1. Типы ОС (общего и специализированного назначения). Классификации ОС

2. Понятие и типы ресурсов. Понятие разделяемости, классификация разделяемых ресурсов.

3. Управление ресурсами. Функциональные компоненты ОС

4. Реентерабельный код, реализация, примеры применения

5. Системный вызов: назначение, реализация, примеры

6. Структуры и модели функционирования различных типов ядер.

7. Основные характеристики систем с различными типами архитектур. Предпочтительность той или иной архитектуры для различных классов ОС.

8. Модели функционирования в микроядерных архитектурах. Достоинства,недостатки

9. Многозадачность

10. Мультипроцессирование

11. Многозадачность при мультипроцессировании. Проблемы реализации кода.

12. Интерфейсы программирования и пользователя. Стандартизация интерфейсов

13. API – назначение, структура, проблемы совместимости, реализация

14. Стандарты POSIX, стандарты API различных ОС

15. Архитектурный слой middleware

16. Организация файлов , типы, форматы

17. Виртуальная ФС. Структуры vfs в Unix

18. Методы размещения и доступа к файлам

19. Методы учета занятого/свободного пространства (ФС)

20. Взаимосвязь методов размещения, доступа и учета свободного пространства (ФС)

21. Жизненный цикл и состояния процесса

22. Контекст и дескриптор процесса на примере Unix

23. Структуры контекстов процессов и нитей

24. Переключение контекста и затраты ресурсов на переключение

25. Методы и алгоритмы планирования

1. Типы ОС (общего и специализированного назначения). Классификации ОС.

Типы ОС (общего и специализированного назначения).

Операционные системы (ОС) общего и специализированного назначения отличаются по своим основным целям и функциональности. Вот некоторые основные различия между ними:

ОС общего назначения:

1. Универсальность: ОС общего назначения разработаны для широкого спектра задач и приложений. Они предоставляют общие функции и возможности для различных пользователей и приложений.
2. Пользовательский интерфейс: ОС общего назначения предоставляют графический интерфейс пользователя (GUI) и удобные средства взаимодействия с компьютером, такие как мышь и клавиатура.
3. Расширяемость: ОС общего назначения предоставляют возможность установки и запуска сторонних приложений и программного обеспечения.
4. Общедоступность: ОС общего назначения широко доступны для обычных пользователей и могут быть установлены на различные устройства, такие как настольные компьютеры, ноутбуки и планшеты.

ОС специализированного назначения:

1. Специализация: ОС специализированного назначения разработаны для конкретных задач или областей применения, таких как встроенные системы, серверы, мобильные устройства и т. д.
2. Оптимизация: ОС специализированного назначения нацелены на оптимизацию производительности и решение узкоспециализированных задач, обычно без излишней нагрузки на ресурсы.
3. Управление ресурсами: ОС специализированного назначения обычно предоставляют дополнительные функции и возможности для управления и контроля конкретными ресурсами, такими как сетевые соединения, серверное оборудование и т. д.
4. Ограниченность: ОС специализированного назначения могут быть ограничены в функциональности и поддержке сторонних приложений, поскольку они нацелены на конкретные сценарии использования.

В целом, ОС общего назначения предназначены для широкого круга пользователей и задач, в то время как ОС специализированного назначения разработаны для решения конкретных задач или работают в определенных областях применения.

Классификация ОС:

! по назначению:

~ универсальные (общего назначения)

~ специализированные (фиксированный набор приложений)

! по способу загрузки:

~ загружаемые (выполняют функции после считывания с некоторого накопителя)

~ резидентные (постоянно загружены в память, чаще специализированные)

! в соответствии с особенностями управления ресурсами:

~ по управлению процессорным временем: наличие многозадачности:

~~ однозадачные (1 user, 1 задача)

~~ многозадачные (много users, много задач)

~ по видам многозадачности

~~ не вытесняющие (процесс выполняется до того, пока сам не освободит ресурс/завершится)

~~ вытесняющие (ОС распределяет кванты времени для всех задач, то есть может быть прервана)

! поддержка работы пользователя:

~ однопользовательские

~ многопользовательские

! в соответствии с алгоритмами управления процессором:

~ многозадачные/однозадачные/мультипрограммирование

~ многонитевые/многопоточные и без

~ многопроцессорные/однопроцессорные/мультипроцессирование

Многозадачные:

+ с разделением времени

+ реального времени (HRT, SRT)

+ с пакетной обработкой

+ с вытесняющей и невытесняющей многозадачностью

2. Понятие и типы ресурсов. Понятие разделяемости, классификация разделяемых ресурсов.

Понятие ресурсов

Ресурсы ОС можно определить, как все, чем ОС может управлять и распределять между процессами и пользователями, включая процессорное время, оперативную память, дисковое пространство, сетевые ресурсы, ресурсы устройств ввода-вывода (клавиатура, мышь и т.д.) и так далее.

Типы ресурсов

- Разделяемые
  - ✓ Одновременное использование (использование несколькими потребителями в один момент времени)
  - ✓ Параллельное использование (в течении некоторого времени потребители используют ресурс попеременно)
- Неразделяемые

Виды ресурсов

- Аппаратные
  - ✓ Процессорное время
  - ✓ Оперативная память

    Основные способы разделения оперативной памяти:
    - ➢ Временной
    - ➢ Пространственный

    Способы выделения памяти процессорам:
    - ➢ Статический (резервация для фиксированного числа процессоров)
    - ➢ Динамический (по запросу процессора)
  - ✓ Внешняя память. 2 ресурса:
    - ➢ Пространство (разделяется пространственным способом)
    - ➢ Доступ (разделяется временным способом)

- ✓ Внешние устройства:
  - ➢ Параллельно разделяемые (прямой доступ) | разделяемый ресурс
  - ➢ Последовательный доступ | неразделяемый ресурс

- Программные

  Основные программные ресурсы – системные программные модули, разделяемые между выполняющимися процессами/программами/ задачами.

  Разделяемыми могут быть только те, для которых возможно многократное использование без искажения кода и данных:

  - ✓ Реентабельные (Ресурсы, которые могут быть использованы несколькими потоками или процессами одновременно без взаимного влияния. Такие ресурсы не содержат состояния, которое было бы изменено одним потоком и использовано бы другим)
  - ✓ Повторно-входимые (Ресурсы, которые могут быть использованы несколькими потоками одновременно, но для этого нужна синхронизация доступа к ресурсам. Повторно входимые ресурсы являются удобным способом разделения данных между несколькими потоками, когда есть несколько разных объектов, которые нужно защитить от несогласованного доступа)

- Информационные (данные):
  - ✓ Доступ по чтению без спец средств
  - ✓ Доступ с возможностью изменения информационных ресурсов требует специальных алгоритмов

3. Управление ресурсами. Функциональные компоненты ОС.

Независимо от типа ресурсов управление ресурсами осуществляется с помощью:

1) Планирование ресурса;
2) Удовлетворение запросов на ресурсы
3) Отслеживание и учет состояния ресурса
4) Разрешение конфликтов между потребителями ресурсов

Функциональные компоненты локальной ОС

Основное представление состава ОС – группирование по функциональному признаку в соответствии с типами локальных ресурсов:

1) подсистема управления процессами. Планировщик процессов определяет порядок и приоритет выполнения процессов на цп.
2) подсистема управления памятью. Диспетчер памяти контролирует выделение и освобождение ресурсов памяти.
3) ФС. Файловая система управляет хранением и извлечением файлов на дополнительных устройствах хранения, таких как жесткие диски или твердотельные накопители.
4) система ввода/вывода. Диспетчер ввода/вывода (ввода/вывода) координирует поток данных между ЦП, памятью и периферийными устройствами. Он управляет запросами ввода-вывода, буферизует данные и обеспечивает эффективное использование ресурсов ввода-вывода.
5) UI/GUI. UI/GUI позволяет пользователям вводить данные, управлять приложениями и отображать информацию с помощью графических элементов.
6) п/с защита данных и администрирования. Они предназначены для обеспечения безопасности данных и управления компьютерной средой на уровне отдельного ПК.

Важнейшей частью ОС, непосредственно влияющей на функционирование ЭВМ, является подсистема управления процессами.

подсистема управления процессами:

- ✓ планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами,
- ✓ занимается созданием и уничтожением процессов,
- ✓ обеспечивает процессы необходимыми системными ресурсами,
- ✓ поддерживает синхронизацию процессов, а также
- ✓ обеспечивает взаимодействие между процессами.

4. Реентерабельный код, реализация, примеры применения.

Реентерабельный код - это многократно используемая процедура, которую несколько программ могут вызывать, прерывать и повторно вызывать одновременно.

Реентерабельность тесно связана с безопасностью функции в многопоточной среде (*thread-safety*), тем не менее, это разные понятия. Обеспечение реентерабельности является ключевым моментом при программировании многозадачных систем, в частности, операционных систем.

Правила для реентерабельности:

1) Реентерабельный код может не содержать никаких статических или глобальных непостоянных данных без синхронизации;
2) Реентерабельный код не может изменять сам себя без синхронизации;
3) Реентерабельный код не может вызывать нереентерабельные программы или подпрограммы.

**** Примечание***

Повторный вход подпрограммы, которая работает с ресурсами операционной системы или нелокальными данными, зависит от атомарности соответствующих операций.

Например, если подпрограмма изменяет 64-разрядную глобальную переменную на 32-разрядной машине, операция может быть разделена на две 32-разрядные операции, и, таким образом, если подпрограмма прерывается во время выполнения и вызывается снова из обработчика прерываний, глобальная переменная может находиться в состоянии, когда было обновлено только 32 бита.

Примеры применения.

В следующем коде ни функции f, ни функции g не являются реентерабельными.

```
int v = 1;

int f()
{
   v += 2;
   return v;
}

int g()
{
   return f() + 2;
}
```

В приведенном выше примере *f()* зависит от непостоянной глобальной переменной *v*; таким образом, если *f()* прерывается во время выполнения *ISR*

(обработчика прерываний), который изменяет *v*, то повторный вход в *f()* вернет неправильное значение *v*.

Значение *v* и, следовательно, возвращаемое значение *f* нельзя предсказать с уверенностью: они будут варьироваться в зависимости от того, изменило ли прерывание *v* во время выполнения *f*., следовательно, *f* не является реентерабельным. Ни один из них не является *g*, потому что он вызывает *f*, который не является реентерабельным.

Эти слегка измененные версии являются реентерабельными:

```
        int f(int i)
{
   return i + 2;
}

int g(int i)
{
   return f(i) + 2;
}
```

Следующий код является потокобезопасным (thread-safety), но не (обязательно) реентерабельным:

```
        int function()
{
   mutex_lock();

   // ...
   // function body
   // ...

   mutex_unlock();
}
```

В приведенном выше примере *function()* может быть вызвана разными потоками без каких-либо проблем. Но, если функция используется в повторном обработчике прерываний и внутри функции возникает второе прерывание, вторая процедура зависнет навсегда. Поскольку обслуживание прерываний может привести к отключению других прерываний, может пострадать вся система.

5. Системный вызов: назначение, реализация, примеры.

Системный вызов является механизмом, который обеспечивает интерфейс между процессом и операционной системой (т.е. связывает их). Это программный метод, при котором компьютерная программа запрашивает сервис у ядра ОС. То есть системный вызов осуществляется в пользовательских программах. Кроме того, системные вызовы являются единственными точками входа в систему ядра.

Системные вызовы создают, удаляют и используют различные объекты, главные из которых – процессы и файлы. Имеются библиотеки процедур, которые загружают машинные регистры определенными параметрами и осуществляют *прерывание процессора*, после чего управление передается *обработчику данного вызова*, входящему в ядро операционной системы. Цель таких библиотек – сделать системный вызов похожим на обычный вызов подпрограммы. В unix\linux, это библиотека libc.

Параметры могут передаваться через:

- Регистры: сохраняются в регистрах процессора. Очень ограниченное количество.
- Память: функция получает указатель на блок памяти с параметрами. Такая реализация в Линукс.
- Стек: фукнция берёт параметры из стека. Такая реализация в Windows.

В отличие от подпрограмм, при системном вызове задача переходит в привилегированный режим или режим ядра (kernel mode). Отсюда другое название системных вызовов – программные прерывания (не путать с аппаратными прерываниями).

В kernel mode работает код ядра ОС, исполняется он в адресном пространстве и в контексте вызвавшей его задачи. Это значит, что ядро ОС имеет полный доступ к памяти пользовательской программы, и при системном вызове достаточно передать адреса ≥ 1 областей памяти с параметрами вызова и адреса ≥ 1 областей памяти для результатов вызова (это для Линукс, см. выше).

В большинстве ОС системный вызов осуществляется командой программного прерывания (INT). Программное прерывание – это синхронное событие (событие не меняется от количества и времени вызовов), оно может быть повторено при выполнении одного и того же программного кода.

Тезисно.

Существует таблица указателей на функции, реализующие системные вызовы. Номер функции – её индекс в ней.

Архитектура системного вызова:

Шаг 1) Процессы выполняются в пользовательском режиме до тех пор, пока системный вызов не прервет его.

Шаг 2) Ядру передаётся номер системного вызова. После этого системный вызов выполняется в kernel mode в приоритетном порядке.

Шаг 3) По завершении выполнения системного вызова управление возвращается в user mode.

Типы системных вызовов:

- Управление процессами
- Управление файлами
- Управление устройствами
- Управление памятью
- Коммуникация

Примеры, когда нужны системные вызовы:

- Чтение и запись из файлов.
- Если файловая система хочет создать или удалить файлы
- Создание и управления новыми процессами.
- Сетевые подключения требуют системных вызовов для отправки и получения пакетов.
- Для доступа к аппаратным устройствам, таким как сканер, принтер и т.д.

Примеры функций системных вызовов, которые мы использовали в лабораторных работах на парах:

- wait() – приостановка родительского процесса . Когда дочерний процесс завершает выполнение, элемент управления возвращается к родительскому процессу.
- fork() – создание процессов, которые являются их копиями. Родительский останавливается до завершения дочернего.
- exec() – замена старого файла или программы из процесса новым файлом или программой. Сохраняет PID, но остальное – меняет.
- kill() – отправляет сигналы, по умолчанию – о завершении процесса.
- exit() – завершение программы.

категории	Windows	Юникс
Контроль процесса	CreateProcess () ExitProcess () WaitForSingleObject ()	fork () exit () wait ()
Манипулирование устройством	SetConsoleMode () ReadConsole () WriteConsole ()	loctl () read () write ()
Манипулирование файлами	CreateFile () ReadFile () WriteFile () CloseHandle ()	Открыть () Читать () написать () закрыты!)
Информационное обслуживание	GetCurrentProcessID () SetTimer () Sleep ()	getpid () alarm () sleep ()
связь	CreatePipe () CreateFileMapping () MapViewOfFile ()	Pipe () shm_open () mmap ()
защита	SetFileSecurity () InitlializeSecurityDescriptor () SetSecurityDescriptorGroup ()	Chmod () Umask () Chown ()

6. Структуры и модели функционирования различных типов ядер.

- **Монолитное ядро.** Все части ядра работают в одном адресном пространстве. Это такая схема операционной системы, при которой все компоненты ее ядра являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур. Монолитное ядро — старейший способ организации операционных систем.

Достоинства: Незначительно выше скорость работы, упрощенная разработка модулей.

Недостатки: Поскольку всё ядро работает в одном адресном пространстве, сбой в одном из компонентов может нарушить работоспособность всей системы. Требуется перекомпиляция ядра при любом изменении состава оборудования.

- **Модульное ядро** — современная, усовершенствованная модификация архитектуры монолитных ядер операционных систем. В отличие от «классических» монолитных ядер, модульные ядра, как правило, не требуют полной перекомпиляции ядра при изменении состава аппаратного обеспечения компьютера. Вместо этого модульные ядра предоставляют тот или иной механизм подгрузки модулей ядра, поддерживающих то или иное аппаратное обеспечение (например, драйверов). При этом подгрузка модулей может быть, как динамической (выполняемой «на лету», без перезагрузки ОС, в работающей системе), так и статической (выполняемой при перезагрузке ОС после переконфигурирования системы на загрузку тех или иных модулей).

- **Микроядро** предоставляет только элементарные функции управления процессами и минимальный набор абстракций для работы с оборудованием. Бóльшая часть работы осуществляется с помощью специальных пользовательских процессов, называемых *сервисами*. Решающим критерием «микроядерности» является размещение всех или почти всех драйверов и модулей в сервисных процессах, иногда с явной невозможностью загрузки любых модулей расширения в собственное микроядро, а также разработки таких расширений.

Достоинства: Устойчивость к сбоям оборудования, ошибкам в компонентах системы. Основное достоинство микроядерной архитектуры — высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонентов. В микроядерной операционной системе можно, не прерывая её работы, загружать и выгружать новые драйверы, файловые системы и т. д. Существенно упрощается процесс отладки компонентов ядра, так как новая версия драйвера может загружаться без перезапуска всей операционной системы. Компоненты ядра операционной системы ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства. Микроядерная архитектура повышает надежность системы, поскольку ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра.

Недостатки: Передача данных между процессами требует накладных расходов.

Классические микроядра предоставляют лишь очень небольшой набор низкоуровневых примитивов, или системных вызовов, реализующих базовые сервисы операционной системы.

Сервисные процессы (в принятой в семействе UNIX терминологии — «демоны») активно используются в самых различных ОС для задач типа запуска программ по расписанию (UNIX и Windows NT), ведения журналов событий (UNIX и Windows NT), централизованной проверки паролей и хранения пароля текущего интерактивного пользователя в специально ограниченной области памяти (Windows NT). Тем не менее, не следует считать ОС микроядерными только из-за использований такой архитектуры.

Примеры: Symbian OS; Windows CE; OpenVMS; Mach, используемый в GNU/Hurd и Mac OS X; QNX; AIX; Minix; ChorusOS; AmigaOS; MorphOS.

- **Экзоядро** — ядро операционной системы, предоставляющее лишь функции для взаимодействия между процессами, безопасного выделения и освобождения ресурсов. Предполагается, что API для прикладных программ будут предоставляться внешними по отношению к ядру библиотеками (откуда и название архитектуры).

Возможность доступа к устройствам на уровне контроллеров позволит эффективно решать некоторые задачи, которые плохо вписываются в рамки универсальной ОС, например, реализация СУБД будет иметь доступ к диску на уровне секторов диска, а не файлов и кластеров, что положительно скажется на быстродействии.

- **Наноядро** — архитектура ядра операционной системы, в рамках которой крайне упрощённое и минималистичное ядро выполняет лишь одну задачу — обработку аппаратных прерываний, генерируемых устройствами компьютера. После обработки прерываний от аппаратуры наноядро, в свою очередь, посылает информацию о результатах обработки (например, полученные с клавиатуры символы) вышележащему программному обеспечению при помощи того же механизма прерываний. Примером является KeyKOS — самая первая ОС на наноядре. Первая версия вышла ещё в 1983-м году.
- **Гибридные ядра** — это модифицированные микроядра, позволяющие для ускорения работы запускать «несущественные» части в пространстве ядра. Пример: ядра ОС Windows семейства NT.

7. Основные характеристики систем с различными типами архитектур. Предпочтительность той или иной архитектуры для различных классов ОС.

Архитектура ОС - структурная и функциональная организация операционной системы на основе некоторых программных модулей.

Типы архитектур:

Монолитная. Единая неструктурированная система программных модулей. В ней используется одно ядро, которое компонуется как одна программа, работающая в привилегированном режиме. Ядро представляет собой набор процедур, каждая из которых имеет доступ друг к другу. Таким образом, все её компоненты являются составными частями одной программы, используют общие структуры данных.

Достоинства:

1) Относительно высокая производительность, так как все компоненты ядра общаются в одном адресном пространстве;

2) Простота добавления новых функций в ядро ОС, так как это делается путем добавления в уже существующий модуль;

Недостатки:

1) Поскольку всё ядро работает в одном адресном пространстве, сбой в одном из компонентов может нарушить работоспособность всей системы;

2) Невозможность модификации компонентов без перекомпиляции всего ядра;

3) Тяжело переносить на другие аппаратные платформы;

Использование: Монолитная архитектура ОС может использоваться в операционных системах для персональных компьютеров и серверов. Она предпочтительна для ОС, которые должны обеспечивать высокую производительность и имеют статический набор функций. Она менее подходит для ОС, которые должны поддерживать динамические функции и приложения, такие как мобильные, встроенные или распределенные системы.

Примеры: linux, ms-dos

Многоуровневая. Структурированная система, состоящая из нескольких уровней абстракции (т.е. логическое разделение), каждый из которых выполняет определённые функции (уровень управления аппаратурой, файловой системы, управления процессами и тд, верхний уровень - интерфейс пользователя). Каждый уровень взаимодействует только с соседним нижележащим через его интерфейс

(запрещается перепрыгивать через уровни). Внутренние структуры данных каждого уровня не доступны другим уровням, а реализации процедур уровня скрыты и не зависят от реализаций процедур внутри других уровней.

