Оглавление

1. Операционная система. Назначение, основные функции, классификации операционных систем

ОС – комплекс системных управляющих и обрабатывающих программ, которые:

1. выступают как интерфейс между аппаратурой компьютера и пользователем с его задачами;
2. предназначены для наиболее эффективного расходования ресурсов вычислительной системы и организации надежных вычислений.

Назначение ОС – динамическое распределение ресурсов и управление ими в соответствии с требованиями вычислительных процессов (задач).

Функции:

1. управление процессами;
2. управление памятью;
3. управление файлами и внешними устройствами;
4. защита данных;
5. администрирование;
6. интерфейс прикладного программирования;
7. пользовательский интерфейс.

Операционные системы классифицируются по:

1. количеству одновременно работающих пользователей:
    a. однопользовательские (MS-DOS);
    b. многопользовательские (Unix, Windows).
2. числу процессов, одновременно выполняемых под управлением системы:
    a. однозадачные (MS-DOS);
    b. многозадачные (Unix, Windows).
3. количеству поддерживаемых процессоров:
    a. однопроцессорные (мобильные ОС, например, все версии ОС Android до версии 3);
    b. многопроцессорные (Linux, Windows, Mac OS X).
4. типу доступа пользователя к ЭВМ:
    a. с пакетной обработкой (ОС ЕС) – не требуют быстрого получения рез-ов, высокая производительность при обработке больших объемов информации.
    b. с разделением времени (Unix, Window) – для выполнения каждой задачи выделяется небольшой промежуток времени.

c. реального времени (LynxOS, QNX) – для управления тех-им процессом или техническим объектом, например, летательным объектом, станком.

5. типу использования ресурсов:
   a. сетевые – предназначены для управления рес-ми компьютеров, объединенных в сеть с целью совместного использования ресурсов. Они представляют мощные средства разграничения доступа к информации, её целостности и другие возможности исп-ия сетевых рес-ов.
   b. локальные – в их состав входит клиентская часть ПО для доступа к удаленным ресурсам и услугам.

2. Архитектуры операционных систем (ядер операционных систем)

Монолитное ядро.

Представляет богатый набор оборудования. Все компоненты монолитного ядра находятся в одном адресном пространстве. Эта схема ОС, когда все части ее ядра – это составные части одной программы. Монолитное ядро – самый старый способ организации ОС.

Достоинства: высокая скорость работы, простая разработка модулей.
Недостатки: ошибка работы одного из компонентов ядра нарушает работу всей системы.

Модульное ядро.

Это современная модификация монолитных ядер ОС, но в отличие от них модульное ядро не требует полной перекомпиляции ядра при изменения аппаратного обеспечения компьютера. Более того модульные ядра имеют механизм погрузки модулей ядра. Погрузка бывает статической – с перезагрузкой ОС, и динамической – без перезагрузки ОС.

Микроядро.

Представляет только основные функции управления процессами и минимальный набор для работы с оборудованием. Классические микроядра дают очень небольшой набор системных вызовов.

Достоинства: устойчивость к сбоям и ошибкам оборудования и компонентов системы, высокая степень ядерной модульности, что упрощает добавление в ядро новых компонентов и процесс отладки ядра.

Для отладки такого ядра можно использовать обычные средства. Архитектура микроядра увеличивает надежность системы.

Недостатки: Передача информации требует больших расходов и большого количества времени.

Экзоядро.

Такое ядро ОС, которое предоставляет лишь функции взаимодействия процессов, безопасное выделение и распределение ресурсов. Доступ к устройствам на уровне контроллеров позволяет решать задачи, которые нехарактерны для универсальной ОС.

Наноядро.

Такое ядро выполняет только единственную задачу- обработку аппаратных прерываний, образуемых устройствами ПК. После обработки наноядро посылает данные о результатах обработки далее идущему в цепи программному обеспечения при помощи той же системы прерываний.

Гибридное ядро.

Модификация микроядер, позволяющая для ускорения работы впускать несущественные части в пространство ядра. На архитектуре гибкого ядра построены последние операционные системы от Windows, в том числе и Windows 7.

Смешанная архитектура

Попытка совместить достоинства различных архитектур и/или компенсировать их недостатки. Встречается часто, т.к. чистые архитектуры на практике адаптированы к конкретным реальным условиям. Все перечисленные подходы к построению ОС безусловно имеют как преимущества, так и недостатки. Поэтому в большинстве современных операционных системах используют различные комбинации подходов к построению. Обычно за основу берут один из подходов и дополняют в него элементы других подходов, стараясь свести к минимумы недостатки.

3. Структура операционной системы, основные компоненты

В состав операционной системы входят следующие подсистемы:

1. Управление процессами.
2. Управление основной памятью.

3. Управление внешней памятью.
4. Управление устройствами ввода/вывода.
5. Управление файлами.
6. Защита системы.
7. Сетевая поддержка.
8. Командный интерфейс системы.
9. Графическая оболочка.

Управление процессами

Процесс - это программа в стадии выполнения.

ОС управляет:

- работой процессов;
- их распределением по процессорам и ядрам;
- порядком их выполнения и размещения в памяти;
- их синхронизацией при параллельном решении частей одной и той же задачи разными процессами.

Управление основной памятью

Основная (оперативная) память может рассматриваться как большой массив.

ОС:

- распределяет ресурсы памяти между процессами;
- выделяет память по запросу;
- освобождает её при явном запросе или по окончании процесса;
- хранит списки занятой и свободной памяти в системе.

Управление внешней памятью

Внешняя память – это расширение оперативной памяти процессора более медленными, но более ёмкими и постоянно хранящими информацию видами памяти (диски).

ОС решает задачи:

- выделения памяти по запросу;
- освобождения памяти;
- хранение списков свободной и занятой памяти;
- поддерживает использование ассоциативной памяти (кэш-памяти) для оптимизации обращения к внешней памяти.

Подсистема управления устройствами ввода/вывода

Подсистема ввода/вывода состоит из:

- системы кэширования - буферирования;
- общего интерфейса драйверов устройств;
- драйверов специализированных устройств.

Подсистема управления файлами

Файл – это именованный набор данных на внешнем носителе памяти, например, на диске.

ОС:

- организует работу пользовательских программ с файлами;
- выполняет их открытие и закрытие и операции над ними (чтение и запись);
- хранит ссылки на файлы в директориях (папках);
- обеспечивает их поиск по символьным именам.

Защита системы

При работе ОС должны быть обеспечены надежность и безопасность, т.е. защита от внешних атак, конфиденциальность личной и корпоративной информации, диагностика и исправления ошибок и неисправностей и др.

ОС:

- обеспечивает защиту компонент компьютерной системы, данных и программ;
- поддерживает фильтрацию сетевых пакетов, обнаружение и предотвращение внешних атак;
- хранит информацию обо всех действиях над системными структурами, полезную для анализа атак и борьбы с ними.

Сетевое обеспечение

Любая современная компьютерная система постоянно или временно находится в различных локальных и глобальных сетях.

ОС:

- обеспечивает использование сетевого оборудования (сетевых карт, или адаптеров), вызов соответствующих драйверов;

- поддержку удаленного взаимодействия с файловыми системами, находящимися на компьютерах сети;
- удаленный вход на другие компьютеры сети;
- использование их вычислительных ресурсов;
- отправку и получение сообщений по сети;
- защиту от сетевых атак.

Командный интерфейс системы

Любая ОС поддерживает командный язык (или набор командных языков), состоящих из пользовательских команд, выполняемых с пользовательского терминала (из пользовательской консоли).

