root/shluz ./root/text

You couldn’t see this text

Результат успешен, так как файл ./root/shluz имеет атрибут SUID.

|  |
| --- |
|  |
| 1. Совместимость ОС. Прикладные среды, реализация (напр. WinNT). **Совместимость**   Совместимость ОС – свойство, позволяющее выполнять приложения, написанные для других ОС.   * Двоичная совместимость – исполняемый файл. Определяется архитектурой процессора, совпадение API. Внутренняя структура файла должна соответствовать структуре, используемой в данной ОС. При несовпадении структуры необходима эмуляция двоичного кода. Эмуляция используется в системах кросс-разработки. * На уровне исходных кодов – совместимость на уровне компиляторов, совместимость библиотек, системных вызовов.   Эмуляция двоичного кода используется в системах кросс-разработки – системах, предназначенных для разработки программ в 2х машинной конфигурации.  Состав системы кросс-разработки:   * Средства редактирования * Средства компиляции * Средства отладки   Все это находится на инструментальной машине, а в готовом виде передается на целевую. Использование кросс-средств: системы программирования МК (Intel, Atmel и др.). ОС WinCE, PalmOS. Такие ОС включают в себя набор компиляторов и ассемблеров на инструментальной машине под ее ОС, библиотеки, выполняющие большую часть функций целевой ОС, средства отладки.  **Прикладные среды. Способы реализации**  Альтернатива эмуляции – прикладные среды. В состав каждой прикладной среды входит набор функций интерфейса API. Прикладные среды имитируют обращение к библиотечным функциям инородной среды. На самом деле обращение происходит к внутренним библиотекам. Это называется трансляцией библиотек. Чтобы программа, написанная под одной ОС работала под другой, необходимо обеспечить бесконфликтное взаимодействие способов управления процессами в разных ОС.  Способы реализации прикладных сред  1. В данном случае прикладная программная среда реализована как верхний слой ядра родной ОС.    Отдельный слой (API). Прикладная программная среда на основе нескольких равноправных API.  3. Реализуется по микроядерному принципу (Windows NT).   1. Процесс сосстояния жизненный цикл многозадачных ОС.Утилиты статуса,информационной структуры.   Жизненный цикл процесса:   * Выполнение – активное состояние процесса, когда он занимает процессорный ресурс * Ожидание – пассивное состояние – процесс заблокирован по внутренним причинам: ожидание данных, прерывания и т.д. (связанные с логикой самого процесса) * Готовность – пассивное состояние – процесс заблокирован по внешним причинам (ожидание события, прерывания и т.д.)       8->3 если есть место в ОЗУ, если нет, то 8->5  4, 6 – ожидание  1, 2 – выполнение  3, 5, 7 - готовность   1. . Жизненный цикл сигнала (unix).   Сигналы имеют определенный жизненный цикл. Вначале сигнал создается – высылается процессом или генерируется ядром. Затем сигнал ожидает доставки в процесс-приемник. Принято считать, что период ожидания сигнала равен промежутку времени между созданием сигнала и действием, которое этот сигнал высылает. В конце жизненного цикла сигнала происходит его перехват (прием) процессом и выполнение связанных с сигналом действий  Сигналы – способ передачи уведомления о событии, произошедшем либо между процессами, либо между процессом и ядром. Сигналы очень ресурсоемки. Ограничены с точки зрения системных средств. Они малоинформативны. Являются простейшим способом IPC. Используются для генерации простейших команд, уведомлений об ошибке. Обработка сигнала похожа на обработку прерывания. Сигнал имеет собственное имя и уникальный номер. Максимальное количество сигналов – 32. SIGINT - генерируется при нажатии клавиши del, Ctrl+C и т.д. Сигнал может быть отправлен процессу ядром или другим процессом. Это можно сделать посредством системного вызова.  Причины генерации сигналов:   1. Ядро отправляет сигнал процессу при нажатии клавиши; 2. Деление на 0; 3. Обращение к недоступной области памяти; 4. Особые аппаратные ситуации; 5. Особые программные ситуации.   Для суперпользователя сигнал может генерировать любой процесс. Если обычный пользователь, то его процессы могут посылать друг другу сигналы в рамках одного и того же пользовательского сеанса. Сигнал процесс может отправить самому себе и своей группе.  