Достоинства:

1) Вносить функциональность достаточно в один уровень, не затрагивая остальные;

2) При переносе на другую платформу меняется только аппаратный уровень;

3) Можно достаточно легко выявить уровень, на котором произошёл сбой; компоненты одного уровня не влияют на другой.

Недостатки:

1) Сложны к разработке (тяжело спроектировать такую систему и правильно определить порядок слоев)

2) Эффективность замедлена из-за общения между слоями, что требует больше вычислительных ресурсов

Использование: Многоуровневая архитектура операционной системы (ОС) предпочтительна для крупных и сложных систем, где высокая модульность и расширяемость являются важными требованиями. Сюда можно отнести серверные ОС, ОС для кластерных вычислительных систем, ОС для научных и инженерных расчетов, ОС для систем управления производством и т.д.

Микроядерная (или архитектура клиент-сервер). Ядро ОС содержит только самые основные функции (управление памятью, процессами, вводом-выводом и прерываниями), а все остальные функции выполняют процессы-сервера (файловая система, драйвера устройств и сетевые протоколы) в пользовательском пространстве. Большинство составляющих ОС являются самостоятельными программами, взаимодействие между которыми осуществляется через микроядро.

Достоинства:

1) Система становится более гибкой из-за минимальной функциональности ядра. Благодаря модульности ядра удобно поддерживать и обновлять систему;

2) Устойчивость к ошибкам: благодаря тому, что компоненты ОС выполняются в отдельных процессах, ошибки в одной из них не влияют на работу другой;

3) Благодаря тому, что компоненты-сервера работают в пользовательском пространстве, система становится более безопасной;

Недостатки:

1) Передача данных между процессами снижает производительность;

Использование: Микроядерная архитектура ОС предпочтительна для систем, где безопасность, надежность и модульность критически важны, например, для ОС, работающих в автоматизированных системах управления производством, телекоммуникационных системах, рабочих станциях для научных и инженерных расчетов.

8. Модели функционирования в микроядерных архитектурах. Достоинства, недостатки

Суть микроядерной архитектуры состоит в следующем. В привилегированном режиме остается работать только очень небольшая часть ОС, называемая микро-ядром. Микроядро защищено от остальных частей ОС и приложений.

В микроядерной архитектуре функциональность операционной системы разбивается на отдельные модули, которые работают независимо друг от друга.

Работа микроядерной операционной системы соответствует известной модели **клиент-сервер**, в которой роль транспортных средств выполняет микроядро.

Идея модели клиент-сервер состоит в следующем: все компоненты операционной системы разделяются на программы – поставщики услуг (программы серверы, выполняющие определенные действия по запросам других программ), и программы – потребители услуг (программы клиенты, обращающиеся к серверам для выполнения определенных действий), где роль транспортных средств выполняет микроядро, так как непосредственная передача сообщений между приложениями невозможна, так как их адресные пространства изолированы друг от друга.

Микроядро, выполняющееся в привилегированном режиме, имеет доступ к адресным пространствам каждого из этих приложений и поэтому может работать в качестве посредника.

Запущенные в системе процессы-серверы постоянно находится в состоянии ожидания клиентских запросов. В случае необходимости, клиенты посылают серверам запросы на оказание требуемых им услуг, например, запрос на чтение файла, запрос на выделение памяти, запрос на вывод результатов на экран и т.п.

Преимущества микроядерной архитектуры:

- **Переносимость – т.к.** весь машинно-зависимый код изолирован в микроядре, поэтому для переноса системы на новый процессор требуется меньше изменений.
- **Расширяемость** - Добавление новой подсистемы требует разработки нового приложения, что никак не затрагивает целостность микроядра.
- **Надежность** - Каждый сервер выполняется в виде отдельного процесса в своей собственной области памяти и таким образом защищен от других серверов операционной системы.

Модель с микроядром хорошо подходит для **реализации распределенных вычислений**, так как использует механизмы, аналогичные сетевым: взаимодействие

клиентов и серверов путем обмена сообщениями. Серверы микроядерной архитектуры могут работать как на одном, так и на разных системах.

Недостаток – **производительность** операционной системы, функционирующей на основе обмена сообщениями между клиентами и серверами через микроядро, **заметно ниже** производительности операционной системы, функционирование которой основано на простых вызовах функций.

9. Многозадачность

Многозада́чность — способность операционной системы (среды выполнения) поддерживать одновременное совместное выполнение и взаимодействие нескольких задач (программ, потоков исполнения (поток исполнения — набор последовательных инструкций, выполняемых процессором во время работы программы)).

При этом задачи делят между собой общие ресурсы.

Виды многозадачности:

- Процессная многозадачность.

Здесь программа — наименьший элемент управляемого кода, которым может управлять планировщик операционной системы. Многозадачная система позволяет двум или более программам выполняться одновременно (процессы в отличии от потоков не делят общую память).

- Поточная многозадачность.

Многопоточность – специализированная форма многозадачности. Наименьший элемент управляемого кода — поток. Многопоточная система предоставляет возможность одновременного выполнения одной программой двух и более задач. (доп. информация: каждый поток имеет свой стек вызовов, но разделяет с другими потоками память и ресурсы процесса. Поток, инструкции которого процессор выполняет в данный момент времени, называется активным. Поскольку на одном процессорном ядре может в один момент времени выполняться только одна инструкция, то активным может быть только один вычислительный поток. Процесс выбора активного вычислительного потока называется планированием).

Также многозадачность бывает параллельной и псевдопараллельной:

- По-настоящему параллельное исполнение процессов возможно только в многопроцессорной системе.
- В однопроцессорной многозадачной системе поддерживается так называемое псевдопараллельное исполнение, при котором создается видимость параллельной работы нескольких процессов. В таких системах, процессы выполняются последовательно, занимая малые кванты процессорного времени.

Типы псевдопараллельной многозадачности (наверное, это можно не писать):

- Вытесняющая многозадачность - это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей.
- Невытесняющая многозадачность - это способ планирования процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

- Система называется однозадачной (первые системы были именно такими), если она не обладает свойством многозадачности, т.е. задачи в ней выполняются последовательно (например, DOS — однозадачная ОС, примеры для многозадачных ОС – Windows, Linux, Android).

10. Мультипроцессирование

Мультипроцессирование – одновременное выполнение нескольких процессов на вычислительной машине, имеющих несколько процессоров; если количество одновременно выполняемых процессов не превышает количества процессоров, то все процессы будут одновременно выполняться по времени, каждый на своем процессоре. Если больше, то на некоторых процессорах будет выполняться мультипрограммирование.

Мультипрограммирование – одновременное выполнение нескольких процессов на вычислительной машине с одним процессором. В этом случае распараллеливание процессов осуществляется за счет того, что пока для одного процесса выполняется операция ввода-вывода, другой процесс использует время ЦП.

При рассмотрении вопросов мультипроцессирования часто используют такие термины, как ***симметричный*** и ***ассиметричный*** мультипроцесс (м/п) (применительно к аппаратуре и к организации вычислительного процесса).

- ***Симметричная*** мультипроцессорная архитектура представляет из себя мультипроцессорную систему с несколькими одинаковыми м/п. Обычно такие м/п организованы в одном корпусе.
- В случае ***асимметричной*** мультипроцессорной архитектуры вычислительная система содержит несколько различных процессоров, выполняющих различные функции. В таких системах 1 из процессоров управляет работой остальных и называется ведущим, все остальные – ведомыми.

При асимметричной архитектуре возможно только асимметричная организация вычислительного процесса, а при симметричной архитектуре – как симметричная, так и асимметричная организация вычислительного процесса.

11. Многозадачность при мультипроцессировании. Проблемы реализации кода.

Многозадачность при мультипроцессировании - это способность ОС выполнять несколько процессов одновременно, используя несколько процессоров или ядер процессора. Это позволяет улучшить производительность приложений, делая возможным параллельное выполнение кода, распределенного между несколькими процессами.

Однако при реализации многозадачности при мультипроцессировании могут возникнуть различные проблемы, которые затрудняют выполнение кода. Рассмотрим некоторые из них:

1. Гонки данных: при использовании нескольких процессов, выполняющих один и тот же код, может возникнуть ситуация, когда два или более процесса пытаются изменить одну и ту же переменную, что приводит к непредсказуемому поведению приложения.

2. Взаимная блокировка: это ситуация, когда два или более процесса блокируют друг друга, не выполняя никакой работы. Например, если оба процесса пытаются получить доступ к ресурсам, защищенным семафором, то один из процессов может быть заблокирован, ожидая, пока другой процесс освободит ресурсы.

3. Недостаток ресурсов: в многозадачной среде необходимо более чем одно ядро процессора и достаточно памяти для выполнения нескольких процессов одновременно. Если эти ресурсы недоступны, то приложение будет работать очень медленно, или даже может вовсе не запуститься.

4. Управление процессами: управление множеством процессов, использующих многозадачность при мультипроцессировании, может быть сложной задачей. Необходима идеальная синхронизация между процессами, чтобы избежать возможных конфликтов и ошибок.

Чтобы избежать этих проблем, при реализации многозадачности при мультипроцессировании необходимо очень осторожно и тщательно проектировать код, обеспечивая должную синхронизацию и управляя доступом к общим ресурсам. Многократное тестирование и отладка приложения также являются ключевыми элементами успешной реализации многозадачности при мультипроцессировании.

12. Интерфейсы программирования и пользователя. Стандартизация интерфейсов

Интерфейс (И.) – совокупность средств информационного взаимодействия двух объектов или процессов. Взаимодействие – обмен информацией и управление. Обычно И. несимметричен: одна его сторона – управляющая (master, client), другая – подчиненная (slave, server). Средства взаимодействия – в зависимости от природы взаимодействующих объектов. Для трех видов объектов: аппаратура, программы, пользователи – имеем четыре вида интерфейса:

1. Аппаратный И. – между цифровыми устройствами. Обеспечивает двусторонний обмен сигналами на физическом уровне. Средства: кабели, шины, разъемы, сигналы, алгоритмы, временные диаграммы. Примеры:

- И. вв/вы – между центральными и внешними устройствами компьютера: RS-232, USB, Unibus, CAMAC и др.
- И. системной шины PC – между ЦП и быстрыми устройствами типа HD.

2. Аппаратно-программный И. – между программой и аппаратурой. Примеры:

- Система команд и прерываний компьютера.
- Взаимодействие приложения с внешним устройством при вв/вы:

Таким образом, *драйвер* является интерфейсной программой, «привязанной» к типу вн. устройства (точнее, контроллера).

3. Межпрограммный И. – между компонентами или слоями программного обеспечения. Традиционное название: CALL-интерфейс (набор форматов вызова процедур; в СУБД – CLI (Call Level Interface)), современное название – API (Application Programming Interface). Аппаратная аналогия:

- Розетка A: int A (char c, int b);
- Вилка X вставляется в розетку A: X = A (letter, 2)

4. UI – И. пользователя – между пользователем и программой. Диалоговое взаимодействие с помощью клавиатуры, мыши, графических элементов на экране и пр. Эпитеты: графический (GUI), дружественный (friendly), устойчивый к ошибкам (idiot-proof).

Принципы хорошего И.:

- *Инкапсуляция* внутреннего устройства, изоляция сторон И. Сервер не должен знать, **зачем** он задействуется, клиент – **как** тот действует.
- Экономность, лаконичность И. – минимум обмениваемой информации.

• Стандартизация, унификация И. – способствует сопрягаемости (interoperability) и повторному использованию компонентов.

Описание И. должно исчерпывающим образом определять функциональность компонента, но не предопределять способ ее реализации. Поэтому-то оно и является центральной частью внешней спецификации компонента или ПП: UI для приложения и API – для библиотеки.

Стандартизация — принятие соглашения по спецификации, производству и использованию аппаратных и программных средств вычислительной техники; установление и применение стандартов, норм, правил и т.п.

Стандарты обеспечивают возможность разработчикам информационных технологий использовать данные, программные, коммуникационные средства других разработчиков, осуществлять экспорт/импорт данных, интеграцию разных компонент информационных технологий.

13. API – назначение, структура, проблемы совместимости, реализация

API (от англ. Application Programming Interface) — описание способов взаимодействия одной компьютерной программы с другими. Обычно входит в описание какого-либо интернет-протокола, программного каркаса (фреймворка) или стандарта вызовов функций операционной системы. Часто реализуется отдельной программной библиотекой или сервисом операционной системы.

Варианты реализации API

1. Реализация функций API на уровне ОС.

Ответственность за выполнение функций API несет ОС. Программа, выполняющая функции, либо непосредственно входит в состав ОС, либо поставляется в составе динамически загружаемых библиотек.

Достигается **наибольшая эффективность выполнения функций API** (т.к. прикладная программа обращается непосредственно к ОС).

Недостаток: **практически полное отсутствие переносимости** не только кода результирующей программы, но и кода исходной программы.

Переносимости можно было бы добиться, если унифицировать функции API различных ОС.

С учетом корпоративных интересов производителей ОС такое направление их развития представляется практически невозможным. В лучшем случае при приложении определенных организационных усилий удается добиться стандартизации смыслового (семантического) наполнения основных функций API, но не их программного интерфейса.

Хорошо известным примером API такого рода может служить набор функций предоставляемых пользователю со стороны ОС типа Microsoft Windows – WinAPI (Windows API).

Даже внутри корпоративного API существует определенная несогласованность, которая несколько ограничивает переносимость программ между различными ОС типа Windows. Еще одним примером такого API можно считать набор сервисных функций ОС типа MS-DOS, реализованный в виде набора подпрограмм обслуживания программных прерываний.

2. Реализация функций API на уровне системы программирования.

Функции API предоставляются пользователю в виде библиотеки функций соответствующего языка программирования. Чаще всего это библиотека времени исполнения – RTL (run time library).

Эффективность будет несколько ниже, чем при непосредственном обращении к функциям ОС, т.к. библиотечная функция все равно выполняет обращения к функциям ОС.

Переносимость исходной программы будет самой высокой, т.к. синтаксис и семантика всех функций будут строго регламентированы в стандарте соответствующего языка программирования.

Для каждой ОС будет требоваться свой код RTL

3. Реализация функций API с помощью внешних библиотек.

Функции API предоставляются пользователю в виде библиотеки процедур и функций, созданной сторонним разработчиком

Эффективность наименьшая, т.к. внешняя библиотека обращается как к функциям ОС, так и к функциям RTL.

Соответственно, API отвечает на вопрос “Как ко мне, к моей системе можно обратиться?”, и включает в себя:

- саму операцию, которую мы можем выполнить,
- данные, которые поступают на вход,
- данные, которые оказываются на выходе (контент данных или сообщение об ошибке).

Совместимость на уровне двоичного кода

Под совместимостью на уровне двоичного кода понимается возможность потребителя API использовать его в более новой версии без перекомпиляции. Такие изменения, как добавление методов или добавление новой реализации интерфейса к типу, не влияют на совместимость двоичных файлов. Однако удаление открытых сигнатур сборки или их изменение таким образом, что потребители больше не имеют доступа к интерфейсу, предоставляемому сборкой, влияют на совместимость на уровне двоичного кода. Изменение такого рода называется изменением, несовместимым на уровне двоичного кода.

Совместимость исходного кода

Под совместимостью исходного кода понимается возможность перекомпиляции существующих потребителей API без изменения исходного кода. Изменение,

несовместимое на уровне исходного кода, имеет место, когда потребителю необходимо изменить исходный код для его успешной сборки в более новой версии API.

Совместимость во время разработки

Под совместимостью во время разработки понимается сохранение возможностей разработки в разных версиях Visual Studio и других средах разработки. Это может касаться поведения или пользовательского интерфейса конструкторов, однако самый важный аспект совместимости во время разработки — совместимость проекта. Необходимо, чтобы проект или решение можно было открывать и использовать в более новой версии среды разработки.

Обратная совместимость

Под обратной совместимостью понимается возможность выполнения существующего потребителя API в новой версии без изменения поведения. На обратную совместимость влияют как изменения поведения, так и изменения в совместимости на уровне двоичного кода. Если потребитель не может работать с более новой версией API или ведет себя иначе при ее использовании, этот API не обладает обратной совместимостью.

Применять изменения, влияющие на обратную совместимость, не рекомендуется, так как разработчик предполагают наличие обратной совместимости в более новых версиях API.

Прямая совместимость

Под прямой совместимостью понимается возможность выполнения существующего потребителя API в предыдущей версии без изменения поведения. Если потребитель не может работать с более старой версией API или ведет себя иначе при ее использовании, этот API не обладает прямой совместимостью.

Обеспечение прямой совместимости практически исключает любые изменения или дополнения в новых версиях, так как такие изменения препятствуют работе потребителя, предназначенного для более поздней версии, с более ранней версией. Разработчики ожидают, что потребитель, использующий более новую версию API, может работать неправильно с более старой версией.

14. Стандарты POSIX, стандарты API различных ОС

POSIX (англ. ***P**ortable **O**perating **S**ystem **I**nterface* — переносимый интерфейс операционных систем) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой. Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода.

Стандарт состоит из четырех основных разделов:

- **Основные определения** (англ. *Base definitions*) — список основных определений и соглашений, используемых в спецификациях, и список заголовочных файлов языка Си, которые должны быть предоставлены соответствующей стандарту системой.
- **Оболочка и утилиты** (англ. *Shell and utilities*) — описание утилит и командной оболочки sh, стандарты регулярных выражений.
- **Системные интерфейсы** (англ. *System interfaces*) — список системных вызовов языка Си.
- **Обоснование** (англ. *Rationale*) — объяснение принципов, используемых в стандарте.

Далее приведены основные версии POSIX с функциями, которые они привнесли.

POSIX.1. Системное API для языка Си

- Создание и управление процессами
- Сигналы
  - ✓ Исключения плавающей точки
  - ✓ Нарушение сегментации
  - ✓ Запрещённые директивы
  - ✓ Ошибки шины
  - ✓ Таймеры
- Операции над файлами (каталогами)
- Каналы
- Библиотека Си (стандарт Си)
- Интерфейс и контроль ввода-вывода

- Триггеры процессов

POSIX.2. Командная оболочка и утилиты

- Командный интерпретатор
- Программные утилиты

POSIX.1b. Расширения реального времени

- Планировка приоритетов
- Сигналы реального времени
- Часы и таймеры
- Семафоры
- Передача сообщений
- Управление памятью
- Синхронизация файлов и асинхронный ввод-вывод
- Интерфейс блокировки виртуальной памяти

POSIX.1c. Потоки

- Создание, контроль и завершение выполнения потоков
- Планировщик потоков
- Синхронизация потоков
- Обработка сигналов

POSIX.1-1996

Объединил все предыдущие (кроме POSIX 2.

POSIX.1d. Дополнительные расширения реального времени

POSIX.1g. Протоколо-независимые интерфейсы

POSIX.1j. Продвинутые расширения реального времени

С следующих версии не вносят значительных изменений, а лишь исправляют ошибки предыдущих.

API (от англ. ***A**pplication **P**rogramming **I**nterface*[1]) — описание способов взаимодействия одной компьютерной программы с другими.

API (интерфейс прикладного программирования) упрощает процесс программирования при создании приложений, абстрагируя базовую реализацию и предоставляя только объекты или действия, необходимые разработчику.

Windows API

Windows API спроектирован для использования в языке Си для написания прикладных программ, предназначенных для работы под управлением операционной системы MS Windows.

Версии

- **Win16** — первая версия *WinAPI* для 16-разрядных версий *Windows*. Изначально назывался *Windows API*, позднее был ретроспективно переименован в *Win16* для отличия от *Win32*. Описан в стандарте *ECMA-234*.
- **Win32** — 32-разрядный *API* для современных версий *Windows*. Самая популярная ныне версия. Базовые функции реализованы в динамически подключаемых библиотеках kernel32.dll и advapi32.dll; базовые модули графического интерфейса пользователя — в user32.dll и gdi32.dll. В современных версиях *Windows*, происходящих от *Windows NT*, работу *Win32 GUI* обеспечивают два модуля: csrss.exe (процесс исполнения клиент-сервер), работающий в пользовательском режиме, и win32k.sys в режиме ядра. Работу же системы обеспечивает ядро — ntoskrnl.exe.
- **Win32s** — подмножество *Win32*, устанавливаемое на семейство 16-разрядных систем *Windows 3.x* и реализующее ограниченный набор функций *Win32* для этих систем.
- **Win64** — 64-разрядная версия *Win32*, содержащая дополнительные функции *Windows* на платформах *x86-64* и *IA-64*.

15. Архитектурный слой middleware

Сервис ПО промежуточного слоя (middleware service) - сервис общего назначения, который располагается между платформами и приложениями. Под платформой мы подразумеваем набор низкоуровневых сервисов и функциональные особенности, определяемые парой: архитектура процессора и API операционной системы. Оно обеспечивает функциональность для интеллектуального и эффективного взаимодействия приложений, содействуя ускорению внедрения инноваций.

Промежуточное ПО действует как мост между различными технологиями, инструментами и базами данных, содействуя интеграции их в единую систему без дополнительных усилий. Затем единая система предоставляет своим пользователям унифицированные услуги. Например, клиентское приложение в среде Windows отправляет и получает данные с сервера Linux, но пользователи приложения не замечают эту разницу операционных систем.