- В системе Windows для выполнения команд используется окно пользовательской консоли MS-DOS (MS-DOS Prompt).
- В системе Linux – специальное окно "Терминал".

Графическая оболочка

Графическая оболочка – подсистема ОС, реализующая графический пользовательский интерфейс пользователей и системных администраторов с операционной системой. Использование одного лишь командного языка и системных вызовов неудобно, поэтому простой и наглядный графический пользовательский интерфейс с ОС необходим.

- Среди графических оболочек, используемых в системах типа UNIX, можно назвать CDE, KDE, GNOME.
- ОС Windows и MacOS имеют собственные, весьма удобные графические оболочки.

4. Многозадачность. Многозадачные операционные системы. Виды многозадачности и параллелизма

Многозадачность – свойство операционной системы или среды выполнения обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких процессов. Истинная многозадачность операционной системы возможна только в распределённых вычислительных системах.

Существует 2 типа многозадачности:

- **Процессная многозадачность** (основанная на процессах - одновременно выполняющихся программах). Здесь программа - наименьший элемент кода, которым может управлять

планировщик операционной системы. Более известна большинству пользователей (работа в текстовом редакторе и прослушивание музыки).

- **Поточная многозадачность** (основанная на потоках). Наименьший элемент управляемого кода - поток (одна программа может выполнять 2 и более задачи одновременно).):

Типы псевдопараллельной многозадачности

Невытесняющая многозадачность

Тип многозадачности, при котором операционная система одновременно загружает в память два или более приложений, но процессорное время предоставляется только основному приложению.

Совместная или кооперативная многозадачность

Тип многозадачности, при котором следующая задача выполняется только после того, как текущая задача явно объявит себя готовой отдать процессорное время другим задачам. Как частный случай, такое объявление подразумевается при попытке захвата уже занятого объекта mutex (ядро Linux), а также при ожидании поступления следующего сообщения от подсистемы пользовательского интерфейса. Все приложения делят процессорное время, периодически передавая управление следующей задаче.

Преимущества: отсутствие необходимости защищать все разделяемые структуры данных объектами типа критических секций и mutex'ов, что упрощает программирование, особенно перенос кода из однозадачных сред в многозадачные.

Недостатки: неспособность всех приложений работать в случае ошибки в одном из них, приводящей к отсутствию вызова операции «отдать процессорное время». Крайне затрудненная возможность реализации многозадачной архитектуры ввода-вывода в ядре ОС, позволяющей процессору исполнять одну задачу в то время, как другая задача инициировала операцию ввода-вывода и ждет её завершения.

Реализована в пользовательском режиме ОС Windows версий до 3.x включительно, Mac OS версий до Mac OS X, а также внутри ядер многих UNIX-подобных ОС, таких, как FreeBSD, а в течение долгого времени — и Linux.

Вытесняющая или приоритетная многозадачность (режим реального времени)

Вид многозадачности, в котором операционная система сама передает управление от одной выполняемой программы другой в случае завершения операций ввода-вывода, возникновения событий в аппаратуре компьютера, истечения таймеров и квантов времени, или же поступлений тех или иных сигналов от одной программы к другой. В этом виде многозадачности процессор может быть переключен с исполнения одной программы на исполнение другой без всякого пожелания первой программы и буквально между любыми двумя инструкциями в её коде. Распределение процессорного времени осуществляется планировщиком процессов. К тому же каждой задаче может быть назначен пользователем или самой операционной системой определенный приоритет, что обеспечивает гибкое управление распределением процессорного времени между задачами (например, можно снизить приоритет ресурсоёмкой программе, снизив тем самым скорость её работы, но повысив производительность фоновых процессов). Этот вид многозадачности обеспечивает более быстрый отклик на действия пользователя.

Преимущества: возможность полной реализации многозадачного ввода-вывода в ядре ОС, когда ожидание завершения ввода-вывода одной программой позволяет процессору тем временем исполнять другую программу. Сильное повышение надежности системы в целом, в сочетании с использованием защиты памяти — идеал в виде «ни одна программа пользовательского режима не может нарушить работу ОС в целом» становится достижимым хотя бы теоретически, вне вытесняющей многозадачности он не достижим даже в теории. Возможность полного использования многопроцессорных и многоядерных систем.

Недостатки: необходимость особой дисциплины при написании кода, особые требования к его реентрантности, к защите всех разделяемых и глобальных данных объектами типа критических секций и mutex’ов.

Реализована в таких ОС, как:VMS; Linux; в пользовательском режиме (а часто и в режиме ядра) всех UNIX-подобных ОС, включая версии Mac OS X, iPod OS и iPhone OS; Windows NT/2000/XP/Vista/7 и в режиме ядра, и в пользовательском режиме.;

5. Вычислительный процесс. Жизненный цикл и состояния процессов. Порождение и завершение процессов

Процесс (или по-другому, задача) – абстракция, описывающая выполняющуюся программу. Для ОС процесс представляет собой единицу работы, заявку на потребление системных ресурсов.

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

Состояние процессов

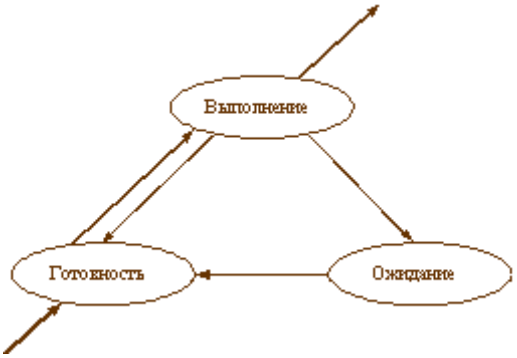
В многозадачной (многопроцессной) системе процесс может находиться в одном из трех основных состояний:

ВЫПОЛНЕНИЕ – активное состояние процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;

ОЖИДАНИЕ – пассивное состояние процесса, процесс заблокирован, он не может выполняться по своим внутренним причинам, он ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого процесса, освобождения какого-либо необходимого ему ресурса;

ГОТОВНОСТЬ – также пассивное состояние процесса, но в этом случае процесс заблокирован в связи с внешними по отношению к нему обстоятельствами: процесс имеет все требуемые для него ресурсы, он готов выполняться, однако процессор занят выполнением другого процесса.

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе. Типичный граф состояний процесса показан на рисунке.



В состоянии ВЫПОЛНЕНИЕ в однопроцессорной системе может находиться только один процесс, а в каждом из состояний ОЖИДАНИЕ и ГОТОВНОСТЬ – несколько процессов, эти процессы образуют очереди

соответственно ожидающих и готовых процессов. Жизненный цикл процесса начинается с состояния ГОТОВНОСТЬ, когда процесс готов к выполнению и ждет своей очереди. При активизации процесс переходит в состояние ВЫПОЛНЕНИЕ и находится в нем до тех пор, пока либо он сам освободит процессор, перейдя в состояние ОЖИДАНИЯ какого-нибудь события, либо будет насильно "вытеснен" из процессора, например, вследствие исчерпания отведенного данному процессу кванта процессорного времени. В последнем случае процесс возвращается в состояние ГОТОВНОСТЬ. В это же состояние процесс переходит из состояния ОЖИДАНИЕ, после того, как ожидаемое событие произойдет.