Основные действии я при приеме сигналов:  - игнорирование;- действия на данный сигнал, согласно умолчанию;  - обработка собственным обработчиком. Одно из возможных действий – диспозиция сигнала. Две фазы существования сигнала:   1. генерация и отправка; 2. доставка и обработка.   Промежуточная стадия – ожидание доставки. Генерация и отправка – в случае особых ситуаций, терминирование прерываний, системных вызовов, управления заданиями, ситуаций, когда возможна отправка сигналов по завершении временного интервала и уведомление (отправка сигнала по запросу самого процесса о каком-либо событии), нарушение квот. Доставка и обработка.  Для каждого сигнала прописана обработка по умолчанию, задаваемая ядром либо обработка из нашего процесса.  Возможные действия при получении сигнала:  - завершение процесса с сохранением образа процесса в ОЗУ;  - завершение процесса без сохранения образа процесса в ОЗУ;  - игнорирование сигнала;  - приостановка процесса;  - возобновление процесса при условии, что он был приостановлен.  Наиболее часты вариант – завершение процесса с сохранением образа в ОЗУ. Можно изменить действия по умолчанию на свой обработчик или указать системный. Можно заблокировать сигнал (отложить обработку на время).  Обработка. Для обработки сигнала необходимо, чтобы на момент его получения процесс выполнялся. Фаза ожидания доставки может быть длительной. Могут существовать задержки между отправкой и доставкой. Обработка не может быть выполнена до тех пор, пока планировщик не выберет процесс обработки на исполнение.  Доставка. Осуществляется ядром. Ядро от имени процесса осуществляет вызов ISSIG().Данный системный вызов выполняется в трех случаях:  - возвращение процесса из режима ядра в пользовательский режим;  - переход процесса в состояние ожидания;  - выход процесса из состояния ожидания.  Все это происходит при условии, что приоритет процесса допускает его прерывание сигналом. Системный вызов ISSIG() определяет, есть ли сигналы, ожидающие доставки. Если нет – то процесс просто выполняется.  ISSIG  Пример: нажатие клавиши del.  Ресурсоемкость сигналов. Для отправки-доставки требуется системный вызов. Для доставки – прерывание и его обработка. Для этого необходимо большое число операций со стеком - копирование пользовательского стека в системную область, извлечение параметров системных вызовов, результатов работы системных вызовов и др. Вывод: затраты слишком велики по сравнению с объемом передаваемой информации.   1. . файловая система FAT и HPFS |
|  |
|  |
| HPFS-тома используют размер сектора 512 байтов и имеют максимальный размер 2199Gb. HPFS том имеет очень небольшое количество фиксированных структур. Секторы 015 тома (BootBlock, имя тома, 32 бита ID, дисковая программа начальной загрузки). Начальная загрузка относительно сложна (в стандартах МСДОС) и может использовать HPFS в ограниченном режиме  Сектора 16 и 17 известны как SuperBlock и SpareBlock соответственно. SuperBlock изменяется только при помощи утилит. Он содержит указатели свободного пространства, список плохих блоков, полосу блока каталога, и корневую директорию. Он также содержит дату, соответствующую последней проверке и восстановлению утилитой CHKDSK/F. SpareBlock содержит различные флажки и указатели которые будут обсуждаться позже; Он изменяется, хотя нечасто, при работе системы.  Остаток диска разделен на 8MB полосы. Каждая полоса имеет собственный список свободного пространства, где биты представляют каждый сектор. Бит 0 если сектор использован 1 если сектор доступен. Списки размещаются в начале или хвосте списка.  Одна полоса, размещенная в "центре" диска, называется полосой блока каталога и обрабатывается специально. Обратите внимание, что размер полосы зависит от текущей реализации и может изменяться в более поздних версиях файловой системы.  Файлы и Fnodes  Каждый каталог или файл в HPFS-томе закрепляется за фундаментальным объектом файловой системы, называемым Fnode (произносится "eff node"). Каждый Fnode занимает одиночный сектор и содержит управляющую информацию, хронологию доступа, расширенные атрибуты и списки управления доступом, длину и первые 15 символов имени, и структуру распределения. Fnode всегда находится рядом с каталогом или файлом, который он представляет.  Структура распределения в Fnode может принимать несколько форм, в зависимости от размера каталога или файлов. HPFS просматривает файл как совокупность одного или более секторов. Из прикладной программы это не видно; файл появляется как непрерывный поток байтов.  