Сервисы middleware представляют приложениям разнообразные функции API, которые, в сравнении с функциями операционных систем и сетевых служб, обеспечивают:

- прозрачный доступ к другим сетевым сервисам и приложениям;
- независимость от других сетевых сервисов;
- высокую надежность и постоянную готовность.

Виды промежуточного ПО:

- Программное обеспечение для межпрограммного взаимодействия (см. также IPC Inter-Process Communication).
- Программное обеспечение доступа к базам данных.

16. Организация файлов, типы, форматы

Файл - именованная совокупность данных, способная долговременно храниться во внешней памяти, доступная пользователям/приложениям для чтения/записи/запуска.

Идентификация файлов:

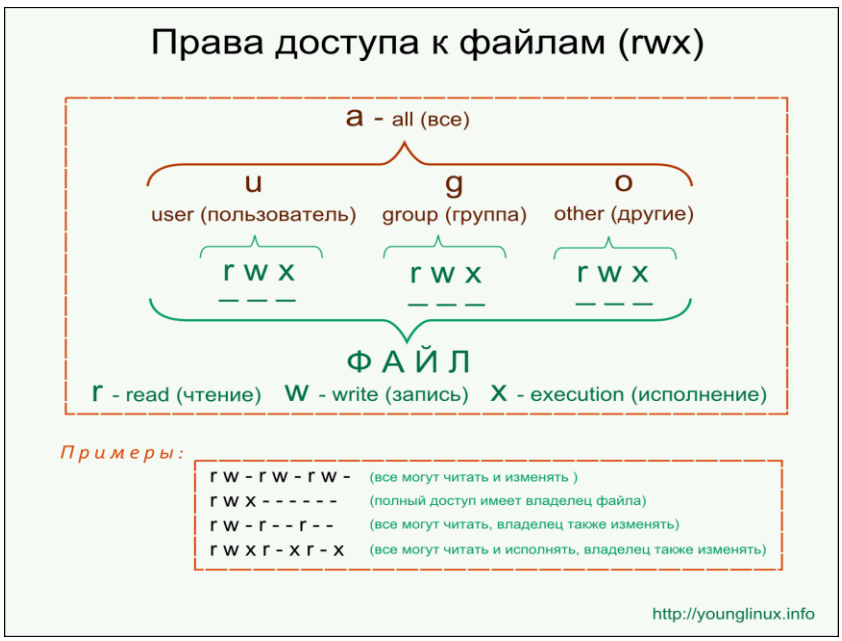
- Имя
- Дескриптор
  - ✓ Индексный, i-node. Просто номер файла в ФС
  - ✓ Файловый, fd. Идентификатор процесса, возвращается например при open()

Принципы размещения файлов, каталогов и системной информации на реальном носителе описываются физ.организацией ФС.

В Unix-подобных системах используется такая концепция, как монтирование файловых систем. Это позволяет подключать новые устройства (например, USB-накопители) к уже существующему дереву каталогов.

Каждый файл в Unix-подобной системе имеет набор атрибутов, которые включают:

- Владелец файла
- Права доступа
- Тип файла



Тип файла можно узнать командой ls –l.

Файлы системы могут быть следующих типов:

• Обычный файл. Это наиболее общий тип файла, который может содержать любые данные. Файлы могут быть текстовыми или бинарными.

b Специальный файл блочного устройства. Представляют физические устройства с блочным доступом (жесткие диски и CD-ROM).

c Файл символьного устройства. Принтеры, клавиатура, мышь.

d Директория. Представляют собой контейнеры для других файлов и каталогов.

l Ссылка.

- ✓ Символьная: содержимое файла - не данные, а путь к другому файлу. Удалять ссылку и файл можно независимо.
- ✓ Жёсткая: другое имя, но I-node тот же.

p FIFO. Канал для взаимодействия процессов.

s Сокет. Используются для обмена данными между процессами, возможно, работающими на разных машинах.

Форматы файлов

• Текстовые файлы - Это файлы, данные в которых представлены в виде текста. Файлы исходного кода, скрипты, конфигурационные файлы и документы.

• Бинарные файлы - Это файлы, данные в которых представлены в бинарном формате. Исполняемые файлы, изображения, аудио и видео файлы, и многие другие типы файлов.

В Unix-подобных системах отсутствуют расширения файлов. Для идентификации формата используются магические числа - специальные значения в начале бинарного файла.

17. Виртуальная ФС. Структуры vfs в Unix

Для организации доступа к разнообразным файловым системам (ФС) в Unix используется промежуточный слой абстракции - виртуальная файловая система (VFS). С точки зрения программиста VFS организована как интерфейс или абстрактный класс в объектно ориентированном языке программирования типа C++. VFS объявляет API доступа к файловой системе, а реализацию этого API отдаёт на откуп драйверам конкретных ФС, которые можно рассматривать, как производные классы, наследующие интерфейс VFS.

Кроме виртуальных функций *VFS* описывает обобщённые структуры

- *superblock*,
- *dentry* (*directory entry* запись в каталоге),
- *inode* (в некоторых ОС называется *vnode*).

Эти структуры содержат все основные структуры данных суперблока ФС, каталогов и *inode* из классической ФС *Unix*. Кроме того, структура *file* содержит информацию, необходимую для работы с открытыми файлами поскольку *VFS* является интерфейсом, то перечисленные структуры не содержат технических подробностей, таких как информации о размещении блоков данных файла или IP адреса сервера сетевой ФС. Для хранения деталей реализации, драйверу ФС в каждой из структур предоставляется дополнительное поле для хранения указателя на специфические для ФС структуры данных.

Список файловых систем, которые поддерживаются ядром, находится в файле **/proc/filesystems**.

Если пользователю необходимо решить задачи, которые не требуют непременного наличия ФС в ядре, применяется модуль FUSE (filesystem in userspace). Он создает ФС в пространстве пользователя. Виртуальные ФС, как правило, поддерживают шифрование и сетевое администрирование.

Сегодня на рынке существует целый спектр виртуальных ФС для ряда задач:

- **EncFS** — шифрует файлы, а затем выполняет сохранение зашифрованных файлов в необходимую пользователю директорию.
- **Aufs (AnotherUnionFS)** — объединяет несколько ФС (то же самое может делать с папками) в одну.
- **NFS (Network Filesystem)** — выполняет монтирование ФС удаленно.
- **ZFS (Zettabyte File System)** — ФС, созданная для ОС Solaris. Главные плюсы: отсутствие фрагментации, управление снапшотами и пулами хранения, изменяющийся размер блоков. Используется посредством FUSE.

18. Методы размещения и доступа к файлам

Методы размещения файлов.

Непрерывное размещение.

Простейшая схема размещения заключается в хранении каждого файла на диске в виде непрерывной последовательности блоков. Таким образом, на диске с блоками, имеющими размер 1 Кбайт, файл размером 50 Кбайт займет 50 последовательных блоков. При блоках, имеющих размер 2 Кбайт, под него будет выделено 25 последовательных блоков.

Следует заметить, что каждый файл начинается от границы нового блока, поэтому, если файл А фактически имел длину 3,5 блока, то в конце последнего блока часть пространства будет потеряна впустую.

Преимущества:

- Просто реализовать, поскольку отслеживание местонахождения принадлежащих файлу блоков сводится всего лишь к запоминанию двух чисел: дискового адреса первого блока и количества блоков в файле. При наличии номера первого блока номер любого другого блока может быть вычислен путем простого сложения.
- Превосходная производительность считывания, поскольку весь файл может быть считан с диска за одну операцию. Для нее потребуется только одна операция позиционирования (на первый блок). После этого никаких позиционирований или ожиданий подхода нужного сектора диска уже не потребуется, поэтому данные поступают на скорости, равной максимальной пропускной способности диска.

Недостатки:

- Со временем диск становится фрагментированным. Как это происходит, показано на рис. 4.7, б. При удалении файла его блоки освобождаются и на диске остается последовательность свободных блоков. Сначала фрагментация не составляет проблемы. Но со временем диск заполнится и понадобится либо его уплотнить, что является слишком затратной операцией, либо повторно использовать последовательности свободных блоков между файлами, для чего потребуется вести список таких свободных участков. Но при создании нового файла необходимо знать его окончательный размер, чтобы выбрать подходящий для размещения участок.

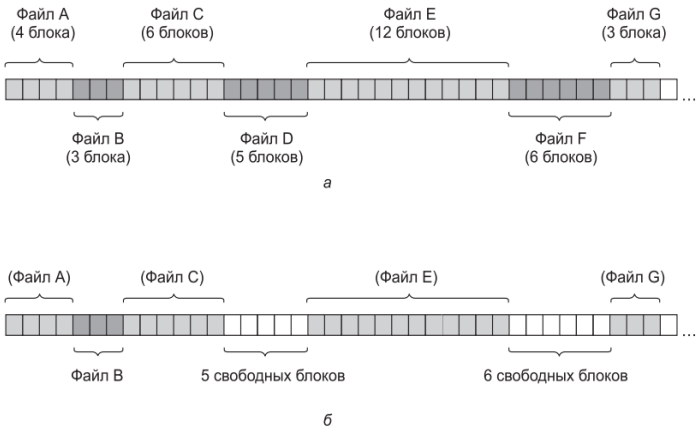


Рис. 4.7. Дисковое пространство: *а* — непрерывное размещение семи файлов; *б* — состояние диска после удаления файлов *D* и *F*

Размещение с использованием связанного списка

Второй метод хранения файлов заключается в представлении каждого файла в виде связанного списка дисковых блоков (рис. 4.8). Первое слово каждого блока используется в качестве указателя на следующий блок, а вся остальная часть блока предназначается для хранения данных.

Преимущества:

- Может быть использован каждый дисковый блок. При этом потери дискового пространства на фрагментацию отсутствуют (за исключением внутренней фрагментации в последнем блоке).
- Достаточно, чтобы в записи каталога хранился только дисковый адрес первого блока. Всю остальную информацию можно найти, начиная с этого блока.

Недостатки:

- Произвольный доступ является слишком медленным. Чтобы добраться до блока n, операционной системе нужно начать со стартовой позиции и прочитать поочередно n − 1 предшествующих блоков. Понятно, что осуществление стольких операций чтения окажется мучительно медленным.
- Объем хранилища данных в блоках уже не кратен степени числа 2, поскольку несколько байтов отнимает указатель. Когда первые несколько байтов каждого блока заняты указателем на следующий блок, чтение полноценного блока требует получения и соединения информации из двух дисковых блоков, из-за чего возникают дополнительные издержки при копировании.

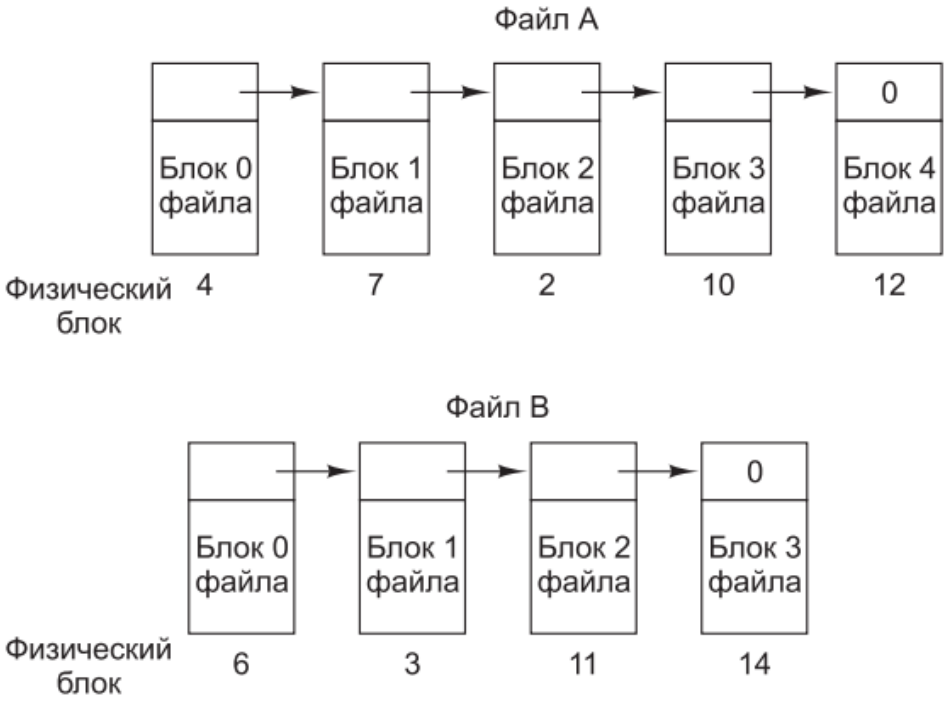


Рис. 4.8. Хранение файла в виде связанного списка дисковых блоков

Размещение с помощью связанного списка, использующего таблицу в памяти.

Недостатки размещения с помощью связанных списков могут быть устранены за счет изъятия слова указателя из каждого дискового блока и помещения его в таблицу в памяти.

На рис. 4.9 показано, как выглядит таблица для примера, приведенного на рис. 4.8. На обоих рисунках показаны два файла. Файл А использует в указанном порядке дисковые блоки 4, 7, 2, 10 и 12, а файл В — блоки 6, 3, 11 и 14. Используя таблицу, показанную на рис. 4.9, можно пройти всю цепочку от начального блока до самого конца. Обе цепочки заканчиваются специальным маркером (например, –1), который не является допустимым номером блока.

Преимущества:

- При использовании такой организации для данных доступен весь блок.
- Намного упрощается произвольный доступ. Хотя для поиска заданного смещения в файле по-прежнему нужно идти по цепочке, эта цепочка целиком находится в памяти, поэтому проход по ней может осуществляться без обращений к диску. Как и в предыдущем методе, в записи каталога достаточно хранить одно целое число (номер начального блока) и по-прежнему получать возможность определения местоположения всех блоков независимо от того, насколько большим будет размер файла.

Недостатки:

- Для работы вся таблица должна постоянно находиться в памяти.

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

I-узлы. Индексное размещение.

Последним из рассматриваемых методов отслеживания принадлежности конкретного блока конкретному файлу является связь с каждым файлом структуры данных, называемой i-узлом (index-node — индекс-узел), содержащей атрибуты файла и дисковые адреса его блоков. Простой пример приведен на рис. 4.10.

При индексируемом размещении блоки файла могут быть расположены как угодно разрозненно, но индексный блок содержит все ссылки на них. Ссылка на блок данных может быть выбрана непосредственно из индексного блока, без какого-либо поиска.

Преимущества:

- Большим преимуществом этой схемы перед связанными списками, использующими таблицу в памяти, является то, что i-узел должен быть в памяти только в том случае, когда открыт соответствующий файл. Если каждый i-узел занимает n байт, а одновременно может быть открыто максимум k файлов, общий объем памяти, занимаемой массивом, хранящим i-узлы открытых файлов, составляет всего лишь kn байт. Заранее нужно будет зарезервировать только этот объем памяти.

Недостатки:

• на индексный блок ложится критическая нагрузка: если его целостность будет нарушена, файл восстановлению не подлежит.

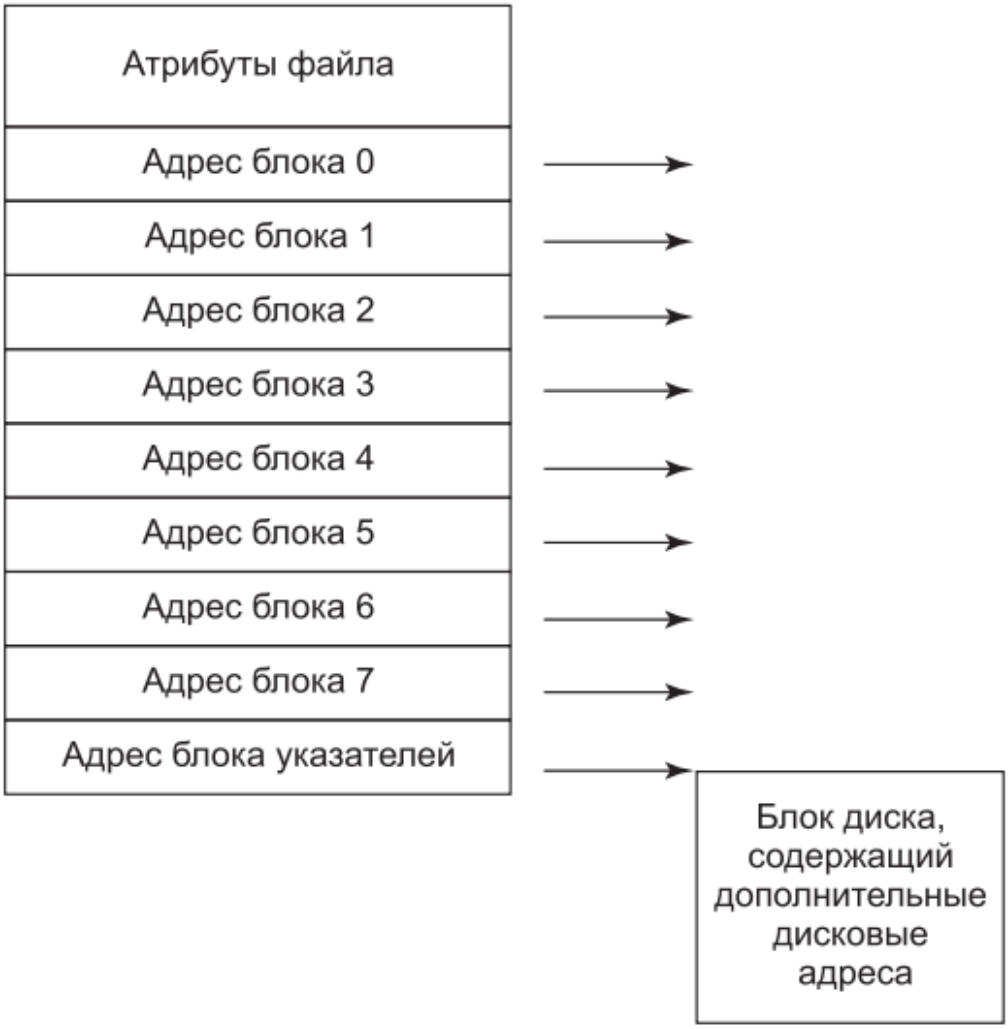


Рис. 4.10. Пример i-узла

Методы доступа к файлам.

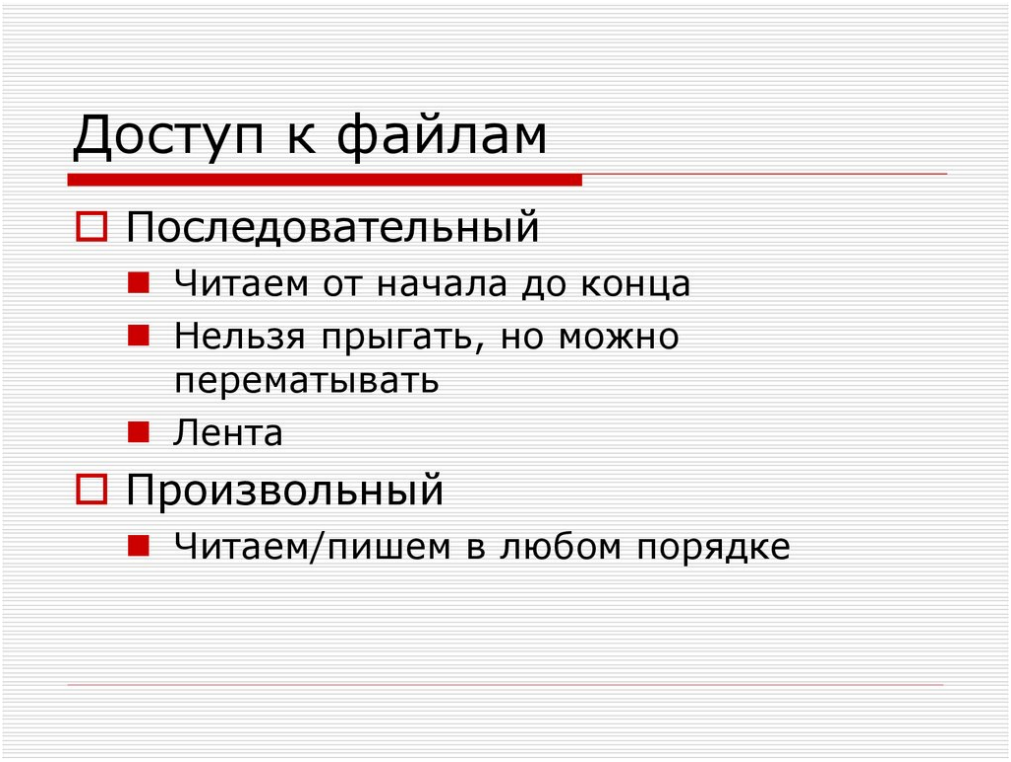
Файловая система может использовать два способа доступа к логическим записям: читать или записывать логические записи последовательно (последовательный доступ) или позиционировать файл на запись с указанным номером (прямой доступ).

Последовательный доступ.