Порождение и завершение процессов

Управление процессами, прежде всего, - создание новых процессов и контроль их выполнения. Unix: новые экземпляры программы обычно создаются с помощью вызова функции fork – возвращает PID дочернего процесса родительскому (0 – дочернему), а также функцию clone (позволяет настраивать разделение ресурсов между процессами, но не умеет раздваивать процесс в точке вызова). Windows: CreateProcess

ОС необходим какой-нибудь способ для создания процессов. В самых простых системах, или в системах, сконструированных для запуска только одного приложения (например, в контроллере микроволновой печи), появляется возможность присутствия абсолютно всех необходимых процессов при вводе системы в действие. Но в универсальных системах нужны определенные способы создания и прекращения процессов по мере необходимости.

События, приводящие к **созданию** процессов:

1. инициализация системы;
2. выполнение работающим процессом системного вызова, предназначенного для создания процесса;
3. запрос пользователя на создание нового процесса;
4. инициация пакетного задания.

Причины **завершения** процессов:

1. обычный выход (добровольно);
2. выход при возникновении ошибки (добровольно);
3. возникновение фатальной ошибки (принудительно);
4. уничтожение другим процессом (принудительно).

6. Потоки и многопоточность. Состояния вычислительных потоков, их создание и завершение

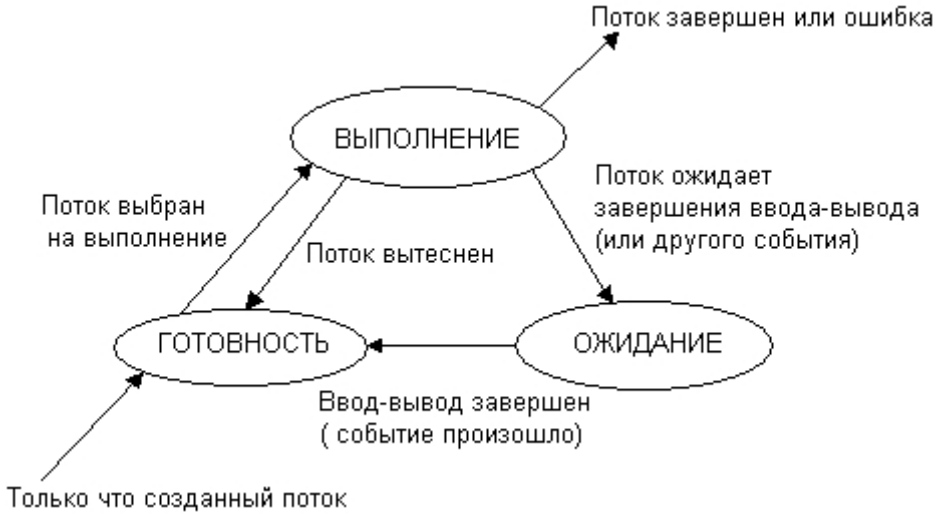
Поток – некая сущность внутри процесса, получающая процессорное время для выполнения.

Многопоточность – свойство платформы (например, ОС) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких *потоков*, выполняющихся «параллельно», то есть без предписанного порядка во времени.

При создании, каждый поток имеет:

- Уникальный идентификатор потока;
- Содержимое набора регистров процессора, отражающих состояние процессора;
- Два стека, один из которых используется потоком при выполнении в режиме ядра, а другой – в пользовательском режиме;
- Закрытую область памяти, называемую локальной памятью потока (thread local storage, TLS) и используемую подсистемами, run-time библиотеками и DLL.

Состояние потоков:



Есть несколько **причин**, почему приложения **создают потоки** в дополнение к их исходному начальному потоку:

1. процессы, обладающие пользовательским интерфейсом, обычно создают потоки для того, чтобы выполнять свою работу и при

этом сохранять отзывчивость основного потока к командам пользователя, связанными с вводом данных и управлением окнами;
2. приложения, которые хотят использовать несколько процессоров для масштабирования производительности или же которые хотят продолжать работать, в то время как потоки останавливают свою работу, ожидая синхронизации операций ввода/вывода, создают
3. потоки, чтобы получить дополнительную выгоду от многопоточной работы.

7. Подсистема управления процессами. Дисциплины планирования, приоритеты

Процесс (задача) - абстракция, описывающая выполняющуюся программу. Для ОС процесс представляет собой единицу работы, заявку на потребление системных ресурсов.

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

Алгоритмы планирования процессов

Планирование процессов включает в себя решение следующих задач:

1. определение момента времени для смены выполняемого процесса;
2. выбор процесса на выполнение из очереди готовых процессов;
3. переключение контекстов "старого" и "нового" процессов.

1, 2 - решаются программными средствами, 3 - аппаратно

В соответствии с алгоритмами, основанными на **квантовании**, смена активного процесса происходит, если:

1. процесс завершился и покинул систему,
2. произошла ошибка,
3. процесс перешел в состояние ОЖИДАНИЕ,
4. исчерпан квант процессорного времени, отведенный данному процессу.

Вытесняющие и невытесняющие алгоритмы планирования

Существует два основных типа процедур планирования процессов: **вытесняющие** и **невытесняющие**.

Невытесняющая многозадачность – активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику ОС для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

Вытесняющая многозадачность – решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком ОС, а не самой активной задачей.

Другая группа алгоритмов использует понятие **приоритет процесса.**

Существует две разновидности приоритетных алгоритмов: алгоритмы, использующие **относительные приоритеты**, и алгоритмы, использующие **абсолютные приоритеты**.

Ядро Windows всегда запускает тот из потоков, готовых к выполнению, который обладает наивысшим приоритетом. Поток не является готовым к выполнению, если он находится в состоянии ожидания, приостановлен или блокирован по той или иной причине.

Потоки получают приоритеты на базе классов приоритета своих процессов. Первоначально функцией CreateProcess устанавливаются четыре класса приоритета, каждый из которых имеет *базовый приоритет* (base priority):

1. IDLE_PRIORITY_CLASS, базовый приоритет 4.
2. NORMAL_PRIORITY_CLASS, базовый приоритет 9 или 7.
3. HIGH_PRIORITY_CLASS, базовый приоритет 13.
4. REALTIME_PRIORITY_CLASS, базовый приоритет 24.

Оба предельных класса используются редко, и в большинстве случаев можно обойтись нормальным (normal) классом.

Один процесс может изменить или установить свой собственный приоритет или приоритет другого процесса, если это разрешено атрибутами защиты.

BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriority)

DWORD GetPriorityClass(HANDLE hProcess)

Приоритеты потоков устанавливаются относительно базового приоритета процесса, и во время создания потока его приоритет устанавливается равным приоритету процесса. Приоритеты потоков могут

принимать значения в интервале ±2 относительно базового приоритета процесса. Результирующим пяти значениям приоритета присвоены следующие символические имена:

1. THREAD_PRIORITY_LOWEST
2. THREAD_PRIORITY_BELOW_NORMAL
3. THREAD_PRIORITY_NORMAL
4. THREAD_PRIORITY_ABOVE_NORMAL
5. THREAD_PRIORITY_HIGHEST

Для установки и выборки относительного приоритета потока следует использовать эти значения.

BOOL SetThreadPriority(HANDLE hThread, int nPriority)
int GetThreadPriority(HANDLE hThread)

Существуют два дополнительных значения приоритета потоков. **Они являются абсолютными**, а не относительными, и используются только в специальных случаях.

• THREAD_PRIORITY_IDLE имеет значение 1 (или 16 — для процессов, выполняющихся в режиме реального времени).