Каталоги в HPFS, как и в FAT, образуют древовидную структуру. Но при этом внутри каталога HPFS строит сбалансированное дерево (B\*-Tree) на основе имён файлов для быстрого поиска файла по имени внутри каталога.  В файловой системе FAT смежные секторы диска объединяются в единицы, называемые кластерами. Количество секторов в кластере может быть равно 1 или степени двойки (см. далее). Для хранения данных файла отводится целое число кластеров (минимум один), так что, например, если размер файла составляет 40 байт, а размер кластера 4 кбайт, реально занят информацией файла будет лишь 1% отведенного для него места. Для избежания подобных ситуаций целесообразно уменьшать размер кластеров, а для сокращения объема адресной информации и повышения скорости файловых операций – наоборот. На практике выбирают некоторый компромисс. Так как емкость диска вполне может и не выражаться целым числом кластеров, обычно в конце тома присутствуют т.н. surplus sectors – «остаток» размером менее кластера, который не может отводиться ОС для хранения информации.  Пространство тома FAT32 логически разделено на три смежные области:  Зарезервированная область. Содержит служебные структуры, которые принадлежат загрузочной записи раздела (Partition Boot Record – PBR, для отличия от Master Boot Record – главной загрузочной записи диска; также PBR часто некорректно называется загрузочным сектором) и используются при инициализации тома;  Область таблицы FAT, содержащая массив индексных указателей ("ячеек"), соответствующих кластерам области данных. Обычно на диске представлено две копии таблицы FAT в целях надежности;  Область данных, где записано собственно содержимое файлов – т.е. текст текстовых файлов, кодированное изображение для файлов рисунков, оцифрованный звук для аудиофайлов и т.д. – а также т.н. метаданные – информация относительно имен файлов и папок, их атрибутов, времени создания и изменения, размеров и размещения на диске.  В FAT12 и FAT16 также специально выделяется область корневого каталога. Она имеет фиксированное положение (непосредственно после последнего элемента таблицы FAT) и фиксированный размер в секторах.  Каталог FAT (папка, директория) является обычным файлом, помеченным специальным атрибутом. Данными (содержимым) такого файла в любой версии FAT является цепочка 32-байтных файловых записей (записей каталога). Директория не может штатно содержать два файла с одинаковым именем. Если программа проверки диска обнаруживает искусственно созданную пару файлов с идентичным именем в одном каталоге, один из них переименовывается.  При создании каталога для него «пожизненно» выставляется DIR\_FileSize = 0. Размер содержимого каталога определяется простым следованием по цепочкам кластеров до метки End Of Chain. Размер самого каталога лимитируется файловой системой в 65 535 32-байтных записей (т.е. записи каталога в таблице FAT не могут занимать более 2Мб). Это ограничение призвано ускорить операции с файлами и позволить различным служебным программам использовать 16 битное целое (WORD) для подсчета количества записей в директории. (Как следствие, возникает теоретическое ограничение на количество файлов в каталоге – 65 535 при условии, что все имена файлов следуют стандарту 8.3). Каталогу отводится один кластер области данных (за исключением случая, если это корневой каталог FAT12/FAT16) и полям DIR\_FstClusHI / DIR\_FstClusLO присваивается значение номера этого кластера. В таблицу FAT для записи, соответствующей этому кластеру, помещается метка EOC, а сам кластер забивается нулями. Далее создаются два специальных файла, без которых директория FAT считается поврежденной (первые две 32-байтных записи в области данных кластера) – файлы нулевого размера “dot” (идентификатор каталога) и “dotdot” (указатель на родительский каталог) с именами “.” (точка) и “..” (две точки) соотв. Штампы даты-времени этих файлов приравниваются значениям для самого каталога на момент создания и не обновляются при изменениях каталога. Поля DIR\_FstClusHI / DIR\_FstClusLO файла «.» содержат значение номера содержащего его кластера, а файла «..» – номера первого кластера каталога, содержащего данный. Таким образом, файл «.» отсылает к самому каталогу, а файл «..» – к начальному кластеру родительского каталога; если родительский каталог – корневой, начальным кластером считается нулевой. |
|  |