Процесс может читать все байты или записи файла только по порядку, с самого начала. Нельзя перепрыгнуть и считать их вне порядка их следования. Но последовательные файлы можно перемотать назад, чтобы считать их по мере надобности.

Произвольный доступ.

Возможно считывать байты или записи файла вне порядка их размещения или получать доступ к записям по адресу(ключу).



19. Методы учета занятого/свободного пространства (ФС)

Существуют различные методы учета занятого/свободного пространства в файловых системах, некоторые из которых приведены ниже:

1. Bit - map: Метод используется для отслеживания использования блоков диска. Каждому блоку на диске соответствует один бит, где 1 означает занятый блок, а 0 - свободный блок. Этот метод обеспечивает быстрый доступ к информации, но требует большого объема памяти для хранения битовой карты.

2. Linked List: Для отслеживания свободных и занятых блоков используются связанные списки. В этом методе каждый блок содержит указатель на следующий свободный блок, что обеспечивает более эффективное использование дискового пространства, но требует большого количества I/O-операций для поиска свободных блоков.

3. Counting: Метод подсчитывает количество занятых и свободных блоков на диске. Этот метод прост в реализации, но не обеспечивает быстрого доступа к информации.

4. Bitmap-count: Метод сочетает преимущества обоих методов - битовой карты и подсчета. В этом методе используются битовые карты, но также подсчитывается количество свободных и занятых блоков на диске.

5. Tree: Этот метод использует дерево для отслеживания использования дискового пространства. Каждый узел дерева представляет собой блок диска, а листья дерева устанавливаются в значение 0 или 1 в зависимости от того, свободен или занят блок диска. Этот метод обеспечивает более эффективное использование дискового пространства, но требует большого количества I/O-операций для поиска свободных блоков.

Конкретный метод учета занятого/свободного пространства зависит от особенностей конкретной файловой системы и ее требований к производительности и объему занимаемой памяти.

20. Взаимосвязь методов размещения, доступа и учета свободного пространства (ФС)

Методы размещения, доступа и учета свободного пространства (ФС) взаимосвязаны и влияют друг на друга.

Метод размещения определяет, как файлы будут сохраняться на ФС и каким образом свободное пространство на диске будет управляться. Например, в некоторых методах файлы сохраняются последовательно, а в других - случайным образом. Этот метод также должен учитывать возможность дефрагментации ФС и эффективности использования свободного пространства.

Метод доступа определяет, как пользователи и программы могут получить доступ к файлам на ФС. Он включает в себя права доступа и механизмы защиты файлов, чтобы предотвратить несанкционированный доступ к конфиденциальной информации.

Метод учета свободного пространства определяет, как информация о свободном пространстве сохраняется и управляется. Он включает в себя механизмы мониторинга использования дискового пространства и оповещение пользователей при достижении критического уровня свободного пространства.

Все эти методы должны быть тщательно реализованы и интегрированы друг с другом, чтобы ФС работала эффективно и безопасно.

21. Жизненный цикл и состояния процесса

Процесс – базовое понятие ОС, часто кратко определяется как программа в стадии выполнения. Это динамический объект; возникает в ОС после того, как пользователь/ОС “запускает” программу на выполнение; новая самодостаточная единица вычислительной работы.

В современных ОС процесс не является неделимой минимальной единицей работы. Существуют понятия потока и нити (thread) для определения вычислительных работ, входящих в состав процесса и разделяющих между собой ресурсы системы.

В ОС, поддерживающих и процессы, и потоки, процесс рассматривается операционной системой как заявка на потребление всех видов ресурсов, кроме процессорного времени. Ресурс времени распределяется ОС между более мелкими единицами работы – потоками, позволяя распараллелить вычисления в рамках одного процесса. При этом ОС назначает процессу адресное пространство и набор ресурсов, которые совместно используются всеми его потоками. Причем в Unix-подобных ОС доминирует процесс, в Windows – поток.

Жизненный цикл содержит: состояния и переходы. Состояния бывают активными и пассивными (по отношению к возможности использования процессора)

Типы состояний:

- выполнение - активное состояние процесса
- ожидание - пассивное (процесс не может выполняться по внутренним причинам); ожидает осуществления некоторого события (например, ввод-вывод, получения сообщения, освобождения необходимого ему ресурса); характерно наличие очередей процессов по ресурсам в соответствии с приоритетами доступа к каждому из этих ресурсов
- готовность - пассивное (процесс заблокирован в связи с внешними обстоятельствами); имеет все требуемые ресурсы и готов выполняться, но процессор занят другим процессом; характерно наличие очередей процессов по приоритетам по отношению к процессору (списки дескрипторов отдельных процессов - позволяет легко проводить манипуляции с приоритетами процессов)

! Рассматриваем однопроцессорную систему!

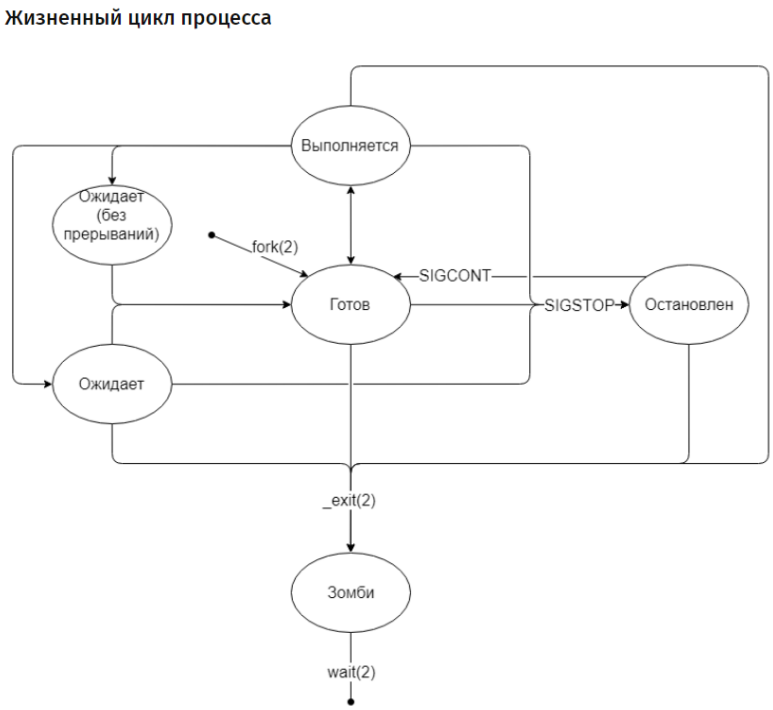
Упрощенный граф состояний:

- выполнение - только 1 процесс
- ожидание/готовность - множество процессов, образующие очереди

Упрощенная последовательность перехода:

1) активация процесса -> выполнение
2) выполнение (если процесс освободил процессор, т.е. был завершен)-> ожидание
3) выполнение (если процесс был насильно “вытеснен”, т.е. исчерпал квант процессорного времени) -> готовность

4) ожидание (если произошло ожидаемое событие) -> готовность



Обобщенное описание состояний:

1) Рождение процесса: Первый процесс в системе рождается через init — порождается непосредственно ядром. Все остальные процессы появляются путём дублирования текущего процесса с помощью системного вызова fork(2).

2) Готов: Сразу после выполнения fork(2) переходит в состояние «готов». Фактически, процесс стоит в очереди и ждет, когда планировщик (scheduler) в ядре даст процессу выполняться на процессоре.

3) Выполняется: Как только планировщик поставил процесс на выполнение, началось состояние «выполняется». Процесс может выполняться весь предложенный промежуток (квант) времени, а может уступить место другим процессам, воспользовавшись системным вывозом sched_yield.

4) Ожидает: Некоторые системные вызовы могут выполняться долго, например, ввод-вывод. В таких случаях процесс переходит в состояние «ожидает». Как только системный вызов будет выполнен, ядро переводит процесс в состояние «готов».В Linux также существует состояние «ожидает», в котором процесс не реагирует на сигналы прерывания. В этом состоянии процесс становится «неубиваемым», а все пришедшие сигналы встают в очередь до тех пор, пока процесс не выйдет из этого состояния. Ядро само выбирает, в какое из состояний перевести процесс. Чаще всего в состояние «ожидает (без прерываний)» попадают процессы, которые запрашивают ввод-вывод.

5) Остановлен: в любой момент можно приостановить выполнение процесса, отправив ему сигнал SIGSTOP. Процесс перейдет в состояние «остановлен» и будет находиться там до тех пор, пока ему не придёт сигнал продолжать работу (SIGCONT) или умереть (SIGKILL). Остальные сигналы будут поставлены в очередь.

6) Завершение: Ни одна программа не умеет завершаться сама. Они могут лишь попросить систему об этом с помощью системного вызова exit или быть завершенными системой из-за ошибки.

7) Зомби: Сразу после того, как процесс завершился (не важно, корректно или нет), ядро записывает информацию о том, как завершился процесс и переводит его в состояние «зомби». Иными словами, зомби — это завершившийся процесс, но память о нём всё ещё хранится в ядре. Более того, это второе состояние, в котором процесс может смело игнорировать сигнал SIGKILL.

8) Примечание: Код возврата и причина завершения процесса всё ещё хранится в ядре и ее нужно оттуда забрать. Для этого можно воспользоваться системными вызовами: wait и waitpid.

Бывают случаи, при которых родитель завершается раньше, чем ребёнок.Тогда родителем ребёнка станет init и он применит вызов wait(2), когда придёт время.После того, как родитель забрал информацию о смерти ребёнка, ядро стирает всю информацию о ребёнке, чтобы на его место вскоре пришел другой процесс.

Подробный перечень состояний (в ОС Unix):

1) Процесс выполняется в пользовательском режиме, или режиме задачи.
2) Процесс выполняется в привилегированном режиме, или 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. Это состояние в действительности не отличается от состояния "готовности к запуску в памяти", но используется для процессов, выполняющихся в режиме ядра, и собирающихся вернуться в режим задачи, т.е. ядро может при необходимости подкачивать процесс из состояния "резервирования".
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.

22. Контекст и дескриптор процесса на примере Unix

Контекст процесса - информация, описывающая состояние процесса и его операционной среды. Эта информация сохраняется, когда выполнение процесса приостанавливается, и восстанавливается, когда планировщик предоставляет процессу вычислительные ресурсы.

Он объединяет:

1) Пользовательский контекст - команды и данные процесса, стек задачи, содержимое совместно используемого пространства памяти в виртуальных адресах процесса.
2) Регистровый контекст - счетчик команда, регистр состояния процесса, указатель вершины стека, регистры общего назначения.
3) Системный контекст - запись в таблице процессов, часть адресного пространства задачи, стек ядра, записи частной таблицы областей процесса, динамическая часть системного контекста процесса.

Контекст содержит две части:

1) Статическая - программы процесса (машинные инструкции), данные, стек, разделяемая память, записи таблицы процессов и др.
2) Динамическая - включает элементы, хранящие регистровый контекст предыдущего уровня и стек ядра текущего уровня.

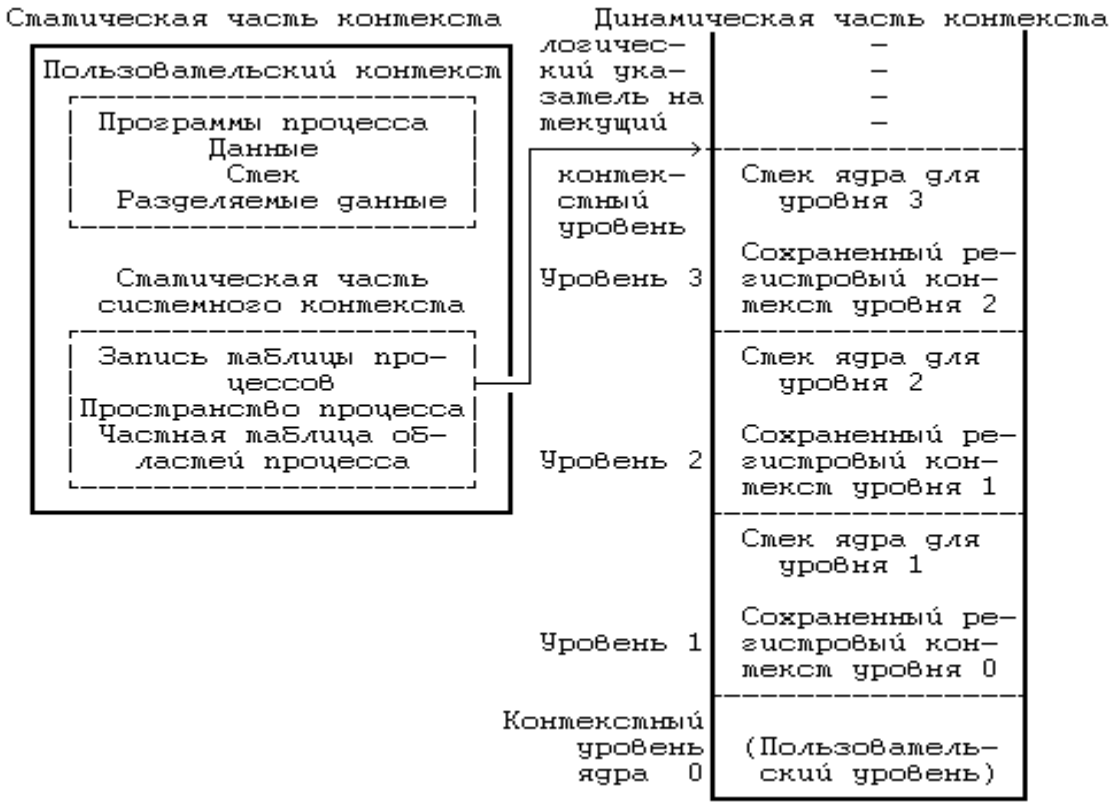
Дескриптор процесса содержит такую информацию о процессе, которая необходима ядру в течение всего жизненного цикла процесса, независимо от того, находится ли он в активном или пассивном состоянии, находится ли образ процесса в оперативной памяти или выгружен на диск.

Он содержит:

- идентификатор процесса,
- состояние процесса,
- данные о степени привилегированности и др.

Дескрипторы отдельных процессов объединены в список, образующий таблицу процессов. Память для нее отводится динамически в области ядра. На ее основе ОС осуществляет планирование и синхронизацию процессов.

23. Структуры контекстов процессов и нитей



динамическая часть контекста имеет вид стека

Поток — особый тип процесса, который делит виртуальное адресное пространство и обработчики сигналов с другими процессами. Для ядра каждый процесс идентифицируется по PID. Для так называемых потоков можно использовать термин TID, но для ядра это одно и то же. Можно понимать это так:

- если процесс однопоточный, то PID будет равен TID этого единственного потока;
- если в процессе работают несколько потоков, то у каждого потока свой TID, а PID идентифицирует группу потоков, которая разделяет адресное пространство, таблицу файловых дескрипторов...

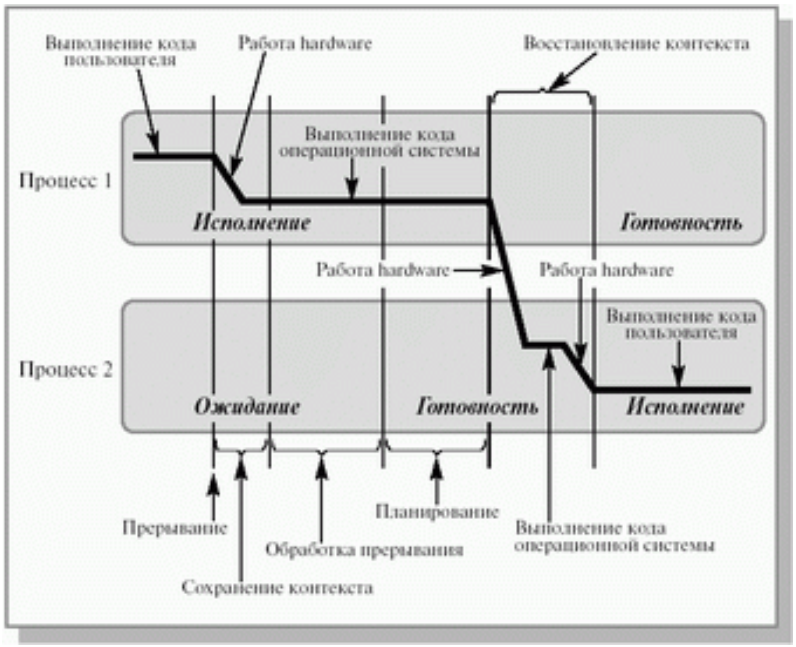
Структура процесса

Процесс в ядре представляется просто как структура с множеством полей, например:

- Идентификатор процесса (pid)
- Открытые файловые дескрипторы (fd)
- Обработчики сигналов (signal handler)
- Текущий рабочий каталог (cwd)
- Переменные окружения (environ)
- Код возврата

24. Переключение контекста и затраты ресурсов на переключение

Одной из функций операционной системы является планирование заданий и использования процессора. Процессор работает с процессом. Деятельность операционной системы состоит из цепочек операций, выполняемых над различными процессами, и сопровождается переключением процессора с одного процесса на другой. Для примера, на рисунке показана операция разблокирования процесса:



Процессор исполняет процесс 1, который находится в состоянии «исполнение». В момент исполнения возникает прерывание от устройства ввода-вывода, сигнализирующее об окончании операций на устройстве. Над выполняющимся процессом производится операция приостановки. Далее операционная система разблокирует процесс 2, инициировавший запрос на ввод-вывод и осуществляет запуск приостановленного процесса, выбранного в результате планирования. В результате обработки информации об окончании операции ввода-вывода произошла смена процесса, находящегося в состоянии «исполнение»

При переключении процессора с одного процесса на другой сохраняется контекст исполняющегося процесса и восстанавливается контекст процесса для исполнения. Процедура сохранения/восстановления контекстов процессов называется переключением контекста. Время, затраченное на переключение контекста, представляет собой накладные расходы, снижающие производительность системы. Оно меняется от машины к машине. В современных операционных системах для сокращения накладных расходов используется расширенная модель процессов, включающая в себя понятие нити исполнения.

25. Методы и алгоритмы планирования

В ОС общего назначения или иных ОС с высокой степенью универсальности по применению обычно предполагается, что входной поток задач случайный (*задачи носят случайный характер с точки зрения прогнозирования запрашиваемых для их решения ресурсов, и априорная информация об этом отсутствует, как и жесткие ограничения на сроки исполнения и время реакции системы на события*). При такой степени неопределенности самое разумное, что можно сделать для оптимизации вычислительного процесса в системе в целом – это минимизировать затраты ресурсов на само планирование, что делается за счет выбора самых простейших алгоритмов планирования и сокращения их количества в ОС. Это отражено в стандарте POSIX в виде рекомендуемых алгоритмов: круговое планирование и FIFO. Основные типы алгоритмов представлены в таблице ниже.

Дисциплина планирования	Порядок обслуживания заданий
RMS rate monotonic scheduling	Задания выполняются в порядке увеличения значений периода поступления *(требуют, чтобы прикладные задачи были периодическими, с постоянством интервалов времени использования центрального процессора)*
DMS deadline monotonic scheduling	Задания выполняются в порядке увеличения значений относительного срока выполнения
EDF earliest deadline first	Задания выполняются в порядке увеличения значений абсолютного срока завершения *(запускается процесс, у которого раньше всех наступит крайний срок. Как только будет готов новый процесс, система проверяет, не наступает ли его крайний срок раньше, чем у процесса, работающего в данный момент)*
LLF least laxity first	Задания выполняются в порядке увеличения запаса времени на выполнение
LWR least work remaining	Задания выполняются в порядке увеличения оставшейся длительности обслуживания
LIFO last in – first out	Процессор предоставляется заданию с наиболее поздним моментом времени регистрации

SPT shortest processing time	Задания выполняются в порядке увеличения значений объема требуемых вычислений

Сегодня в Unix подобных ОС, в частности в Linux, и других ОС, следующих стандарту POSIX, поддерживаются три базовые политики планирования: SCHED_FIFO, SCHED_RR, и SCHED_OTHER: одна для обычных процессов и две для процессов «реального» времени. Их реализация обеспечивается ядром, а точнее, планировщиком. Каждому процессу присваивается **статический приоритет** sched_priority, который можно изменить только при помощи системных вызовов. Ядро хранит в памяти списки всех работающих процессов для каждого возможного значения sched_priority. Для того, чтобы определить, какой процесс будет выполняться следующим, планировщик ищет непустой список (очередь) с наибольшим статическим приоритетом и запускает первый процесс из этого списка. Алгоритм планирования определяет, как процесс будет добавлен в список-очередь с тем же статическим приоритетом, и как он будет перемещаться внутри этого списка.

1) SCHED_FIFO: планировщик FIFO (First In-First Out). Можно использовать только со значениями статического приоритета, большими нуля. Это означает, что если процесс готов к работе, то он сразу запустится, а все обычные процессы с алгоритмом SCHED_OTHER будут приостановлены. Это простой алгоритм без квантования времени. Процесс с алгоритмом SCHED_FIFO, приостановленный другим процессом с большим приоритетом, останется в начале очереди процессов с равным приоритетом, и его исполнение будет продолжено сразу после того, как закончатся процессы с большими приоритетами. Когда процесс с алгоритмом SCHED_FIFO готов к работе, он помещается в конец очереди процессов с тем же приоритетом. Процесс работает до тех пор, пока не будет заблокирован запросом на ввод/вывод, приостановлен процессом с большим статическим приоритетом или не вызовет sched_yield.