• THREAD_PRIORITY_TIME_CRITICAL имеет значение 15 (или 31 — для процессов, выполняющихся в режиме реального времени).

Приоритеты потоков автоматически изменяются при изменении приоритета процесса. Помимо этого, ОС может регулировать приоритеты потоков динамическим путем на основании поведения потоков.

8. Взаимодействие процессов и потоков. Проблемы и задачи межпроцессного взаимодействия

Довольно часто процессам необходимо взаимодействовать с другими процессами. Например, в канале оболочки выходные данные одного процесса могут передаваться другому процессу, и так далее вниз по цепочке. Поэтому возникает необходимость во взаимодействии процессов, и желательно, по хорошо продуманной структуре без использования прерываний.

Способы взаимодействия процессов (потоков) можно классифицировать по степени осведомленности одного процесса о существовании другого.

1. Процессы ***не осведомлены*** о наличии друг друга (например, процессы разных заданий одного или различных пользователей). Это независимые процессы, не предназначенные для совместной работы. Хотя эти процессы и не работают совместно, ОС должна решать вопросы конкурентного использования ресурсов. Например, два независимых приложения могут затребовать доступ к одному и тому же диску или принтеру. ОС должна регулировать такие обращения.

2. Процессы ***косвенно осведомлены*** о наличии друг друга (например, процессы одного задания). Эти процессы не обязательно должны быть осведомлены о наличии друг друга с точностью до идентификатора процесса, однако они разделяют доступ к некоторому объекту, например, буферу ввода-вывода, файлу или БД. Такие процессы демонстрируют сотрудничество при разделении общего объекта.

3. Процессы ***непосредственно осведомлены*** о наличии друг друга (например, процессы, работающие последовательно или поочередно в рамках одного задания). Такие процессы способны общаться один с другим с использованием идентификаторов процессов и изначально созданы для совместной работы. Эти процессы также демонстрируют сотрудничество при работе.

Таким образом, потенциальные проблемы, связанные с взаимодействием и синхронизацией процессов и потоков, могут быть представлены следующей таблицей.

Степень осведомленности	**Взаимосвязь**	**Влияние одного процесса на другой**	**Потенциальные проблемы**
Процессы не осведомлены друг о друге	Конкуренция	Результат работы одного процесса НЕ зависит от действий других. Возможно влияние одного процесса на время работы другого	Взаимоисключения Взаимоблокировки Голодание
Процессы косвенно осведомлены о наличии друг друга	Сотрудничество с использованием разделения	Результат работы одного процесса МОЖЕТ зависеть от информации, полученной от других процессов. Возможно влияние одного процесса на время работы другого	Взаимоисключения Взаимоблокировки Голодание Синхронизация

Процессы непосредственно осведомлены о наличии друг друга	Сотрудничество с использованием связи	Результат работы одного процесса ЗАВИСИТ от информации, полученной от других процессов. Возможно влияние одного процесса на время работы другого	Взаимоблокировки (расходуемые ресурсы) Голодание

При необходимости использовать один и тот же ресурс параллельные процессы вступают в конфликт (конкурируют) друг с другом. Каждый из процессов не подозревает о наличии остальных и не повергается никакому воздействию с их стороны. Отсюда следует, что каждый процесс не должен изменять состояние любого ресурса, с которым он работает. Примерами таких ресурсов могут быть устройства ввода-вывода, память, процессорное время, часы.

Между конкурирующими процессами не происходит никакого обмена информацией. Однако выполнение одного процесса может повлиять на поведение конкурирующего процесса. Это может, например, выразиться в замедлении работы одного процесса, если ОС выделит ресурс другому процессу, поскольку первый процесс будит ждать завершения работы с этим ресурсом. В предельном случае блокированный процесс может никогда не получить доступ к нужному ресурсу и, следовательно, никогда не сможет завершиться.

В случае конкурирующих процессов (потоков) возможно возникновение трех проблем. Первая из них – необходимость взаимных исключений (mutual exclusion). Предположим, что два или большее количество процессов требуют доступ к одному неразделяемому ресурсу, как например принтер. О таком ресурсе будем говорить как о критическом ресурсе, а о части программы, которая его использует, – как о критическом разделе (critical section) программы. Крайне важно, чтобы в критической ситуации в любой момент могла находиться только одна программа. Например, во время печати файла требуется, чтобы отдельный процесс

имел полный контроль над принтером, иначе на бумаге можно получить чередование строк двух файлов.

Задачи межпроцессного взаимодействия:

1. обмен информацией между процессами,
2. недопущение конфликтных ситуаций,
3. согласование действий процессов.

9. Задачи межпроцессного взаимодействия взаимное исключение, синхронизация, обмен данными

Процессам часто нужно взаимодействовать друг с другом, например, один процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. Во всех этих случаях возникает проблема синхронизации процессов, которая может решаться приостановкой и активизацией процессов, организацией очередей, блокированием и освобождением ресурсов.

Задачи межпроцессного взаимодействия:

4. обмен информацией между процессами,
5. недопущение конфликтных ситуаций,
6. согласование действий процессов.

В случае конкурирующих процессов (потоков) возможно возникновение трех проблем.

Первая из них – необходимость **взаимных исключений** (mutual exclusion).

Осуществление взаимных исключений создает две дополнительные проблемы. Одна из них – **взаимоблокировки** (deadlock) или **тупики.** Рассмотрим, например, два процесса – P1 и P2, и два ресурса – R1 и R2. Предположим, что каждому процессу для выполнения части своих функций требуется доступ к общим ресурсам. Тогда возможно возникновение следующей ситуации: ОС выделяет ресурс R1 процессу P2, а ресурс R2 – процессу P1. В результате каждый процесс ожидает получения одного из двух ресурсов. При этом ни один из них не освобождает уже имеющийся

ресурс, ожидая получения второго ресурса для выполнения функций, требующих наличие двух ресурсов. В результате процессы оказываются взаимно заблокированными.

Синхронизация нужна для того, чтобы избежать:

1. *Гонок* (или взаимных состязаний) - возникают когда 2 или более процессов обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков.
2. *Тупиков* (взаимных блокировок, клинчей) - возникает когда два взаимосвязанных процесса блокируют друг друга при обращении к критической области.

Обмен информацией между процессами

Средства обмена информацией можно разделить на три категории:

1. **Сигнальные**. Передается минимальное количество информации – один бит, "да" или "нет".Используются, как правило, для извещения процесса о наступлении какого-либо события.
2. **Канальные**. «Общение» процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи.
3. **Разделяемая память**. Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием разделяемой памяти занимается операционная система (если, конечно, ее об этом попросят). «Общение» процессов напоминает совместное проживание студентов в одной комнате общежития. Возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, но требует повышенной осторожности (если вы переложили на другое место вещи вашего соседа по комнате, а часть из них еще и выбросили). Использование разделяемой памяти для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как сигнальным и канальным средствам коммуникации для этого необходимы специальные системные вызовы. Разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

10. Модели описания задач межпроцессного взаимодействия

В литературе по операционным системам можно встретить множество интересных проблем использования различных методов синхронизации, ставших предметом широких дискуссий и анализа.

Задача обедающих философов

В 1965 году Дейкстра сформулировал и решил проблему синхронизации, названную им задачей обедающих философов. С тех пор все изобретатели очередного примитива синхронизации считали своим долгом продемонстрировать его наилучшие качества, показав, насколько элегантно с его помощью решается задача обедающих философов.

Суть задачи довольно проста. Пять философов сидят за круглым столом и у каждого из них есть тарелка спагетти. Эти спагетти настолько скользкие, что есть их можно только двумя вилками. Между каждыми двумя тарелками лежит одна вилка. Жизнь философа состоит из чередующихся периодов приема пищи и размышлений. (Это положение из разряда абстракций даже в отношении философов, но вся их остальная деятельность к задаче не относится.) Когда философ становится голоден, он старается поочередно в любом порядке завладеть правой и левой вилкой. Если ему удастся взять две вилки, он на некоторое время приступает к еде, затем кладет обе вилки на стол и продолжает размышления.

Основной вопрос состоит в следующем: можно ли написать программу для каждого философа, который действует предполагаемым образом и никогда при этом не попадает в состояние зависания?

Задача читателей и писателей

Задача обедающих философов хороша для моделирования процессов, которые соревнуются за исключительный доступ к ограниченному количеству ресурсов, например к устройствам ввода-вывода. Другая общеизвестная задача касается читателей и писателей. В ней моделируется доступ к базе данных. Представим, к примеру, систему бронирования авиабилетов, в которой есть множество соревнующихся процессов, желающих обратиться к ней по чтению и записи. Вполне допустимо наличие нескольких процессов, одновременно считывающих информацию из базы данных, но если один процесс обновляет базу данных (проводит операцию записи), никакой другой процесс не может получить доступ к базе данных даже для чтения информации.

Вопрос в том, как создать программу для читателей и писателей?

Проблема спящего брадобрея

Действие еще одной классической проблемной ситуации межпроцессного взаимодействия разворачивается в парикмахерской. В парикмахерской есть один брадобрей, его кресло и n стульев для посетителей. Если желающих воспользоваться его услугами нет, брадобрей сидит в своем кресле и спит. Если в парикмахерскую приходит клиент, он должен разбудить брадобрея. Если клиент приходит и видит, что брадобрей занят, он либо садится на стул (если есть место), либо уходит (если места нет).

Необходимо запрограммировать брадобрея и посетителей так, чтобы избежать состояния состязания.

У этой задачи существует много аналогов в сфере массового обслуживания, например информационная служба, обрабатывающая одновременно ограниченное количество запросов, с компьютеризированной системой ожидания для запросов.

11. Задача взаимного исключения. Коллизии, критические ресурсы, критические секции

Задача взаимного исключения

При работе нескольких параллельных процессов с общими данными возникает необходимость взаимоисключать одновременный доступ процессов к данным. При этом участки программ процессов для работы с разделяемыми данными образуют так называемые критические области (секции). В общем виде постановка задачи взаимного исключения формулируется следующим образом: необходимо согласовать работу $n > 2$ параллельных процессов при использовании некоторого критического ресурса таким образом, чтобы удовлетворить следующим требованиям:

- одновременно внутри критической области должно находиться не более одного процесса;
- критические области не должны иметь приоритета по отношению друг к другу;
- остановка какого-либо процесса вне его критической области не должна влиять на дальнейшую работу процессов по использованию критического ресурса;
- решение о вхождении процессов в их критические области при одинаковом времени поступления запросов на такое вхождение и равноприоритетности процессов не откладывается на неопределенное время, а является конечным по времени;

- относительные скорости развития процессов неизвестны и произвольны;
- любой процесс может переходить в любое состояние, отличное от активного, вне пределов своей критической области;
- освобождение критического ресурса и выход из критической области должны быть произведены процессом, использующим критический ресурс, за конечное время.

Ресурс, который допускает обслуживание только одного пользователя за один раз, называется **критическим ресурсом**. Если несколько процессов хотят пользоваться критическим ресурсом в режиме разделения времени, им следует синхронизировать свои действия таким образом, чтобы этот ресурс всегда находился в распоряжении не более чем одного их них.

Критическая секция – участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком исполнения. При нахождении в критической секции двух (или более) процессов возникает состояние «гонки» («состязания»). Для избежания данной ситуации необходимо выполнение четырех условий:

1. Два процесса не должны одновременно находиться в критических областях.
2. В программе не должно быть предположений о скорости или количестве процессоров.
3. Процесс, находящийся вне критической области, не может блокировать другие процессы.
4. Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

12. Задача синхронизации. Синхронизация функциями ожидания. Событийное управление

Особенности каждого конкретного взаимодействия между двумя или более параллельными процессами определяются **задачей синхронизации.** Количество различных задач синхронизации неограниченно. Однако некоторые из них являются типичными. К ним относятся: **взаимное исключение, производители-потребители, читатели-писатели, обедающие философы** и т.д.

Большинство задач в реальных ОС по согласованию параллельных процессов можно решить либо с помощью этих типовых задач, либо с помощью их модификаций.

Способы синхронизации, основанные на глобальных переменных процесса, обладают существенным недостатком – они не подходят для синхронизации потоков различных процессов.

В таких случаях ОС должна предоставлять потокам системные объекты синхронизации, которые были бы видны для всех потоков, даже если они принадлежат разным процессам и работают в разных адресных пространствах.

Примерами таких синхронизирующих объектов являются системные *семафоры, мьютексы, события, таймеры и др*. Набор таких объектов определяется конкретной ОС. Чтобы разные процессы могли разделять синхронизирующие объекты, используются различные методы. Некоторые ОС возвращают указатель на объект. Этот указатель может быть доступен всем родственным процессам, наследующим характеристики общего родительского процесса. В других ОС процессы в запросах на создание объектов синхронизации указывают имена, которые должны им быть присвоены. Далее эти имена используются различными процессами для манипуляций объектами синхронизации. В этом случае работа с синхронизирующими объектами подобна работе с файлами. Их можно создавать, открывать, закрывать, уничтожать.

Ко второму классу объектов синхронизации относится **ожидающий таймер** (waitable timer).

К третьему классу объектов синхронизации относятся объекты, **которые переходят в сигнальное состояние** по завершении своей работы или при получении некоторого сообщения.

Примерами таких объектов синхронизации являются **потоки и процессы**. Пока эти объекты выполняются, они находятся в несигнальном состоянии. Если выполнение этих объектов заканчивается, то они переходят в сигнальное состояние.

Теперь перейдем к **функциям ожидания.**

Функции ожидания в Windows – это такие функции, параметрами которых являются объекты синхронизации. Эти функции обычно используются для блокировки потоков, которая выполняется следующим образом. Если дескриптор объекта синхронизации является параметром функции ожидания, а сам объект синхронизации находится в несигнальном

состоянии, то поток, вызвавший эту функцию ожидания, блокируется до перехода этого объекта синхронизации в сигнальное состояние.

Для ожидания перехода в сигнальное состояние одного объекта синхронизации используется функция **WaitForSingleObject**, которая имеет следующий прототип:

```
DWORD WaitForSingleObject(
  HANDLE hHandle,  // дескриптор объекта
  DWORD dwMilliseconds  // интервал ожидания в миллисекундах
);
```

Функция WaitForSingleObject в течение интервала времени, равного значению параметра dwMilliseconds, ждет пока объект синхронизации с дескриптором hHandle перейдет в сигнальное состояние. Если значениепараметра dwMilliseconds равно нулю, то функция только проверяет состояние объекта. Если же значение параметра dwMilliseconds равно INFINITE, то функция ждет перехода объекта синхронизации в сигнальное состояние бесконечно долго.