2) SCHED_RR: циклический алгоритм планирования. Все, относящееся к алгоритму SCHED_FIFO, справедливо и для SCHED_RR за исключением того, что каждому процессу разрешено работать непрерывно не дольше некоторого времени, называемого квантом. Если процесс работал столько же или дольше, чем квант, то он помещается в конец очереди процессов с тем же приоритетом. Процесс с алгоритмом SCHED_RR, приостановленный процессом с большим приоритетом, возобновляя работу, использует остаток своего кванта.

3) SCHED_OTHER: стандартный алгоритм планировщика с разделением времени. Можно использовать только со значениями статического приоритета, равными нулю. Предназначенный для процессов, не требующих специальных механизмов реального времени со статическими приоритетами. Порядок предоставления процессорного времени процессам со статическим приоритетом,

равным нулю, основывается на **динамических приоритетах**, существующих только внутри этого списка. Динамический приоритет основан на уровне nice и увеличивается с каждым квантом времени, при котором процесс был готов к работе, но ему было отказано в этом планировщиком. Это приводит к тому, что, рано или поздно, всем процессам с приоритетом SCHED_OTHER выделяется процессорное время.

26. Инверсия приоритетов

В Linux инверсия приоритетов может возникать, когда задача с низким приоритетом блокирует ресурсы, необходимые для выполнения задачи с более высоким приоритетом. Это может произойти, например, когда задача с низким приоритетом запрашивает доступ к файлу или использует сетевые ресурсы, которые нужны для выполнения задачи с более высоким приоритетом.

Для предотвращения инверсии приоритетов в Linux используются различные механизмы. Например, планировщик задач в Linux использует приоритеты процессов и потоков для управления выполнением задач. Задачи с более высоким приоритетом получают больше времени на выполнение, чем задачи с более низким приоритетом.

Кроме того, в Linux используются механизмы управления блокировками, которые позволяют избежать блокировки выполнения задач с более высоким приоритетом. Например, для работы с файлами в Linux используется механизм блокировки файловых систем (File System Locking), который позволяет задачам блокировать только те части файла, которые им действительно нужны, минимизируя возможность блокировки выполнения задач с более высоким приоритетом.

Мьютексы реального времени доступны только тогда, когда ядро собрано с параметром CONFIG_RT_MUTEXES, что проверяем так:

```
# cat /boot/config-2.6.32.9-70.fc12.i686.PAE | grep RT_MUTEX
CONFIG_RT_MUTEXES=y
# CONFIG_DEBUG_RT_MUTEXES is not set
# CONFIG_RT_MUTEX_TESTER is not set
```

В отличие от регулярных мьютексов, мьютексы реального времени обеспечивают наследование приоритетов (priority inheritance, PI), что является одним из нескольких (немногих) известных способов, препятствующих возникновению инверсии приоритетов (priority inversion). Если RT мьютекс захвачен процессом A, и его пытается захватить процесс B (более высокого приоритета), то:

- процесс B блокируется и помещается в очередь ожидающих освобождения процессов wait_list (в описании структуры rt_mutex);
- при необходимости, этот список ожидающих процессов переупорядочивается в порядке приоритетов ожидающих процессов;
- приоритет владельца мьютекса (текущего выполняющегося процесса) B повышается до приоритета ожидающего процесса A (максимального приоритета из ожидающих в очереди процессов);
- это и обеспечивает избежание потенциальной инверсии приоритетов.

27. Методы IPC в Unix-подобных ОС: состав и общая характеристика каждого типа IPC (включая сетевые)

В Unix-подобных системах основными методами межпроцессорного взаимодействия являются:

1. Сигналы
2. Неименованные и именованные каналы
3. Очереди сообщений
4. Семафоры
5. Разделяемая память
6. Сокеты

1) Сигналы.

Сигналы позволяют осуществить самый примитивный способ коммуникации между двумя процессами. Сигналы в системе UNIX используются для того, чтобы: сообщить процессу о том, что возникло асинхронное событие; или необходимо обработать исключительное состояние.

Изначально сигналы были разработаны для уведомления об ошибках. В дальнейшем их стали использовать и как простейшую форму межпроцессного взаимодействия (IPC), например, для синхронизации процессов или для передачи простейших команд от одного процесса другому. Сигнал позволяет передать уведомление о некотором произошедшем событии между процессами или между ядром системы и процессами. Это означает, что посредством сигналов можно выполнять две основные функции IPC: передачу информации и синхронизацию процессов или потоков.

Для отправки и доставки сигнала требуется системный вызов. Для доставки – прерывание и его обработка. При этом требуется проведение довольно большого числа операций со стеком – копирование пользовательского стека в системную область, извлечение параметров и результатов работы системных вызовов и прерываний. Поскольку объем передаваемой информации при этом способе взаимодействия не велик, а затраты на его реализацию существенны, сигналы считаются одним из самых ресурсоемких способов IPC.

В Unix системах можно выделить три типа сигналов: надежные, ненадежные и сигналы реального времени (Real-time signals).

Надежные сигналы (reliable signals) гарантируют, что сигнал будет доставлен и обработан только один раз. При получении надежного сигнала ядро ожидает, пока получатель не обработает его, и затем продолжает работу. Например, сигнал SIGINT, отправляемый при нажатии сочетания клавиш Ctrl+C, является надежным, так как гарантирует, что процесс получит сигнал и будет прерван.

Ненадежные сигналы (unreliable signals) могут быть доставлены несколько раз и в случайном порядке, что может привести к проблемам обработки. Например, сигналы POSIX-стандарта SIGALRM, SIGIO и SIGURG являются ненадежными.

Сигналы реального времени (Real-time signals) — это надежный, асинхронный механизм межпроцессного взаимодействия, который позволяет передавать пользовательские данные с сигналом. В отличие от обычных сигналов, сигналы реального времени гарантируют доставку сигнала в порядке его отправки и могут быть использованы для передачи более сложной информации, например, вектора событий. Сигналы реального времени имеют номера в диапазоне от SIGRTMIN до SIGRTMAX, где SIGRTMIN и SIGRTMAX определяются в файле заголовков <signal.h>.

Отличие между тремя типами сигналов состоит в их поведении при доставке и обработке, а также в рекомендуемых областях применения.

2) Каналы

Различают два типа каналов анонимные (неименованные) и именованные. Они по - разному реализованы, но доступ к ним организуется одинаково с помощью обычных функций read и write. Одним из свойств программных каналов и FIFO является то, что данные по ним передаются в виде потоков байтов.

Неименованные каналы: являются однонаправленными и имеют ограниченную пропускную способность. Они создаются с помощью системного вызова pipe() и могут быть использованы только между процессами, которые являются родственными (хотя могут использоваться и неродственными процессами, если предоставить им возможность передавать друг другу дескрипторы). Чтобы использовать неименованный канал, необходимо создать его перед запуском дочернего процесса. Дочерний процесс наследует дескрипторы канала и может использовать их для чтения или записи данных. Главное применение неименованных каналов в ОС Unix – реализация конвейеров команд в интерпретаторах командной строки.

Именованные каналы: (FIFO) также являются однонаправленными, но имеют более широкий спектр применения. Они создаются с помощью системного вызова mkfifo() и могут быть использованы между процессами, которые не являются родственными. Именованный канал является файлом, который хранится в файловой системе. Процессы могут открывать этот файл как для чтения, так и для записи и обмениваться данными через него. FIFO может быть открыт либо только на чтение, либо только на запись. Нельзя открывать канал на чтение и запись одновременно, поскольку именованные каналы могут быть только односторонними. Несмотря на то, что именованные каналы являются отдельным типом файлов и могут быть видимы разными процессами даже в распределенной файловой системе,использование FIFO для взаимодействия удаленных процессов и обмена информацией между ними невозможно. Так как и в этом случае для передачи данных задействовано ядро. Создаваемый файл служит для получения данных о расположении FIFO в адресном пространстве ядра и его состоянии. Именованные каналы могут существовать на

протяжении бОльшего времени, в то время как неименованные каналы удаляются сразу после того, как все процессы закрыли свои дескрипторы на канал.

На неименованные каналы и каналы FIFO системой накладываются всего два ограничения: OPEN_MAX — максимальное количество дескрипторов, которые могут быть одновременно открыты некоторым процессом, PIPE_BUF — максимальное количество данных, для которого гарантируется атомарность операции записи

3) Очередь сообщений

Очередь сообщений находится в адресном пространстве ядра и имеет ограниченный размер. В отличие от каналов, которые обладают теми же самыми свойствами, очереди сообщений сохраняют границы сообщений. Это значит, что ядро ОС гарантирует, что сообщение, поставленное в очередь, не смешается с предыдущим или следующим сообщением при чтении из очереди. Кроме того, с каждым сообщением связывается его тип. Процесс, читающий очередь сообщений, может отбирать только сообщения заданного типа или все сообщения кроме сообщений заданного типа.

Очередь сообщений можно рассматривать как связный список сообщений. Каждое сообщение представляет собой запись, очереди сообщений автоматически расставляют границы между записями, аналогично тому, как это делается в дейтаграммах UDP. Для записи сообщения в очередь не требуется наличия ожидающего его процесса в отличие от неименованных каналов и FIFO, в которые нельзя произвести запись, пока не появится считывающий данные процесс. Поэтому процесс может записать в очередь какие-то сообщения, после чего они могут быть получены другим процессом в любое время, даже если первый завершит свою работу. С завершением процесса-источника данные не исчезают (данные, остающиеся в именованном или неименованном канале, сбрасываются, после того как все процессы закроют его).

Следует заметить, что, к сожалению, не определены системные вызовы, которые позволяют читать сразу из нескольких очередей сообщений, или из очередей сообщений и файловых дескрипторов. Видимо, отчасти и поэтому очереди сообщений широко не используются.

4) Семафоры и разделяемая память

Семафоры - это механизм синхронизации в Unix-подобных операционных системах, который позволяет контролировать доступ к разделяемым ресурсам (например, критические секции кода или общую память) в многопроцессном окружении.

Семафоры могут работать в разных режимах:

- Бинарарные (0 или 1), рассматриваются как флаги при создании семафоров и обычно используются для разделения доступа к ресурсам.
- Счетные, имеют числовое значение и использоваться для контроля количества доступов к ресурсам.

Семафоры реализуются с помощью системных вызовов: semget(), semop() и semctl(). Функция semget() создает массив семафоров или возвращает идентификатор существующего массива. Функция semop() позволяет изменять значения семафоров в массиве, путем блокировки, разблокировки или ожидания. Функция semctl() используется для управления семафорами в массиве, например, для удаления массива семафоров.

Применение семафоров может свести к минимуму конфликты и задержки при доступе к разделяемым ресурсам, что делает их полезными в разработке многопоточных и распределенных систем.

Разделяемая память (Shared memory) - это механизм в Unix-подобных операционных системах, который позволяет нескольким процессам обмениваться данными через общую область памяти. Этот механизм позволяет исключить затраты на передачу больших объемов данных между процессами и ускорить обмен данными.

В Unix-подобных операционных системах механизм разделяемой памяти реализуется с помощью системных вызовов shmget(), shmat(), shmdt() и shmctl().

Процедура создания общей памяти включает в себя следующие шаги:

1. Создание сегмента общей памяти с помощью системного вызова shmget().
2. Присоединение сегмента общей памяти к адресному пространству процесса с помощью системного вызова shmat().
3. Работа с общей памятью, как с обычной областью памяти.
4. Отсоединение сегмента общей памяти от процесса с помощью системного вызова shmdt().
5. Удаление сегмента общей памяти с помощью системного вызова shmctl().

Разделяемая память может использоваться для обмена большими объемами данных между процессами, обмена данными между процессами в режиме реального времени, реализации межпроцессного взаимодействия. Однако, использование общей памяти также имеет свои риски, например, возможность несогласованного доступа разных процессов к памяти или трудно обнаруживаемых ошибок в программе, такие как гонки данных. Поэтому использование разделяемой памяти требует тщательной проработки и тестирования работы программы.

6) Сокеты

Программисты в Unix-подобных операционных системах используют сокеты (sockets) для реализации механизмов межпроцессного взаимодействия. Сокеты - это механизм, который позволяет обеспечивать связь между двумя процессами через сеть или локальный компьютер.

Сокеты в Unix-подобных операционных системах описываются файлом дескриптора (file descriptor). Как правило, созданием, открытием, закрытием, отправкой и получением сообщений сокетов занимается прикладной уровень. Однако, некоторые функции низкоуровневых API управляют своими собственными сокетами.

Сокеты могут использоваться для реализации различных протоколов - TCP (Transmission Control Protocol), UDP (User Datagram Protocol) и т.д. Каждый протокол имеет свои собственные характеристики, сильные и слабые стороны.

Для работы с сокетами, в Unix-подобных системах используют различные функции, такие как send(), recv() и многие другие. Большинство из них находятся в стандартной библиотеке языка программирования C и доступны из-под любой операционной системы.

Сокеты позволяют программистам решать различные задачи, связанные с межпроцессным взаимодействием, например, создавать серверные приложения, взаимодействовать с веб-серверами, программно общаться по протоколу TCP/IP с другими компьютерами, управлять сетевыми соединениями и т.д.

28. Обработка событий

Обработка событий в операционной системе - это процесс обработки событий, которые происходят в операционной системе. События могут быть вызваны пользователем или программой. Обработка событий может включать в себя запись информации о событии в журнале событий, запуск другой программы или выполнение других действий. Это может быть любое событие, связанное с работой компьютера: мышь была нажата, клавиатура была нажата, окно было открыто, устройство было подключено, и т.д.

Обработка событий может быть реализована различными способами в зависимости от конкретной операционной системы. В Windows, например, обработка событий осуществляется через цикл обработки сообщений. В Linux обработка событий осуществляется через систему файловых дескрипторо

Обработка событий в операционной системе Linux осуществляется с использованием механизмов ядра и различных служб операционной системы. Вот общая схема работы обработки событий в Linux:

1) Ядро операционной системы: Ядро Linux отвечает за управление всеми ресурсами компьютера, включая процессор, память, диски и устройства ввода-вывода. Ядро также отвечает за планирование выполнения процессов и управление событиями в системе

2) Прерывания: Прерывания – это механизм, позволяющий устройствам (например, клавиатуре, мыши, сетевой карте) прерывать обычный ход выполнения программы и привлекать внимание ядра. Когда устройство генерирует прерывание, ядро обрабатывает его и вызывает соответствующий обработчик прерывания, который выполняет необходимые действия

3) Обработчики прерываний: Обработчики прерываний – это часть ядра Linux, которая вызывается в ответ на прерывание устройства. Обработчики прерываний выполняются с высоким приоритетом и обрабатывают прерывание, выполняя необходимые действия, например, чтение данных с устройства или запись данных на диск

4) Планировщик: Планировщик ядра отвечает за управление выделением процессорного времени между различными процессами. Когда процесс завершает выполнение или происходит событие, требующее переключения контекста, планировщик выбирает следующий процесс для выполнения и переключает контекст выполнения на него

5) Сигналы: Сигналы - это механизм взаимодействия между процессами и ядром. Процессы или ядро могут отправлять сигналы другим процессам для уведомления о различных событиях, таких как завершение процесса, нажатие клавиши Ctrl+C и т.д. Процессы могут устанавливать обработчики сигналов для обработки событий, связанных с сигналами

6) Сокеты и файловые дескрипторы: Linux предоставляет механизмы сокетов и файловых дескрипторов для обработки событий ввода-вывода. Процессы могут ожидать событий ввода-вывода из сокетов или файловых дескрипторов и блокироваться до наступления события. Когда событие происходит, процесс разблокируется и может обработать событие

Обработка событий в Linux включает в себя сложные механизмы и подсистемы, которые обеспечивают эффективную и отзывчивую работу операционной системы. Это лишь общая схема работы, и внутри каждой подсистемы существуют более подробные механизмы и алгоритмы обработки событий

29. Синхронизация при разделении ресурсов, средства синхронизации

Цели и средства синхронизации

Потребность в синхронизации потоков возникает в мультипрограммной операционной системе и связана с совместным использованием аппаратных и информационных ресурсов вычислительной системы. Синхронизация необходи-ма для исключения гонок и тупиков при обмене данными между потоками, разделении данных, при доступе к процессору и устройствам ввода-вывода.

Средства синхронизации называются - *Inter Process Communications (IPC)*

Все потоки в общем случае протекают независимо, асинхронно друг другу.

Любое взаимодействие процессов или потоков связано с их синхронизацией, ко-торая заключается в согласовании их скоростей путем приостановки потока до наступления некоторого события и последующей его активизации при наступ-лении этого события. Например, поток-получатель должен обращаться за данными только после того, как они помещены в буфер потоком-отправителем. Если же поток-получатель обратился к данным до момента их поступления в буфер, то он должен быть приостановлен.

Для синхронизации потоков прикладных программ программист может исполь-зовать как собственные средства и приемы синхронизации, так и средства опера-ционной системы.

Обычно разработчики операционных систем предоставляют в распоряжение прикладных и системных программистов широкий спектр средств синхронизации. Эти средства могут образовывать иерархию, когда на основе более простых средств строятся более сложные, а также быть функционально специализированными, например средства для синхронизации потоков одного процесса, средства для синхронизации потоков разных процессов при обмене данными и т. д. Часто функциональные возможности разных системных вызовов синхронизации пере-крываются, так что для решения одной задачи программист может воспользо-ваться несколькими вызовами в зависимости от своих личных предпочтений.

Критическая секция

Критическая секция – это часть программы, результат выпол-нения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программы, изменяются другими потоками в то время, когда вы-полнение этой части еще не завершено.

Чтобы исключить эффект гонок по отношению к критическим данным, необхо-димо обеспечить, чтобы в каждый момент времени в критической секции свя-занной с этими данными, находился только один поток. При этом неважно, находится этот поток в активном или в приостановленном состоянии. Этот прием называют взаимным исключением. Операционная система использует разные способы реализации взаимного исключения.

Блокирующие переменные

Для синхронизации потоков одного процесса прикладной программист может использовать *глобальные блокирующие переменные*. С этими переменными, к ко-торым все потоки процесса имеют прямой доступ, программист работает, не об-ращаясь к системным вызовам ОС.

Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает. На рисунке 3.11 показан фрагмент алго-ритма потока, использующего для реализации взаимного исключения доступа к критическим данным D блокирующую переменную F(D). Перед входом в критическую секцию поток проверяет, не работает ли уже какой-нибудь поток с данными D. Если переменная F(D) установлена в 0, то данные заняты и проверка циклически повторяется. Если же данные свободны (F(D) = 1), то значение пере-менной F(D) устанавливается в 0 и поток входит в критическую секцию. После того как поток выполнит все действия с данными D, значение переменной F(D) снова устанавливается равным 1.

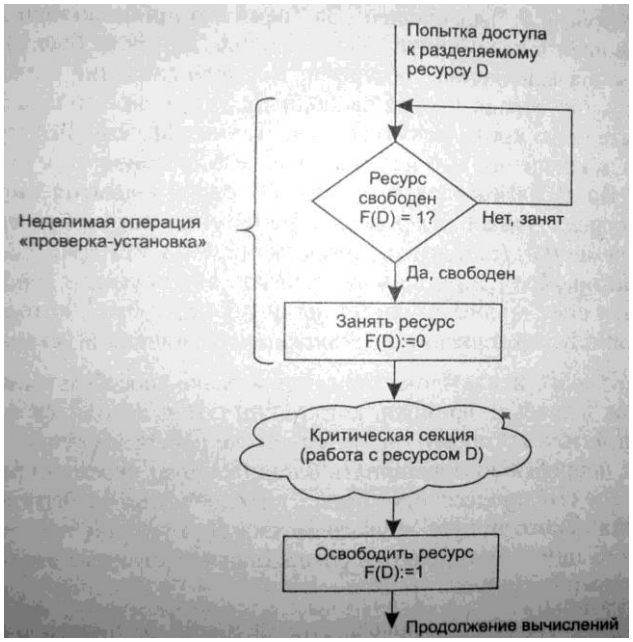


Рисунок 3.11 – Реализация критических секций с использованием блокирующих переменных.

Если все потоки написаны с учетом вышеописанных соглашений, то взаимное исключение гарантируется. При этом потоки могут быть прерваны операционной системой в любой момент и в любом месте, в том числе в критической секции.