В случае удачного завершения **функция WaitForSingleObject возвращает** одно из следующих значений:

1. WAIT_OBJECT_0 означает, что объект синхронизации находился или перешел в сигнальное состояние.
2. WAIT_ABANDONED означает, что объектом синхронизации является мьютекс, который не был освобожден потоком, завершившим свое исполнение.
3. WAIT_TIMEOUT означает, что время ожидания истекло, а объект синхронизации не перешел в сигнальное состояние.

Событием называется оповещение о некотором выполненном действии. В программировании события используются для оповещения одного потока о том, что другой поток выполнил некоторое действие. Сама же и задача оповещения одного потока о некотором действии, которое совершил другой поток называется задачейусловной синхронизации или иногда задачей оповещения.

В операционных системах Windows события описываются объектами ядра Events. При этом **различают два типа событий**:

- события с ручным сбросом (можно перевести в несигнальное состояние только посредством вызова функции ResetEvent);
- события с автоматическим сбросом (переходит в несигнальное состояние как при помощи функции ResetEvent, так и при помощи функции ожидания).

Создаются события вызовом функции CreateEvent, которая имеет следующий прототип:

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES  lpSecurityAttributes, // атрибуты защиты
  BOOL        bManualReset,   // тип события, if (TRUE) ручной сброс else автоматический.
  BOOL        bInitialState, // начальное состояние события, if (TRUE) сигнальное else несигнальное.
  LPCTSTR lpName // имя события которое позволяет обращаться к нему из потоков, выполняющихся в разных процессах.
);
```

В случае удачного завершения функция CreateEvent возвращает дескриптор события, а в случае неудачи – значение NULL. Если событие с заданным именем уже существует, то функция CreateEvent возвращает дескриптор этого события, а функция GetLastError, вызванная после функции CreateEvent вернет значение ERROR_ALREADY_EXISTS.

13. Средства межпроцессного взаимодействия (IPC) - назначение, разновидности

IPC – inter process communication. Механизм межпроцессного взаимодействия при использовании разделенного адресного пространства.

1. Через **сигналы** – программный механизм, аналогичный прерыванию(без использования приоритетов), который информирует процесс о наступлении синхронного события.

   Минус – скорость выполнения. Плюс – зарезервированные константы.
2. Через **разделяемые файлы**.
3. Через **каналы** – именованная (неименованная) очередь сообщения по принципу FIFO. Записывает один поток - считывает другой.

Буфер обмена (clipboard)

Это одна из самых примитивных и хорошо известных форм IPC. Он появился еще в самых ранних версиях Windows. Основная его задача - обеспечивать обмен данными между программами по желанию и под контролем пользователя.

Сообщение WM_COPYDATA

Стандартное сообщение для передачи участка памяти другому процессу. Работает однонаправленно, принимающий процесс должен

расценивать полученные данные как read only. Посылать это сообщение необходимо только с помощью SendMessage, которая (напомню) в отличие от PostMessage ждет завершения операции. Таким образом, посылающий поток "подвисает" на время передачи данных. Вы сами должны решить, насколько это приемлемо для вас. Это не имеет значения для небольших кусков данных, но для больших объемов данных или для real-time приложений этот способ вряд ли подходит.

Разделяемая память (shared memory)

Этот способ взаимодействия реализуется не совсем напрямую, а через технологию File Mapping - отображения файлов на оперативную память. Вкратце, этот механизм позволяет осуществлять доступ к файлу таким образом, как будто это обыкновенный массив, хранящийся в памяти (не загружая файл в память явно). "Побочным эффектом" этой технологии является возможность работать с таким отображенным файлом сразу нескольким процессам. Таким образом, можно создать объект file mapping, но не ассоциировать его с каким-то конкретным файлом. Получаемая область памяти как раз и будет общей между процессами. Работая с этой памятью, потоки обязательно должны согласовывать свои действия с помощью объектов синхронизации.

Библиотеки динамической компоновки (DLL)

Библиотеки динамической компоновки также имеют способность обеспечивать обмен данными между процессами. Когда в рамках DLL объявляется переменная, ее можно сделать разделяемой (shared). Все процессы, обращающиеся к библиотеке, для таких переменных будут использовать одно и то же место в физической памяти. (Здесь также важно не забыть о синхронизации.)

Протокол динамического обмена данными (Dynamic Data Exchange, DDE)

Этот протокол выполняет все основные функции для обмена данными между приложениями. Он очень широко использовался до тех пор, пока для этих целей не стали применять OLE (впоследствии ActiveX). На данный момент DDE используется достаточно редко, в основном для обратной совместимости. Больше всего этот протокол подходит для задач, не требующих продолжительного взаимодействия с пользователем. Пользователю в некоторых случаях нужно только установить соединение между программами, а обмен данными происходит без его участия. Замечу, что все это в равной степени относится и к технологии OLE/ActiveX.

OLE/ActiveX

Это действительно универсальная технология, и одно из многих ее применений - межпроцессный обмен данными. Хотя стоит думаю отметить, что OLE как раз для этой цели и создавалась (на смену DDE), и только потом была расширена настолько, что пришлось поменять название ;-). Специально для обмена данными существует интерфейс IDataObject. А для обмена данными по сети используется DCOM, которую под некоторым углом можно рассматривать как объединение ActiveX и RPC.

Каналы (pipes)

Каналы - это очень мощная технология обмена данными. Наверное, именно поэтому в полной мере они поддерживаются только в Windows NT/2000. В общем случае канал можно представить в виде трубы, соединяющей два процесса. Что попадает в трубу на одном конце, мгновенно появляется на другом. Чаще всего каналы используются для передачи непрерывного потока данных. Каналы делятся на анонимные (anonymous pipes) и именованные (named pipes). Анонимные каналы используются достаточно редко, они просто передают поток вывода одного процесса на поток ввода другого. Именованные каналы передают произвольные данные и могут работать через сеть. (Именованные каналы поддерживаются только в WinNT/2000.)

Сокеты (sockets)

Это очень важная технология, т.к. именно она отвечает за обмен данными в Интернет. Сокеты также часто используются в крупных ЛВС. Взаимодействие происходит через т.н. разъемы-"сокеты", которые представляют собой абстракцию конечных точек коммуникационной линии, соединяющей два приложения. С этими объектами программа и должна работать, например, ждать соединения, посылать данные и т.д. В Windows входит достаточно мощный API для работы с сокетами.

Почтовые слоты (mailslots)

Почтовые слоты - это механизм однонаправленного IPC. Если приложению известно имя слота, оно может помещать туда сообщения, а приложение-хозяин этого слота (приемник) может их оттуда извлекать и соответствующим образом обрабатывать. Основное преимущество этого способа - возможность передавать сообщения по локальной сети сразу нескольким компьютерам за одну операцию. Для этого приложения-приемники создают почтовые слоты с одним и тем же именем. Когда в дальнейшем какое-либо приложение помещает сообщение в этот слот, приложения-приемники получают его одновременно.

Объекты синхронизации

Как ни странно, объекты синхронизации тоже можно отнести к механизмам IPC.

Microsoft Message Queue (MSMQ)

Этот протокол действительно оправдывает свое название - он обеспечивает посылку сообщений между приложениями с помощью очереди сообщений. Основное его отличие от стандартной очереди сообщений Windows в том, что он может работать с удаленными процессами и даже с процессами, которые на данный момент недоступны (например, не запущены). Доставка сообщения по адресу гарантируется. Оно ставится в специальную очередь сообщений и находится там до тех пор, пока не появляется возможность его доставить.

Удаленный вызов процедур (Remote Procedure Call, RPC)

Строго говоря, это не совсем технология IPC, а скорее способ значительно расширить возможности других механизмов IPC. С помощью этой технологии общение через сеть становится совешенно прозрачным как для сервера, так и для клиента. Им обоим начинает казаться, что их "собеседник" расположен локально по отношению к ним .

14. Концепция событий. Событийное управление

Событием называется оповещение о некотором выполненном действии. В программировании события используются для оповещения одного потока о том, что другой поток выполнил некоторое действие. Сама же и задача оповещения одного потока о некотором действии, которое совершил другой поток называется задачей условной синхронизации или иногда задачей оповещения.

В операционных системах Windows события описываются объектами ядра Events. При этом **различают два типа событий**:

- события с ручным сбросом (можно перевести в несигнальное состояние только посредством вызова функции ResetEvent);
- события с автоматическим сбросом (переходит в несигнальное состояние как при помощи функции ResetEvent, так и при помощи функции ожидания).

Создаются события вызовом функции CreateEvent, которая имеет следующий прототип:

```
HANDLE CreateEvent(
 LPSECURITY_ATTRIBUTES  lpSecurityAttributes, // атрибуты защиты
```

BOOL bManualReset, // тип события, **if (TRUE)** *ручной сброс* **else** *автоматический*.
BOOL bInitialState, // начальное состояние события, **if (TRUE)** *сигнальное* **else** *несигнальное*.
LPCTSTR lpName // имя события которое позволяет обращаться к нему из потоков, выполняющихся в разных процессах.
);

В случае удачного завершения функция CreateEvent возвращает дескриптор события, а в случае неудачи – значение NULL. Если событие с заданным именем уже существует, то функция CreateEvent возвращает дескриптор этого события, а функция GetLastError, вызванная после функции CreateEvent вернет значение ERROR_ALREADY_EXISTS.

15. Сигналы, сообщения, прерывания в межпроцессном взаимодействии

Сигналы – одно из традиционных средств межпроцессного взаимодействия в UNIX. Сигнал может быть отправлен процессу операционной системой или другим процессом. Операционная система использует сигналы для доставки процессу уведомлений об ошибках и неправильном поведении.

При получении сигнала исполнение процесса приостанавливается и запускается специальная подпрограмма – *обработчик сигнала*. Обработчики сигналов могут быть явно определены в исходном тексте исполняемой программы, если же они отсутствуют, используется стандартный обработчик, определённый операционной системой.

У сигнала есть только одна характеристика, несущая информацию – его номер (целое число). Иначе говоря, сигналы – это заранее определённый и пронумерованный список сообщений. Для удобства использования каждый сигнал имеет сокращённое буквенное имя. Список сигналов и их имён стандартизован и практически не отличается в различных версиях UNIX. Для отправки сигналов процессам используется специальный системный вызов kill и одноимённая ему пользовательская утилита. Стандарт POSIX определяет 28 сигналов, вот несколько примеров:

1. SIGINT (2) — Сигнал передается активному приложению при нажатии сочетания Ctrl+C, по умолчанию завершает процесс.
2. SIGKILL (9) — Сигнал аварийного завершения процесса. По этому сигналу процесс завершается немедленно — без

освобождения ресурсов. Этот сигнал не может быть перехвачен, заблокирован или переопределён самим процессом, всегда используется стандартный обработчик опрационной системы. Этот сигнал используется для гарантированного завершения процесса.

3. SIGTERM (15) — Сигнал завершения процесса, как правило используется для корректного завершения его работы.
4. SIGUSR1 (10) и SIGUSR2 (12) — Зарезервированные сигналы под нужды программистов.

Сигналы являются ограниченным средством межпроцесснного обмена. Они прекрасно подходят для уведомлений, но не могут использоваться для передачи информации между процессами. Сигналы передаются без каких-либо сопутствующих данных, поэтому они обычно комбинируются с другими способами обмена.

Вызовы POSIX — *signal*(), *kill*().

Передача сообщений

В *UNIX* ***сообщения*** используются для решения проблемы согласованности процессов при передаче информации, существуют специальные механизмы обмена данными, основанные на мьютексах. Это системные вызовы *send()* и *recieve()*. Первый посылает сообщение заданному адресату, второй получает сообщение от указанного источника. Если сообщения нет, второй запрос блокируется до поступления сообщения либо немедленно возвращает код ошибки.

В *Windows* ***сообщения*** используются для передачи данных. Сообщение *WM_COPYDATA* – стандартное сообщение для передачи участка памяти другому процессу. Работает однонаправленно, принимающий процесс должен расценивать полученные данные как read only. Посылать это сообщение необходимо только с помощью SendMessage, которая в отличие от PostMessage ждет завершения операции.

Сигналы являются программными прерываниями, которые посылаются процессу, когда случается некоторое событие. Сигналы могут возникать синхронно с ошибкой в приложении, например **SIGFPE** (ошибка вычислений с плавающей запятой) и **SIGSEGV** (ошибка адресации), но большинство сигналов является асинхронными. Сигналы могут посылаться процессу, если система обнаруживает программное событие, например, когда пользователь дает команду прервать или остановить выполнение, или

получен сигнал на завершение от другого процесса. Сигналы могут прийти непосредственно от ядра ОС, когда возникает сбой аппаратных средств ЭВМ. Система определяет набор сигналов, которые могут быть отправлены процессу. При этом каждый сигнал имеет целочисленное значение и приводит к строго определенным действиям.

Механизм передачи сигналов состоит из следующих частей:

1. установление и обозначение сигналов в форме целочисленных значений;
2. маркер в строке таблицы процессов для прибывших сигналов;
3. таблица с адресами функций, которые определяют реакцию на прибывающие сигналы.

Отдельные сигналы подразделяются на три класса:

1. системные сигналы (ошибка аппаратуры, системная ошибка и т.д.);
2. сигналы от устройств;
3. сигналы, определенные пользователем.

Как только сигнал приходит, он отмечается записью в таблице процессов. Если этот сигнал предназначен для процесса, то по таблице указателей функций в структуре описания процесса выясняется, как нужно реагировать на этот сигнал. При этом номер сигнала служит индексом таблицы.

Известно три варианта реакции на сигналы:

1. вызов собственной функции обработки;
2. игнорирование сигнала (не работает для SIGKILL);
3. использование предварительно установленной функции обработки по умолчанию.

Чтобы реагировать на разные сигналы, необходимо знать концепции их обработки. Процесс должен организовать так называемый обработчик сигнала в случае его прихода.

16. Сообщения и очереди сообщений. Использование очередей сообщений в межпроцессном взаимодействии

Очереди сообщений представляют собой связный список в адресном пространстве ядра. Сообщения могут посылаться в очередь по порядку и доставаться из очереди несколькими разными путями. Каждая очередь сообщений однозначно определена идентификатором IPC.

Очереди сообщений как средство межпроцессной связи позволяют процессам взаимодействовать, обмениваясь данными. Данные передаются между процессами дискретными порциями, **называемыми сообщениями.** Процессы, использующие этот тип межпроцессной связи, могут выполнять две операции: **послать или принять сообщение.**

Процесс, прежде чем послать или принять какое-либо сообщение, должен запросить систему породить программные механизмы, необходимые для обработки данных операций. Процесс делает это при помощи системного вызова **msgget**. Обратившись к нему, процесс становится владельцем или создателем некоторого средства обмена сообщениями; кроме того, процесс специфицирует первоначальные права на выполнение операций для всех процессов, включая себя. Впоследствии владелец может уступить право собственности или изменить права на операции при помощи системного вызова **msgctl**, однако на протяжении всего времени существования определенного средства обмена сообщениями его создатель остается создателем. Другие процессы, обладающие соответствующими правами для выполнения различных управляющих действий, также могут использовать системный вызов **msgctl**.