Нельзя прерывать поток между выполнением операций проверки и установки блокирующей переменной. Пусть в результате проверки перемен-ной поток определил, что ресурс свободен, но сразу после этого, не успев уста-новить переменную в 0, был прерван. За время его приостановки другой поток занял ресурс, вошел в свою критическую секцию, но также был прерван, не за-вершив работы с разделяемым ресурсом. Когда управление было возвращено первому потоку, он, считая ресурс свободным, установил признак занятости и начал выполнять свою критическую секцию. Таким образом, был нарушен прин-цип взаимного исключения, что потенциально может привести к нежелательным последствиям. Во избежание таких ситуаций в системе команд многих компьютеров предусмотрена единая, неделимая команда анализа и присвоения значения логической переменной (например, команды BTC, BTR и BTS процессора Pentium). При отсутствии такой команды в процессоре соответствующие действия должны реализовываться специальными системными примитивами (Примитив – базовая функция ОС), которые бы запре-щали прерывания на протяжении всей операции проверки и установки.

Но при таком способе имеется существенный недостаток: в течение времени, когда один поток находится в критической секции, другой поток, которому требуется тот же ресурс, получив доступ к про-цессору, будет непрерывно опрашивать блокирующую переменную, бесполезно тратя выделяемое ему процессорное время, которое могло бы быть использовано для выполнения какого-нибудь другого потока. Для устранения этого недостат-ка во многих ОС предусматриваются специальные системные вызовы для рабо-ты с критическими секциями.

Семафоры

Семафоры Дийкстры – обобщение блокирующих переменных. Вместо двоичных переменных Дийкстра (Dijkstra) предложил использовать переменные, которые могут принимать целые неотрицательные значения. Такие переменные, используемые для синхронизации вычислительных процессов, получили название семафоров.

Для работы с семафорами вводятся два примитива, традиционно обозначаемых P и V. Пусть переменная S представляет собой семафор. Тогда действия V(S) и P(S) определяются следующим образом:

1) V(S): переменная S увеличивается на 1 единым действием. Выборка, наращивание и запоминание не могут быть прерваны. К переменной S нет доступа другим потокам во время выполнения этой операции;

2) P(S): уменьшение S на 1, если это возможно. Если S = 0 и невозможно уменьшить S, оставаясь в области целых неотрицательных значений, то в этом случае поток, вызывающий операцию P, ждет, пока это уменьшение станет возможным. Успешная проверка и уменьшение также являются неделимой операцией.

Никакие прерывания во время выполнения примитивов V и P недопустимы.

Семафоры позволяют эффективно решать задачу синхронизации доступа к ресурсным пулам.

Семафор может использоваться и в качестве блокирующей переменной. Обеспечим взаимное исключение с помощью двоичного семафора b, что показано на рисунке 3.12. Оба потока после проверки доступности буферов должны выполнить проверку доступности критической секции.

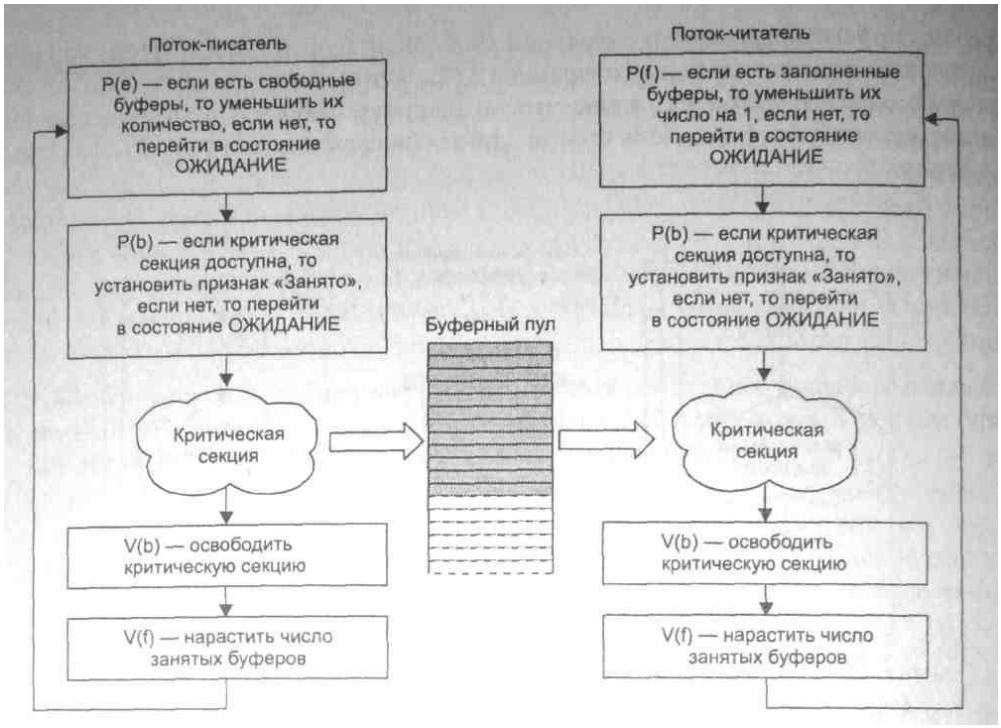


Рисунок 3.12 – Использование двоичного семафора.

Тупики

Приведенный выше пример позволяет также проиллюстрировать еще одну проблему синхронизации — *взаимные блокировки*, называемые также дедлоками (deadlocks), клинчами (clinch), или тупиками. Покажем, что если переставить местами операции P(e) и P(b) в потоке-писателе, то при некотором стечении обстоя-тельств эти два потока могут взаимно блокировать друг друга.

Итак, пусть поток-писатель начинает свою работу с проверки доступности критической секции — операции P(b), и пусть он первым войдет в критическую секцию.

Выполняя операцию P(e), он может обнаружить отсутствие свободных бу-феров и перейти в состояние ожидания. Из этого состоя-ния его может вывести только поток-читатель, который возьмет очередную за-пись из буфера. Но поток-читатель не сможет этого сделать, так как для этого ему потребуется войти в критическую секцию, вход в которую заблокирован по-током-писателем. Таким образом, ни один из этих потоков не может завершить начатую работу и возникнет тупиковая ситуация, которая не может разрешиться без внешнего воздействия.

В рассмотренном примере тупик был образован двумя потоками, но взаимно блокировать друг друга может и большее число потоков. На рисунке 3.13 показано такое распределение ресурсов Ri между несколькими потоками Tj, которое при-вело к возникновению взаимных блокировок. Стрелки обозначают потребность потока в ресурсах. Например, потоку T1 для выполнения работы необходимы ресурсы R1 и R2, из ко-торых выделен только один — R1, а ресурс R2 удерживается потоком T2. Ни один из четырех показанных на рисунке потоков не может продолжить свою работу, так как не имеет всех необходимых для этого ресурсов.

Невозможность потоков завершить начатую работу из-за возникновения взаимных блокировок снижает производительность вычислительной системы. По-этому проблеме предотвращения тупиков уделяется большое внимание. На тот случай, когда взаимная блокировка все же возникает, система должна предоста-вить администратору-оператору средства, с помощью которых он смог бы распо-знать тупик, отличить его от обычной блокировки из-за временной недоступности ресурсов. Если тупик диагностирован, то нужны средства для снятия взаимных блокировок и восстановления нормального вычислительного процесса.

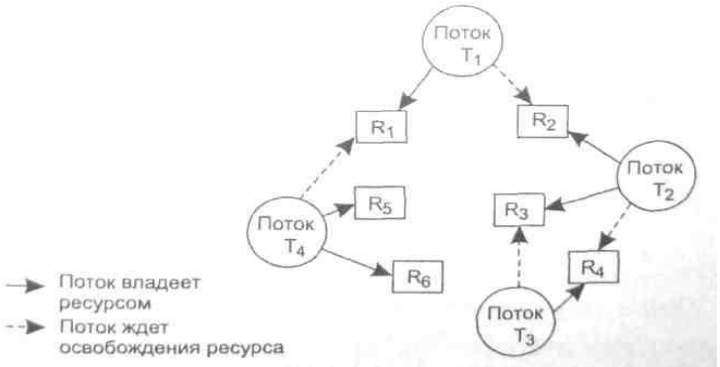


Рисунок 3.13 – Взаимная блокировка нескольких потоков.

Если тупиковая ситуация образована множеством по-токов, занимающих массу ресурсов, распознавание тупика является нетривиаль-ной задачей. Существуют формальные, программно-реализованные методы рас-познавания тупиков, основанные на ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам. Анализ этих таблиц позволяет обнаружить взаимные блокировки.

Если же тупиковая ситуация возникла, то не обязательно снимать с выполнения все заблокированные потоки. Можно совершить «откат» некоторых потоков до так называемой контроль-ной точки, в которой запоминается вся информация, необходимая для восстановле-ния выполнения программы с данного места. Контрольные точки расставляются в программе в тех местах, после которых возможно возникновение тупика.

Синхронизирующие объекты ОС

Примерами синхронизирующих объектов ОС являются *системные сема-форы, мьютексы, события, таймеры и другие* – их набор зависит от конкретной ОС, которая создает эти объекты по запросам процессов.

Для синхронизации могут быть использованы такие «обычные» объ-емы ОС, как файлы, процессы и потоки. Все эти объекты могут находиться в двух состояниях: *сигнальном* и *несигнальном*. Для каждого объекта смысл, вкладываемый в понятие «сигнальное состояние», зависит от типа объекта.

Потоки с помощью специального системного вызова сообщают операционной системе о том, что они хотят синхронизировать свое выполнение с состоянием некоторого объекта. Будем далее называть этот системный вызов Wa1t (X), где X – указатель на объект синхронизации. Системный вызов, с помощью которого поток может перевести объект синхронизации в сигнальное состояние, назовем Set (X).

Поток, выполнивший системный вызов Wait (X), переводится ОС в состояние ожидания до тех пор, пока объект X не перейдет в сигнальное состояние.

Поток может ожидать установки сигнального состояния не одного объекта, а нескольких. При этом поток может попросить ОС активизировать его при установ-ке либо одного из указанных объектов, либо всех объектов. Поток может в каче-стве аргумента системного вызова Wait () указать также максимальное время, которое он будет ожидать перехода объекта в сигнальное состояние, после чего ОС должна его активизировать в любом случае. Может случиться, что установки некоторого объекта в сигнальное состояние ожидают сразу несколько потоков. В зависимости от объекта синхронизации в состояние готовности могут перево-диться либо все ожидающие это событие потоки, либо один из них.

Синхронизация тесно связана с планированием потоков:

1) любое обращение потока с системным вызовом Wa1t (X) влечет за собой действия, в подсистеме планирования – этот поток снимается с выполнения и помещается в очередь ожидающих потоков, а из очереди готовых потоков выбирается и активизируется новый поток;

2) при переходе объекта в сигнальное состояние ожидающий этот объект поток переводится в очередь готовых к выполнению потоков.

В обоих случаях осуществляется перепланиро-вание потоков, при этом если в ОС предусмотрены изменяемые приоритеты и / или кванты времени, то они пересчитывают по правилам, принятым в этой ОС.

Мьютекс, как и семафор, обычно используется для управления доступом к данным. Объект-мьютекс «освобождает» из очереди ожидающих только один поток.

Работа мьютекса хорошо поясняется в терминах «владения». Пусть поток, который, пытаясь получить доступ к критическим данным, выполнил системный вы-зов Wait(X), где X — указатель на мьютекс. Предположим, что мьютекс находится в сигнальном состоянии, в этом случае поток тут же становится его владельцем, устанавливая его в несигнальное состояние, и входит в критическую секцию. После того как поток выполнил работу с критическими данными, он «отдает» мьютекс, устанавливая его в сигнальное состояние. В этот момент мьютекс сво-боден и не принадлежит ни одному потоку. Если какой-либо поток ожидает его освобождения, то он становится следующим владельцем этого мьютекса, одно- временно мьютекс переходит в несигнальное состояние.

Объект-файл, переход которого в сигнальное состояние соответствует завершению операции ввода-вывода с этим файлом, используется в тех случаях, когда поток, инициировавший эту операцию, решает дождаться ее завершения, прежде чем продолжить свои вычисления.

Объект-событие обычно исполь-зуется для того, чтобы оповестить другие потоки о том, что некоторые действия завершены.

Сигналы

Сигнал дает возможность задаче реагировать на событие, источником которого может быть операционная система или другая задача. Сигналы вызывают прерывание задачи и выполнение заранее предусмотренных действий.

Сигналы мо-гут вырабатываться:

1) *синхронно,* то есть как результат работы самого процесса. Чаще всего приходят от системы прерываний процессора и свидетельствуют о действиях процесса, блокируемых аппаратурой, например, деление на нуль, ошибка адресации, нарушение защиты памяти и т. д.;

2) *асинхронно*, то есть направлены процессу другим процессом. Например, сигнал с терминала. Во многих ОС предусматривается оперативное снятие процесса с

выполнения. Для этого пользователь может нажать некоторую комбинацию клавиш (Ctrl+C, Ctrl+Break), в ре-зультате чего ОС вырабатывает сигнал и направляет его активному процессу. Сигнал может поступить в любой момент выполнения процесса, требуя от процесса немедленного завершения работы. В дан-ном случае реакцией на сигнал является безусловное завершение процесса.

В системе может быть определен набор сигналов.

Сигналы обеспечивают логическую связь между процессами, а также между процессами и пользователями (терминалами). Поскольку посылка сигнала предусматривает знание идентификатора процесса, то взаимодействие посредством сигналов возможно только между родственными процессами, которые могут получить данные об идентификаторах друг друга.

В распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, блокирующие переменные, семафоры, сигналы и другие аналогичные средства, основанные на разделяемой памяти, оказываются непригодными. В таких системах синхронизация может быть реализована только посредством *обмена сообщениями.*

30. Типы памяти. Виды организации виртуальной памяти

По назначению:

- **Буферная память** — память, предназначенная для временного хранения данных при обмене ими между различными устройствами или программами.
- **Временная (промежуточная) память** — память для хранения промежуточных результатов обработки.
- **Кеш-память** — часть архитектуры устройства или программного обеспечения, осуществляющая хранение часто используемых данных для предоставления их в более быстрый доступ, нежели кешируемая память.
- **Корректирующая память** — часть памяти ЭВМ, предназначенная для хранения адресов неисправных ячеек основной памяти. Также используются термины relocation table и remap table.
- **Управляющая память** — память, содержащая управляющие программы или микропрограммы. Обычно реализуется в виде ПЗУ.
- **Разделяемая память или память коллективного доступа** — память, доступная одновременно нескольким пользователям, процессам или процессорам.

По организация адресного пространства:

- **Реальная или физическая память** — память, способ адресации которой соответствует физическому расположению её данных;

- **Виртуальная память** — память, способ адресации которой не отражает физического расположения её данных;

- **Оверлейная память** — память, в которой присутствует несколько областей с одинаковыми адресами, из которых в каждый момент доступна только одна.

По удалённость и доступность для процессора:

- **Первичная память** (сверхоперативная, СОЗУ) — доступна процессору без какого-либо обращения к внешним устройствам. Данная память отличается крайне малым временем доступа и тем, что недоступна для явного использования в программе.
- **Вторичная память** — доступна процессору путём прямой адресацией через шину адреса (адресуемая память). Таким образом доступна оперативная память (память, предназначенная для хранения текущих данных и выполняемых

программ) и порты ввода-вывода (специальные адреса, через обращение к которым реализовано взаимодействие с прочей аппаратурой).

• **Третичная память** — доступна только путём нетривиальной последовательности действий. Сюда входят все виды внешней памяти — доступной через устройства ввода-вывода. Взаимодействие с третичной памятью ведётся по определённым правилам (протоколам) и требует присутствия в памяти соответствующих программ. Программы, обеспечивающие минимально необходимое взаимодействие, помещаются в ПЗУ, входящее во вторичную память .

По доступности техническими средствами:

• **Непосредственно управляемая** (оперативно доступная) память— память, непосредственно доступная в данный момент.

• **Автономная память** — память, доступ к которой требует внешних действий — например, вставку оператором архивного носителя с указанным программой идентифиткатором

• **Полуавтономная память** — то же, что автономная, но физическое перемещение носителей осуществляется роботом по команде системы, то есть не требует присутствия оператора

Виды организации виртуальной памяти:

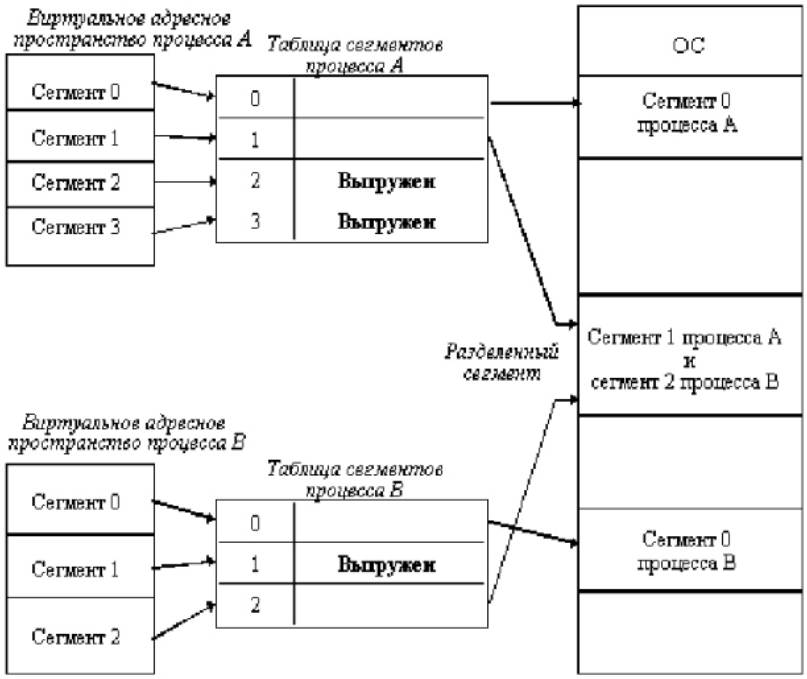
- ✓ Страничная
- ✓ Сегментная

Страничная память — способ организации виртуальной памяти, при котором единицей отображения виртуальных адресов на физические является регион постоянного размера — страница.

При использовании страничной модели вся виртуальная память делится на N страниц таким образом, что часть виртуального адреса интерпретируется как номер страницы, а часть — как смещение внутри страницы. Вся физическая память также разделяется на блоки такого же размера — фреймы. Таким образом в один фрейм может быть загружена одна страница. Свопинг — это

выгрузка страницы из памяти на диск (или другой носитель большего объема), который используется тогда, когда все фреймы заняты. При этом под свопинг попадают страницы памяти неактивных на данный момент процессов.

Сегментная организация виртуальной памяти реализует следующий механизм: вся память делиться на сегменты фиксированной или произвольной длины, каждый из которых характеризуется своим начальным адресом — базой или селектором. Виртуальный адрес в такой системе состоит из 2-х компонент: базы сегмента, к которому мы хотим обратиться, и смещения внутри сегмента. Физический адрес вычисляется по формуле:
addr = base + offset



31. Управление памятью, распределенной динамическими разделами

Распределение динамическими разделами

При распределении динамическими разделами память выделяется по запросам. В запросе содержится объем требуемого участка памяти.

Запросы выдаются программой управления процессами при создании очередного процесса.

Система управления памятью при удовлетворении запроса находит свободный участок требуемого объема и создает раздел процесса.

При завершении процесса этот раздел уничтожается.

Реализация функций

При способе управления памятью, распределяемой динамическими разделами, необходимо учитывать свободные и занятые участки, т.е. хранить информацию о базовых адресах и размерах этих участков памяти. Во многих ОС используется способ учета с помощью списка блоков управления памятью.

В системной области памяти SV содержится адрес начала списка. В начало каждого свободного участка помещается блок управления памятью МСВ. В простейшем случае этот блок должен содержать адрес следующего свободного участка и объем участка, которым он управляет. МСВ последнего свободного участка содержит 0 в поле адреса следующего участка.

С этим списком работает программа ядра, занимающаяся выделением памяти. Для освобождения памяти необходимо знать адрес и размер занятого участка. Эта информация хранится в блоке управления процессом. При уничтожении процесса освобождается занятый участок, и эта информация используется для реорганизации списка блоков свободной памяти.

Планирование запросов на выделение памяти осуществляется, так же как и при распределении статическими разделами. Программы планирования процессов или системный загрузчик выдают запрос для очередного созданного или загружаемого процесса. Объем памяти предоставляемый по запросу, должен быть не менее запрашиваемого объема.

Алгоритмы программного обеспечения

При выделении памяти необходимо просмотреть список МСВ и найти подходящий участок. Наиболее популярны следующие алгоритмы просмотра.

Алгоритм "первый подходящий"

Выбор первого свободного участка, размер которого больше требуемого объема. В этом случае выбранный свободный участок делится и оставшаяся часть свободного участка включается в список на место выбранного свободного участка. Таким образом, крупные свободные участки сдвигаются в конец списка.

С одной стороны, это положительный момент, так как запросы на малые участки удовлетворяются в одной области памяти, а на большие - в другой. Это уменьшает дробление памяти на небольшие участки, уменьшает фрагментацию.

С другой стороны, увеличивается время поиска свободного участка, так как в начале списка группируются участки малого размера. Кроме того, существует опасность, что оставшаяся свободная память от выбранного участка будет настолько мала, что не удовлетворит никакому запросу.

В этом случае целесообразно установить некоторый порог, и в случае, если размер оставшегося свободного участка меньше этого порогового значения, то следует занять весь выбранный участок. Чтобы участки малого размера не группировались в начале списка, список можно закольцевать, а указатель на начало списка перемешать, т.е. после выделения свободного участка указатель содержит адрес следующего свободного участка. Просмотр при удовлетворении следующего запроса начнется с другого элемента списка.

Алгоритм "самый подходящий"

Выбор свободного участка наименьшего размера, удовлетворяющего запросу.

В этом случае можно использовать такой же алгоритм работы со списком, как и алгоритм "первый подходящий", но список должен быть упорядочен по возрастанию размеров свободных участков. При подобной организации списка первый найденный свободный участок и будет самым подходящим.

Достоинства:

1) Если существует свободный участок, размер которого в точности соответствует размеру запроса, то этот участок будет выбран.

2) При использовании данного алгоритма свободные области больших размеров остаются нетронутыми.

Недостатки:

1) Увеличение вероятности получения маленьких остатков и, как следствие, увеличение дробления памяти.

2) При использовании упорядоченного по возрастанию списка необходимо просматривать список для поиска места, куда должен быть вставлен новый образовавшийся свободный блок.

Достоинства и недостатки распределения динамическими разделами

Достоинства:

1) Возможность организации мультипрограммного режима работы

Недостатки:

1) Усложнение алгоритмов управления и системных структур данных

2) Память используется неэффективно по причине загрузки редко используемого логического адресного пространства

3) Фрагментация ОП

32. Механизмы преобразования виртуального адреса в физический при различных организациях памяти

Механизмы преобразования виртуального адреса в физический адрес при различных организациях памяти могут отличаться, но общий принцип заключается в том, что процессор преобразует виртуальный адрес (вычисленный программой) в физический адрес в памяти. Это необходимо для того, чтобы процессор мог обратиться к нужной ячейке памяти для чтения или записи данных.

Рассмотрим несколько организаций памяти и их механизмы преобразования виртуального адреса.

1. Организация памяти с постоянным размещением (Fixed partition memory organization). В этой организации память разбивается на фиксированные блоки, каждый из которых имеет свой физический адрес. Виртуальный адрес в данном случае состоит из номера раздела (partition number) и смещения (offset). Для преобразования виртуального адреса в физический процессор использует формулу физический адрес = базовый адрес + смещение. Базовый адрес для каждого раздела можно сохранить в таблице фиксированных разделов (Fixed Partition Table).

2. Организация памяти с динамическим размещением (Dynamic partition memory organization). В этой организации память разбивается на блоки произвольного размера, которые могут быть использованы для размещения программ и данных. Для управления блоками в памяти используется таблица страниц (Page table), которая содержит информацию о каждом блоке в памяти, включая его физический адрес и состояние (занят/свободен). Виртуальный адрес в данном случае состоит из номера страницы (page number) и смещения (offset). Для преобразования виртуального адреса в физический процессор использует таблицу страниц, которая содержит соответствие между виртуальными и физическими адресами блоков в памяти.

3. Организация памяти сегментированной системой (Segmented memory organization). В этой организации память разбивается на разделы (сегменты), которые могут содержать код программы, данные и стек вызовов функций. Каждый сегмент имеет свой физический адрес. Виртуальный адрес в данном случае состоит из номера сегмента и смещения внутри сегмента. Процессор использует таблицу сегментов (Segment table), которая хранит информацию о каждом сегменте, включая его начальный адрес и размер. Для преобразования виртуального адреса в физический, процессор использует таблицу сегментов, чтобы определить физический адрес сегмента, и затем добавляет к нему смещение внутри сегмента.

Каждая из этих организаций памяти имеет свои преимущества и недостатки, и выбор зависит от требований к производительности, экономичности и безопасности.

33. Базовые сервисы диспетчера виртуальной памяти

Система виртуальной памяти в Linux поддерживает адресное пространство, видимое каждому процессу: она создает страницы виртуальной памяти по требованию и управляет загрузкой этих страниц с диска или откачкой их обратно на диск, если требуется.

Менеджер виртуальной памяти поддерживает две точки зрения на адресное пространство каждого процесса:

- **Логическую** – поддержка команд управления адресным пространством. Адресное пространство рассматривается как совокупность непересекающихся смежных областей.
- **Физическую** – с помощью таблицы страниц для каждого процесса.

Для управления виртуальной памятью используются:

- **Файл откачки (backing store),** описывающий, откуда берутся страницы для заданного региона; регионы обычно поддерживаются либо файлом, либо не поддерживаются вообще **(память, обнуляемая по требованию)**
- Реакция региона на запись (совместное использование страниц или копирование при записи - COW).

Ядро создает новое виртуальное адресное пространство:

- Когда процесс запускает новую программу системным вызовом exec;
- При создании нового процесса системным вызовом fork.

При исполнении новой программы процессу предоставляется новое, пустое адресное пространство; процедуры загрузки программ наполняют это адресное пространство регионами виртуальной памяти.

Создание нового процесса с помощью fork включает создание полной копии адресного пространства существующего процесса.

Ядро копирует дескрипторы доступа к виртуальной памяти родительского процесса, затем создает новый набор таблиц страниц для дочернего процесса.

Таблицы страниц процесса-родителя копируются непосредственно в таблицы страниц дочернего, причем счетчик ссылок на каждую страницу увеличивается.

После исполнения fork родительский и дочерний процесс используют одни и те же физические страницы в своих виртуальных адресных пространствах.

Система управления страницами откачивает страницы физической памяти на диск, если они требуются для какой-либо другой цели.

Система управления страницами делится на две части:

- Алгоритм откачки, который определяет, какие страницы и когда откачать на диск;
- Механизм подкачки фактически выполняет передачу и подкачивает данные обратно в физическую память, если требуется.

Ядро Linux резервирует постоянный, зависящий от архитектуры регион виртуального адресного пространства каждого процесса для его собственного внутреннего использования.

Эта область виртуальной памяти ядра содержит два региона:

- Статическую область, содержащую ссылки из таблицы страниц на каждую доступную физическую страницу памяти в системе, так что используется простая трансляция физического адреса в виртуальный при исполнении кода ядра.
- Остаток зарезервированной части не используется ни для какой другой цели; его элементы таблицы страниц могут быть модифицированы и указывать на любые страницы в памяти.

34. Принцип кэширования данных. Стратегия подкачки страниц. Свопинг.

Кэш-память - это способ организации совместного функционирования двух типов запоминающих устройств, отличающихся временем доступа и стоимостью хранения данных, который позволяет уменьшить среднее время доступа к данным за счет динамического копирования в "быстрое" ЗУ наиболее часто используемой информации из "медленного" ЗУ. Кэш-памятью часто называют не только способ организации работы двух типов запоминающих устройств, но и одно из устройств - "быстрое" ЗУ. Оно стоит дороже и, как правило, имеет сравнительно небольшой объем. Важно, что механизм кэш-памяти является прозрачным для пользователя, который не должен сообщать никакой информации об интенсивности использования данных и не должен никак участвовать в перемещении данных из ЗУ одного типа в ЗУ другого типа, все это делается автоматически системными средствами.

В системах, оснащенных кэш-памятью, каждый запрос к оперативной памяти выполняется в соответствии со следующим алгоритмом:

- Просматривается содержимое кэш-памяти с целью определения, не находятся ли нужные данные в кэш-памяти; кэш-память не является адресуемой, поэтому поиск нужных данных осуществляется по содержимому - значению поля "адрес в оперативной памяти", взятому из запроса.
- Если данные обнаруживаются в кэш-памяти, то они считываются из нее, и результат передается в процессор.
- Если нужных данных нет, то они вместе со своим адресом копируются из оперативной памяти в кэш-память, и результат выполнения запроса передается в процессор. При копировании данных может оказаться, что в кэш-памяти нет свободного места, тогда выбираются данные, к которым в последний период было меньше всего обращений, для вытеснения из кэш-памяти. Если вытесняемые данные были модифицированы за время нахождения в кэш-памяти, то они переписываются в оперативную память. Если же эти данные не были модифицированы, то их место в кэш-памяти объявляется свободным.

На практике в кэш-память считывается не один элемент данных, к которому произошло обращение, а целый блок данных, это увеличивает вероятность так называемого "попадания в кэш", то есть нахождения нужных данных в кэш-памяти.

В реальных системах вероятность попадания в кэш составляет примерно 0,9. Высокое значение вероятности нахождения данных в кэш-памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

• *Пространственная локальность*. Если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам.

• *Временная локальность*. Если произошло обращение по некоторому адресу, то следующее обращение по этому же адресу с большой вероятностью произойдет в ближайшее время.

Подкачка страниц - один из механизмов виртуальной памяти, при котором отдельные фрагменты памяти (обычно неактивные) перемещаются из ОЗУ во вторичное хранилище (жёсткий диск или другой внешний накопитель, такой как флеш-память), освобождая ОЗУ для загрузки других активных фрагментов памяти. Такими фрагментами в современных ЭВМ являются страницы памяти.

Описание алгоритма **свопинга** можно разбить на три части: управление пространством на устройстве выгрузки, выгрузка процессов из основной памяти и подкачка процессов в основную память.

Стратегия считывания определяет, когда надо перемещать страницу с диска в память. Можно пытаться загрузить страницы, которые потребуются процессу, до того как он их запросит (заранее). А можно использовать стратегию подкачки по запросу, в этом случае страница загружается в память только тогда, происходит страничная ошибка.

При возникновении страничной ошибки система виртуальной памяти должна определить, в какое место физической памяти следует загрузить эту виртуальную страницу. Здесь начинает действовать стратегия размещения.

Если страничная ошибка происходит, когда вся физическая память заполнена, то применяется стратегия замещения. Она определяет, какую страницу нужно извлечь из памяти, чтобы освободить место для новой страницы.

35. Совместное использование памяти процессами.

В большинстве случаев отдельные вычислительные процессы не общаются друг с другом, выполняя самостоятельную и никак не связанную работу. Но существуют задачи, для решения которых создается не один процесс, а несколько, которые, работая совместно, выполняют общую работу. Такие процессы называются *кооперативными*.

Для взаимодействия процессов, в частности кооперативных, сложился ряд средств, часть которых – семафоры уже рассматривались, а некоторые будут изучаться в следующих главах. При всем их многообразии следует вспомнить, что основной информационный ресурс, используемый программами – оперативная память – это самый универсальный и быстрый ресурс компьютера. Поэтому не удивительно, что в состав всех современных ОС включены средства использования памяти для взаимодействия независимых процессов. Основой этих средств является *разделяемая память*, сами средства представляют набор системных функций использования этой памяти, а на более детальном уровне – и информационные средства описания этой памяти.

Изучение средств совместного использования памяти начнем с ОС Unix. Здесь разделяемая память находится под непосредственным управлением ядра, которое содержит таблицу описания областей разделяемой памяти. Каждая из областей обозначается в этой таблице целочисленным идентификатором (а не текстовым именем, как в других ОС). Кроме того, каждая такая область описывается в этой таблице атрибутами доступа и размером. Области разделяемой памяти относятся к адресному пространству ядра ОС.

Доступ к разделяемой памяти со стороны процесса осуществляется в два этапа. На первом из них получается хэндл области памяти, причем на этом этапе либо открывается доступ к уже имеющейся в ОС области памяти, либо такая область создается операционной системой. (Формально ситуация очень напоминает предварительные действия перед непосредственной работой с файлом.) На втором этапе процесс подключается к разделяемой области (to attach), используя ранее полученный хэндл. (Заметим, что сам термин *хэндл* в первоисточниках по Unix не используется, а применяется термин *идентификатор*, который в данном тексте действительно точнее. Мы же будем использовать термин *хэндл* для единообразного рассмотрения средств разделяемой памяти в различных ОС.)

На этапе подключения происходит подсоединение указанной области памяти, находящейся в ведении ОС, к виртуальному адресному пространству процесса, запросившего такое подключение. Результатом этой операции является базовый виртуальный адрес, начиная с которого в текущем процессе можно обращаться к разделяемой памяти.

36. Защита памяти. Средства преобразования. Распределение памяти между системными и прикладными задачами

Защита памяти — это способ управления правами доступа к отдельным регионам памяти. Используется большинством многозадачных операционных систем. Основной целью защиты памяти является запрет доступа процессу к той памяти, которая не выделена для этого процесса. Такие запреты повышают надежность работы как программ так и операционных систем, так как ошибка в одной программе не может повлиять непосредственно на память других приложений.

Методы защиты базируются на некоторых классических подходах, которые получили свое развитие в архитектуре современных ЭВМ. К таким методам можно отнести защиту отдельных ячеек, метод граничных регистров, метод ключей защиты.

Преобразование памяти в операционных системах обычно происходит с помощью специальных механизмов, таких как страницы и сегменты памяти.

Страницы памяти - это небольшие блоки памяти фиксированного размера, которые используются для управления виртуальной памятью. Система управления памятью разделяет физическую память на страницы определенного размера и создает соответствующие виртуальные адреса для каждой страницы. Это позволяет приложениям использовать большой объем памяти, намного превышающий фактическую физическую память компьютера.

Сегменты памяти - это логические блоки памяти, которые используются для управления кодом, данными и стеком приложения. Каждый сегмент имеет свой собственный начальный адрес и размер, который определяется во время компиляции или загрузки программы.

Операционные системы могут использовать различные стратегии для управления памятью, такие как пейджинг, виртуальная память и фрагментация памяти. Все эти механизмы позволяют эффективно использовать ресурсы компьютера и максимально оптимизировать работу приложений.

В операционных системах используется механизм виртуальной памяти для **распределения** физической памяти между системными и прикладными задачами. Каждой задаче выделяется свой адресное пространство, которое может превышать объем физической памяти в системе.

Операционная система использует страничное преобразование для трансляции виртуальных адресов, используемых задачами, в физические адреса в памяти. Каждая

страница памяти может иметь свой признак доступа и прав на чтение/запись/исполнение. Операционная система может также использовать различные алгоритмы для определения, какие страницы должны быть сохранены в физической памяти, а какие могут быть выгружены на жесткий диск.

В целом, операционная система старается как можно более эффективно использовать имеющуюся память в системе, давая приоритет системным задачам, которые обычно потребляют меньше памяти по сравнению с прикладными задачами.

37. Состав, функции, структура системы ввода/вывода. BIOS, EFI/UEFI.

базовая система ввода-вывода (Basic Input Output System – BIOS) является, с одной стороны, составной частью аппаратных средств, с другой – одним из программных модулей ОС. Возникновение данного названия связано с тем, что BIOS включает в себя набор программ ввода-вывода. С помощью этих программ ОС и прикладные программы могут взаимодействовать как с различными устройствами самого компьютера, так и с периферийными устройствами.

Как составная часть аппаратных средств система BIOS в ПК реализована в виде одной микросхемы, установленной на материнской плате компьютера. Большинство современных видеоадаптеров и контроллеров-накопителей имеют собственную систему BIOS, которая дополняет системную BIOS. Одним из разработчиков BIOS является фирма IBM, создавшая NetBIOS. Данный программный продукт не подлежит копированию, поэтому другие производители компьютеров были вынуждены использовать микросхемы BIOS независимых фирм. Конкретные версии BIOS связаны с набором микросхем (или чипсетом), находящихся на системной плате.

Как программный модуль ОС система BIOS содержит программу тестирования при включении питания компьютера POST (Power On Self Test – самотестирование при включении питания компьютера). При запуске этой программы тестируются основные компоненты компьютера (процессор, память и др.). Если при подаче питания компьютера возникают проблемы, т. е. BIOS не может выполнить начальный тест, то извещение об ошибке будет выглядеть как последовательность звуковых сигналов.

В «неизменяемой» памяти CMOS RAM хранится информация о конфигурации компьютера (количестве памяти, типах накопителей и др.). Именно в этой информации нуждаются программные модули системы BIOS. Данная память выполнена на основе определенного типа CMOS-структур (CMOS – Complementary Metal Oxide Semiconductor), которые характеризуются малым энергопотреблением. Память CMOS энергонезависима, так как питается от аккумулятора, расположенного на системной плате, или батареи гальванических элементов, смонтированной на корпусе системного блока.

Аббревиатура BIOS означает Basic Input/Output System - базовая система ввода-вывода. Системный BIOS является программой самого нижнего уровня - он действует как интерфейс между аппаратными средствами (особенно чипсетом и процессором) и операционной системой. BIOS обеспечивает доступ к аппаратным средствам PC и позволяет разработать операционные системы более высокого уровня (DOS, Windows 95 и др), с помощью которых пользователь запускает приложения. BIOS также отвечает за управление параметрами аппаратных средств, за загрузку PC при включении питания или нажатии кнопки сброса, а также за другие системные функции.

Новые компьютеры используют прошивку UEFI вместо традиционного BIOS. Обе эти программы – примеры ПО низкого уровня, запускающегося при старте

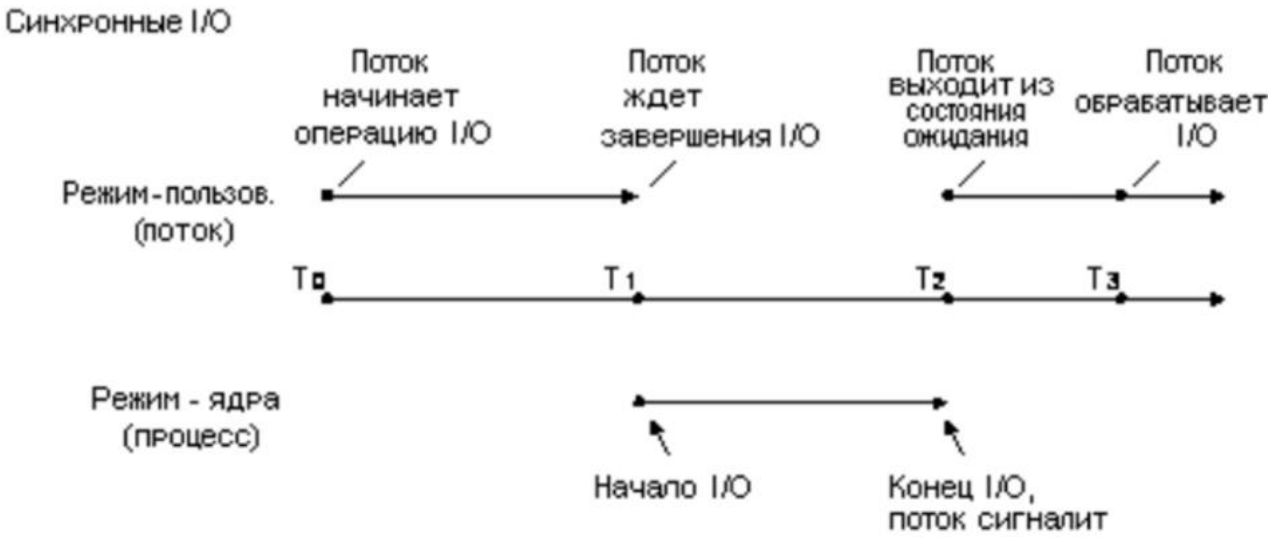
компьютера перед тем, как загрузится операционная система. UEFI – более новое решение, он поддерживает жёсткие диски большего объёма, быстрее грузится, более безопасен – и, что очень удобно, обладает графическим интерфейсом и поддерживает мышь.

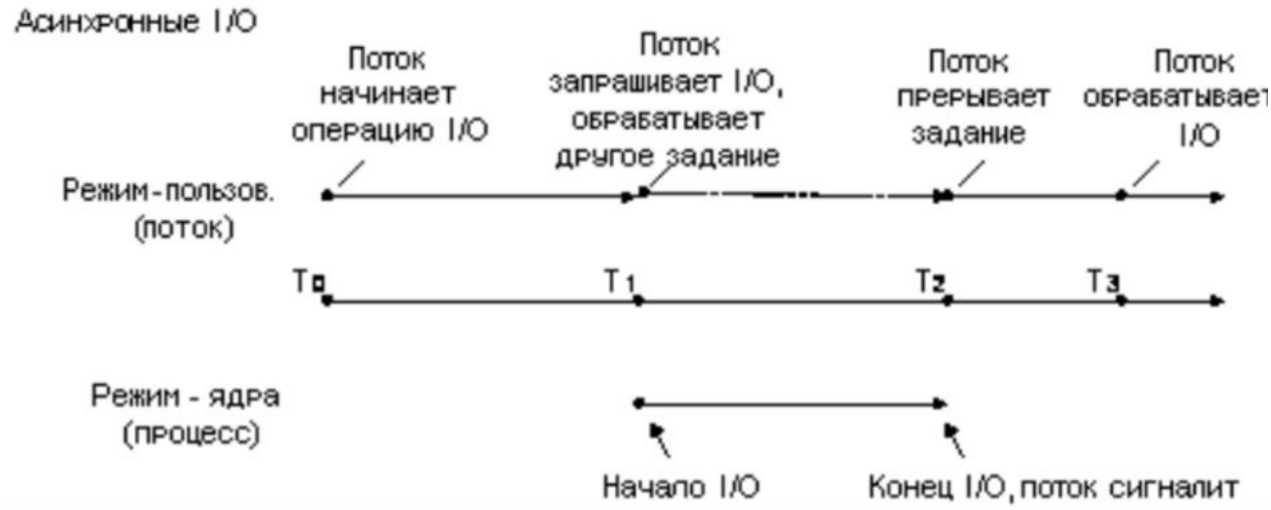
38. Синхронный и асинхронный ввод/вывод.

Существует два типа синхронизации ввода-вывода: синхронный ввод-вывод и асинхронный ввод-вывод. Асинхронный ввод-вывод также называется перекрывающимся вводом-выводом.

В *синхронном файловом вводе-вывод* поток запускает операцию ввода-вывода и сразу же переходит в состояние ожидания, пока запрос ввода-вывода не завершится. Поток, выполняющий *асинхронный файловый ввод-вывод*, отправляет запрос ввода-вывода в ядро путем вызова соответствующей функции. Если запрос принимается ядром, вызывающий поток продолжает обработку другого задания, пока ядро не сигнализирует потоку о завершении операции ввода-вывода. Затем он прерывает текущее задание и при необходимости обрабатывает данные операции ввода-вывода.

Процесс синхронизации двух типов:





В ситуациях, когда запрос ввода-вывода занимает много времени, например обновление или резервное копирование большой базы данных или медленное соединение связи, асинхронный ввод-вывод обычно является хорошим способом

оптимизации эффективности обработки. Однако для относительно быстрых операций ввода-вывода затраты на обработку запросов ввода-вывода ядра и сигналов ядра могут сделать асинхронные операции ввода вывода менее полезными, особенно если необходимо выполнить множество быстрых операций ввода-вывода. В этом случае лучше использовать синхронные операции ввода-вывода. Механизмы и сведения о реализации выполнения этих задач зависят от типа используемого дескриптора устройства и конкретных потребностей приложения. Другими словами, обычно существует несколько способов решения проблемы.

39. Способы обмена с внешними устройствами, их программная реализация. Драйверы.

Существует два основных способа обмена с внешними устройствами:

1) Прямой доступ к памяти.
2) Программно-управляемый ввод-вывод.

Прямой доступ к памяти:

Прямой доступ к памяти (DMA - Direct Memory Access) является способом обмена данными между внешним устройством и памятью без участия процессора и предназначен в основном для устройств, обменивающихся большими блоками данных с оперативной памятью. Инициатором обмена всегда выступает внешнее устройство. Процессор инициализирует контроллер DMA, и далее обмен выполняется под управлением контроллера. Если выбранный режим обмена не занимает всей пропускной способности шины, во время операций DMA процессор может продолжать работу.

Программно-управляемый ввод-вывод:

Программно-управляемый ввод-вывод означает обмен данными с внешними устройствами с использованием команд процессора. Передача данных происходит через регистры процессора и при этом в конечном счете может реализовываться обмен собственно с процессором, обмен внешнего устройства с памятью, обмен между внешними устройствами.

Процессоры x86 имеют раздельную адресацию памяти и портов ввода-вывода и соответственно ввод-вывод может быть отображен либо в пространство ввода-вывода, либо в пространство оперативной памяти (memory-mapped I/O). В последнем случае адрес памяти декодируется во внешнем устройстве и для выполнения ввода-вывода могут быть использованы все команды обращения к памяти.

Каждый адресуемый элемент адресного пространства ввода-вывода именуется портом ввода, портом вывода или портом ввода-вывода. Для обращения к портам предназначены четыре основные команды процессора: **In** (ввод в порт), **Out** (вывод из порта), **Ins** (ввод из порта в элемент строки памяти) и **Outs** (вывод элемента из строки памяти). Последние две строковые команды ввода-вывода используются для быстрой пересылки блоков данных между портом и памятью в случае последовательно расположенных адресов портов в внешнем устройстве. Обмен данными с портами, при котором используются строковые команды ввода-вывода, получил название PIO (Programmed Input/Output) - программированный ввод-вывод.

Драйвер - компьютерное программное обеспечение, с помощью которого другое программное обеспечение (операционная система) получает доступ к аппаратному обеспечению некоторого устройства. Обычно с операционными системами поставляются драйверы для ключевых компонентов аппаратного обеспечения, без которых система не сможет работать. Однако для некоторых устройств (таких, как

видеокарта или принтер) могут потребоваться специальные драйверы, обычно предоставляемые производителем устройства.

40. Иерархия драйверов. Многослойная модель драйвера.

Драйверы устройств в компьютерных системах обычно организованы в иерархическую структуру, которая также может быть описана как многослойная модель. Это означает, что драйверы устройств работают на разных уровнях системы, в зависимости от их функций и взаимодействия с аппаратным обеспечением.

Многослойная модель драйвера представляет собой подход, в котором драйверы разбиты на отдельные слои, каждый из которых выполняет определенные функции. Например, верхние слои модели могут предоставлять абстракцию устройства и управлять его конфигурацией, а нижние слои могут обрабатывать непосредственное взаимодействие с аппаратурой. Многослойная модель позволяет разделять ответственность между различными частями драйвера, облегчает его разработку, тестирование и поддержку.

В общем виде иерархия драйверов может быть представлена следующим образом:

1) **Уровень ядра (Kernel Level):** Это самый низкий уровень, где драйверы взаимодействуют непосредственно с аппаратным обеспечением. Драйверы на этом уровне обычно управляют такими вещами, как прерывания процессора, доступ к памяти и другие низкоуровневые функции.
2) **Уровень аппаратного обеспечения (Hardware Level):** На этом уровне драйверы управляют конкретными устройствами, такими как диски, сетевые карты и т.д. Они обеспечивают интерфейс между аппаратным обеспечением и операционной системой.
3) **Уровень операционной системы (OS Level):** На этом уровне драйверы обеспечивают функциональность, которую операционная система может использовать для взаимодействия с аппаратным обеспечением. Это включает в себя такие вещи, как управление файловой системой, сетевые операции и т.д.
4) **Уровень приложений (Application Level):** На этом уровне драйверы обеспечивают интерфейс, который приложения могут использовать для взаимодействия с аппаратным обеспечением через операционную систему. Это включает в себя такие вещи, как API для графических операций, звуковые драйверы и т.д.

Такая многоуровневая организация позволяет маскировать аппаратные детали, упрощает разработку и делает драйвер более переносимым. Использование стандартных протоколов и API уровней также повышает совместимость с различными ОС.

41. Унифицированная модель драйвера

Унифицированная модель драйвера (Unified Driver Model, UDM) - это концепция и подход, направленные на создание единой модели программного драйвера, которая

может быть использована на различных платформах и с разными устройствами. Цель UDM состоит в том, чтобы упростить разработку драйверов, повысить их совместимость и обеспечить более эффективное использование ресурсов.

В унифицированной модели драйвера ключевыми элементами являются:

1. **Единый интерфейс:** UDM определяет единый интерфейс или набор API, который драйвер должен использовать для взаимодействия с устройством и операционной системой. Этот интерфейс предоставляет абстракцию от конкретных деталей устройства и позволяет разработчикам создавать драйверы, которые могут работать на разных платформах без изменений.
2. **Переносимость:** Унифицированная модель драйвера стремится к созданию драйверов, которые могут быть перенесены на различные операционные системы. Это позволяет разработчикам создавать единую версию драйвера, которая может быть использована на Windows, Linux, macOS и других платформах.
3. **Абстракция устройства:** UDM предоставляет абстракцию устройства, которая скрывает конкретные детали и особенности каждого устройства. Драйвер взаимодействует с абстрактным представлением устройства через единый интерфейс, что упрощает разработку и обслуживание драйвера.
4. **Общая функциональность:** Унифицированная модель драйвера обычно предоставляет общую функциональность, необходимую для работы с устройством, такую как управление памятью, вводом-выводом, прерываниями и другими аппаратными возможностями. Это позволяет разработчикам использовать общий код для реализации базовых функций, что повышает эффективность разработки и снижает возможность ошибок.
5. **Обратная совместимость:** Унифицированная модель драйвера также обычно поддерживает обратную совместимость с существующими драйверами. Это означает, что новые версии операционной системы будут поддерживать старые драйверы, разработанные с использованием предыдущих версий UDM, что обеспечивает плавный переход и минимизирует проблемы совместимости.

В итоге, унифицированная модель драйвера представляет собой концепцию и инфраструктуру, которая способствует разработке драйверов, обеспечивает их совместимость и повышает эффективность использования ресурсов. Это имеет важное значение для создания стабильных и совместимых систем, где драйверы играют важную роль в обеспечении работы устройств и операционной системы.

42. Обработка прерываний, исключений, ловушек. Типы прерываний.

Обработчик прерываний — специальная процедура, вызываемая по прерыванию для выполнения его обработки. Обработчики прерываний могут выполнять множество функций, которые зависят от причины, которая вызвала прерывание.

Обработчики вызываются либо по аппаратному прерыванию (внешние устройства, по типу мышь, клавиатура), либо по программному (инструкция в самой программе), и соответственно обычно предназначены для взаимодействия с устройствами или для осуществления вызова функций операционной системы.

На современных ПК обработчики основных аппаратных и программных прерываний находятся в памяти BIOS. Современная операционная система, во время своей загрузки, заменяет эти обработчики своими. При загрузке драйверов устройств операционная система распределяет управление обработкой прерывания между ними. В операционных системах семейства Windows программные прерывания используются для вызовов многих API функций. В ассемблере x86 прерывание вызывается командой INT (Например, INT 16h – прерывание для взаимодействия с клавиатурой).

Главные механизмы прерывания:

1) **Распознавание прерывания**

На данном этапе устанавливается факт прерывания, запоминается состояние прерванного процесса.

2) **Передача управления обработчику прерывания**

На данном этапе происходит передача управления обработчику и сама обработка прерывания.

3) **Корректное возвращение к прерванной программе**

На данном этапе происходит восстановление информации, относящейся к прерванной программе, а также возврат в прерванную программу.

Обработчики делятся следующие типы:

1) **Высокоприоритетные обработчики прерываний (ВОП)**

К ним предъявляются жёсткие требования: малое время на выполнение, малое количество операций, разрешенных к выполнению, особая надежность, так как ошибки, допущенные во время выполнения, могут обрушить операционную систему, которая не может корректно их обработать. Поэтому ВОП обычно выполняют минимально необходимую работу.

2) **Низкоприоритетные обработчики прерываний (ВОП)**

НОП завершает обработку прерывания. НОП либо имеет собственный поток для обработки, либо заимствует на время обработки поток из системного пула. Эти потоки планируются наравне с другими, что позволяет добиться более гладкого выполнения процессов. НОП выполняется с гораздо менее жесткими ограничениями по времени и ресурсам, что облегчает программирование и использование драйверов.

Типы прерываний:

1) **Асинхронные (или внешние)** – события, которые исходят от внешних аппаратных устройств (например, периферийных устройств) и могут произойти в любой произвольный момент: сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши.

2) **Синхронные (или внутренние)** – события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода: деление на ноль или переполнение стека, обращение к недопустимым адресам памяти.

3) **Программные (частный случай внутреннего прерывания)** – инициируются исполнением специальной инструкции в коде программы. Программные прерывания, как правило, используются для обращения к функциям встроенного программного обеспечения.

Термин ловушка применяется для внутренних прерываний.

Обработка исключений – механизм в языках программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (*исключения*), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.

Примеры исключительных ситуаций: деление на ноль, ошибка считывания данных с устройства, отказ в выделении памяти, аварийное отключение питания.

Выделяют 2 вида исключительных ситуаций:

1) **Синхронные**

Могут возникнуть только в заранее известных точках программы. Например, известно где потребуется выделить динамическую память, и система может её не дать, такая ситуация подлежит обработке.

2) **Асинхронные**

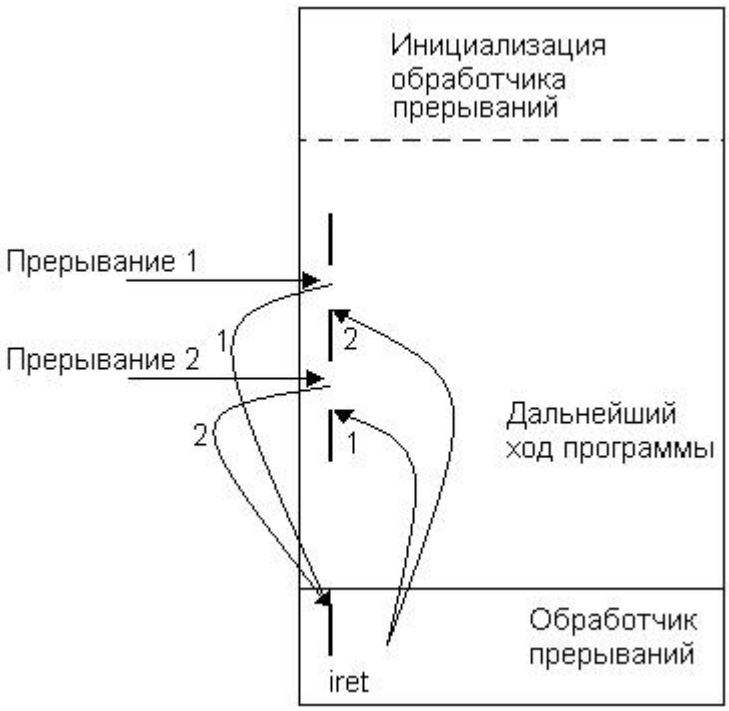
Могут возникать в любой момент времени и не зависят от того, какую конкретно инструкцию программы выполняет система. Например, аварийный отказ питания, но при наличии резервного питания данную ситуацию можно обработать.

43. Обработчики аппаратных прерываний, правила написания программного кода обработчиков

Прерывания разделяются на внутренние, возникающие в самом микропроцессоре в случае определенных сбоев (попытка деления на 0, несуществующая команда), внешние, приходящие из периферийного оборудования (клавиатура, мышь, диски, нестандартные устройства, подключенные к компьютеру) и программные, являющиеся реакцией процессора на команду **int** с тем или иным номером. В прикладных программах приходится обрабатывать, главным образом, внешние и программные прерывания.

Общие принципы обслуживания тех и других прерываний одинаковы, однако условия функционирования обработчиков аппаратных прерываний имеют значительную специфику, связанную, главным образом, с тем, что прерывания от аппаратуры приходят в произвольные моменты времени и могут прервать текущую программу в любой ее точке. Обработчик прерывания должен быть написан таким образом, чтобы его выполнение ни в какой степени не отразилось на правильном функционировании текущей (прерываемой) программы.

Рассмотрим схематически структуру и функционирование программного комплекса, включающего собственный обработчик какого-либо аппаратного прерывания.



Обработчик прерываний может входить в состав программы в виде процедуры, или просто являться частью программы, начинающейся с некоторой метки (входной точки обработчика) и завершающейся командой выхода из прерывания **iret**. Обработчик может представляет собой самостоятельную резидентную программу.

Программа, начиная свою работу, прежде всего должна выполнить инициализирующие действия по установке обработчика прерываний. В простейшем случае эти действия заключаются в занесении в соответствующий вектор полного адреса (сегмента и смещения) обработчика. Поскольку обработчик входит в состав программы, его относительный адрес известен; это имя его процедуры или метка входной точки. Что же касается сегментного адреса, то обработчик может входить в сегмент основной части программы, если она невелика по объему и занимает один сегмент, но может образовывать и самостоятельный сегмент. В любом случае в качестве сегментного адреса можно использовать имя соответствующего сегмента.

Часто инициализация обработчика, помимо установки вектора, предполагает и другие действия: сохранение исходного содержимого вектора прерывания, размаскирование соответствующего уровня прерываний в контроллере прерываний, посылка в устройство команды разрешения прерываний и проч.

Установив обработчик, программа может приступить к дальнейшей работе. В случае прихода прерывания, процессор сохраняет в стеке флаги и текущий адрес программы, извлекает из вектора адрес обработчика и передает управление на его входную точку. Все время, пока выполняется программа обработчика, основная программа, естественно, стоит. Завершающая обработчик команда irct извлекает из стека сохраненные там данные и возвращает управление в прерванную программу, которая может продолжить свою работу. Последующие прерывания обслуживаются точно так же.

Функции обработчика прерываний зависят от решаемой задачи и назначения того устройства, от которого поступают сигналы прерываний. В случае прерываний от клавиатуры задача обработчика прерываний – принять и сохранить код нажатой клавиши. Прерывания от мыши свидетельствуют о ее перемещении, что требует обновления положения курсора на экране. Если обслуживаемым устройством является физическая установка, то сигнал прерывания может говорить о том, что в установке накоплен определенный объем данных, которые надо перенести из памяти установки в память компьютера. В любом случае обработчик прерываний должен быть программой несложной, для выполнения которой не требуется много машинного времени.

44. API- поддержка обработки прерываний и событий для проектирования приложений

Программное обеспечение может столкнуться с различными событиями, такими как нажатия клавиш, сетевые запросы, изменения состояния устройств и другие. Вместо активного ожидания наступления этих событий, приложения могут зарегистрировать обработчики событий и прерываний, чтобы быть проактивными в их обработке. Это позволяет эффективно использовать ресурсы системы и реагировать на события в режиме реального времени.

API поддержки обработки прерываний и событий предоставляет набор функций и интерфейсов, которые позволяют разработчикам регистрировать обработчики событий, настраивать их параметры и обрабатывать полученные данные или действия.

Преимущества использования API поддержки обработки прерываний и событий заключаются в следующем:

- **Асинхронная обработка:** Приложения могут эффективно обрабатывать события, не блокируя основной поток выполнения. Это позволяет предотвратить задержки и повысить отзывчивость приложения.
- **Гибкость:** Разработчики имеют возможность выбирать, какие события они хотят обрабатывать, и настраивать обработчики событий согласно своим потребностям.
- **Расширяемость:** Поддержка обработки прерываний и событий позволяет легко добавлять новые функции и интегрировать внешние системы или устройства, которые могут генерировать события.
- **Масштабируемость**: Архитектура, основанная на обработке событий, облегчает создание масштабируемых систем, так как приложения могут обрабатывать события асинхронно и распределять нагрузку между различными обработчиками.

Одним из примеров таких API является Windows API: Операционная система Windows предоставляет API для обработки прерываний и событий. Например, функции RegisterDeviceNotification и WndProc позволяют регистрировать обработчики событий устройств, таких как вставка или удаление USB-устройств. Это позволяет приложению реагировать на изменения в подключенных устройствах.

Различные операционные системы предоставляют свои собственные API для обработки событий и прерываний. Например, в Windows это может быть API, связанное с обработкой сообщений и оконных событий, а в Unix-подобных системах это может быть API для работы с файловыми дескрипторами и системными вызовами, такими как select() или epoll().

45. Современные средства разработки ПО (SDK, IDE, Toolkits). Инструментальные средства ОС для трассировки, анализа и отладки кода.

Современные средства разработки программного обеспечения (ПО) включают в себя широкий спектр инструментов и технологий, которые значительно упрощают и ускоряют процесс разработки. Среди них можно выделить следующие:

- **Интегрированные среды разработки (IDE)** - это программные пакеты, которые объединяют в себе несколько инструментов, необходимых для создания и отладки приложений. Как правило, IDE содержат текстовый редактор, компилятор, отладчик, а также другие инструменты, такие как графический дизайнер интерфейсов, базы данных и т.д. Примеры популярных IDE: Visual Studio, Eclipse, IntelliJ IDEA, NetBeans.

- **Комплекты разработчика ПО (SDK)** - это наборы ПО и инструментов, необходимых для разработки приложений на определенной платформе или для определенного языка программирования. SDK может включать в себя компиляторы, библиотеки, инструменты отладки, документацию и т.д. Например, Android SDK, iOS SDK, Java SDK.

- **Инструментарий разработки пользовательского интерфейса (UI Toolkits)** - это библиотеки и инструменты, используемые для создания графических интерфейсов пользователей. Они снижают сложность и время разработки интерфейса, позволяют создавать качественные визуальные элементы и обеспечивают совместимость с различными платформами. Примеры UI Toolkits: Qt, wxWidgets, JavaFX.

В ОС также есть встроенные инструментальные средства для трассировки, анализа и отладки кода. Некоторые из них:

- **Профилировщик (Profiler)** - это инструмент для измерения производительности и нахождения узких мест в программе. Позволяет определить точки, где программа затрачивает наибольшее количество времени, и улучшить ее производительность. Примеры: Visual Studio Profiler, Xcode Instruments.

- **Отладчик (Debugger)** - это инструмент для устранения ошибок в коде, позволяет запускать программу в режиме отладки шаг за шагом, задавать точки останова и искать ошибки. Примеры: gdb (Linux), WinDbg (Windows), lldb (macOS/iOS).

- **Монитор памяти (Memory Profiler) -** это инструмент для обнаружения утечек памяти, позволяет отслеживать использование оперативной памяти в программе и находить утечки. Примеры: Visual Studio Memory Profiler, Valgrind.

- **Трассировщик (Tracer)** - это инструмент для изучения работы программы и ее взаимодействия с внешней средой. Позволяет записывать и анализировать последовательность вызовов функций в программе. Примеры: strace (Linux), Process Monitor (Windows), DTrace (macOS)