Процессы, имеющие права на операции и пытающиеся послать или принять сообщение, могут приостанавливаться, если выполнение операции не было успешным. В частности это означает, что процесс, пытающийся послать сообщение, может ожидать, пока процесс-получатель не будет готов; и наоборот, получатель может ждать отправителя. Если указано, что процесс в таких ситуациях должен приостанавливаться, говорят о выполнении над сообщением ``операции с блокировкой". Если приостанавливать процесс нельзя, говорят, что над сообщением выполняется "операция без блокировки".

Процесс, выполняющий операцию с блокировкой, может быть приостановлен до тех пор, пока не будет удовлетворено одно из условий:

1. операция завершилась успешно;
2. процесс получил сигнал;

3. очередь сообщений ликвидирована.

Системные вызовы позволяют процессам пользоваться этими возможностями обмена сообщениями. Вызывающий процесс передает системному вызову аргументы, а системный вызов выполняет (успешно или нет) свою функцию. Если системный вызов завершается успешно, он выполняет то, что от него требуется, и возвращает некоторую содержательную информацию. В противном случае процессу возвращается значение -1, известное как признак ошибки, а внешней переменной errno присваивается код ошибки.

17. Файлы, каналы, почтовые ящики в межпроцессном взаимодействии. Особенности обмена данными посредством каналов

Использование файлов, отображаемых в память

Отображение файлов в память позволяет процессу интерпретировать содержимое файла так, как будто оно является блоком памяти в его адресном пространстве. Процесс может использовать простые операции для чтения и модификации содержимого файла. Когда два или более процесса получают доступ к одному и тому же отображенному в память файлу (данным), каждый процесс получает указатель на память в своем собственном адресном пространстве и может читать или модифицировать содержимое файла. Процессы должны использовать синхронизирующие объекты, такие как семафоры, для того чтобы предотвратить разрушение данных в многозадачной среде.

Можно также использовать специальный вид отображения файлов, называемый именованной **разделяемой памятью** (*named shared memory*). Если создаваемый объект отображения файла в память во время его создания определить как системный файл подкачки, то он будет интерпретироваться как блок разделяемой памяти. Разные процессы могут получить доступ к одному и тому же блоку памяти путем открытия одного и того же объекта.

Отображение файла в память также обеспечивает поддержку атрибутов безопасности ОС, которая может помочь предохранить данные от несанкционированного доступа. Отображение файла в память может быть использовано только для взаимодействия между процессами на одном компьютере, но не в сети.

Таким образом, отображаемые в память файлы предоставляют возможность нескольким процессам получать совместный доступ к данным, однако процессы должны обеспечивать синхронизацию доступа к данным.

Каналы

Канал – поток данных между двумя или несколькими процессами, имеющий интерфейс, аналогичный чтению или записи в файл. Каналы бывают одно- и двунаправленными. В UNIX каналы, как и многие другие системные объекты, представлены в виде файлов, вся работа с ними производится через базовый файловый интерфейс — открытие и закрытие файла, чтение и запись данных и т. п.

В этом смысле каналы можно представлять в виде специализированных файлов, которые не хранят информацию, а лишь накапливают её до следующей операции чтения из канала другим процессом, образуя очередь.

Почтовые ящики (MailSlots)

Подобны именованным каналы, но предоставляют более простой однонаправленный интерфейс. Процесс-сервер может создать канал и дать ему имя, глобальное в сети. Любой клиент может с помощью операций работы с файлами отправить данные в этот ящик. Сервер по мере необходимости читает переданные ему данные. Возможна также широковещательная передача информации клиентом всем серверам домена.

Почтовый ящик создается функцией CreateMailslot , причем используется имя вида \\.\mailslot\[<*path*>\]<*slotname*>, где собственно имя ящика может включать символы ‘\’, не имея соответствия в реальной структуре каталогов (т.н. *псевдодиректории*).

Для доступа к ящику файловыми функциями его надо открыть функцией CreateFile с использованием следующих вариантов имени файла:

1. \\.\mailslot\<*slot_name*> – локальный ящик;
2. \\<*computer_name*>\mailslot\<*slot_name*> – удаленный ящик;
3. \\<*domain_name*>\mailslot\<*slot_name*> – используя доменное имя;
4. *\mailslot\<*slot_name*> – используя первичный (primary) домен системы.

В ящик, открытый через доменное имя, **нельзя записывать более 400 байт в один прием**.

Если ящик будет использоваться более чем одним процессом (владельцем), то CreateFile должен задавать флаг доступа FILE_SHARE_READ.

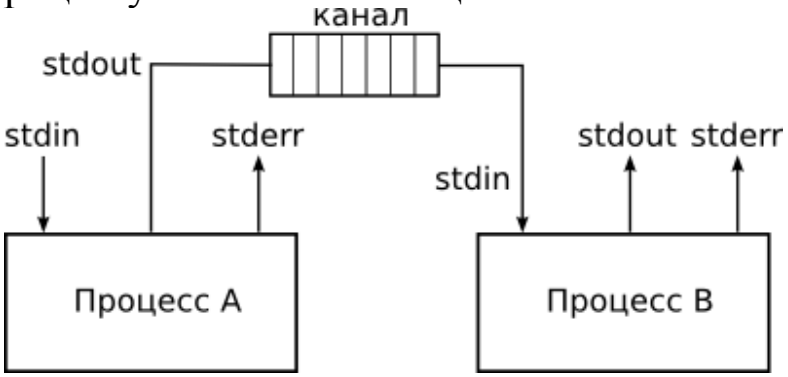
Созданный ящик существует до тех пор, пока существует процесс-владелец, либо пока владелец или его потомок не удалят его описатель вызовом CLOSE_HANDLE.

Резюме. Почтовые ящики предоставляют простой способ для приложений посылать и получать сравнительно короткие сообщения. Они также предоставляют простую возможность рассылать сообщения всем компьютерам домена.

Особенности обмена данными посредством каналов.

По умолчанию в UNIX каждому процессу при запуске ставится в соответствие три открытых файла: *стандартного ввода, стандартного вывода и стандартного вывода для ошибок*. С помощью средств командной строки такие потоки для разных процессов могут быть объединены так, что, к примеру, вывод одного процесса будет подаваться на ввод другого. То есть процесс работает с тремя потоками данных одинаково вне зависимости от того, обычные это файлы или же каналы.

В более общем смысле такие потоки называют **неименованными каналами**. Канал создаётся по запросу и существует только в ходе работы двух процессов, другие процессы в системе не могут обратиться к этому каналу. Если процесс на одной из сторон канала завершается и закрывает канал, другому процессу посылается специальный сигнал — SIGPIPE.



Другой вид каналов в UNIX — именованные каналы — представляют собой особый тип файлов. Эти файлы располагаются в файловой системе и могут быть открыты любым процессом. Одни

процессы записывают данные в канал, другие — читают из него, данные продвигаются по каналу в порядке очереди (FIFO).

Каналы широко используются в UNIX, как при запуске программ в командной строке, так и при взаимодействии системных процессов. **Главное достоинство каналов** — простота и удобство использования привычного файлового интерфейса. С другой стороны, данные в каналах передаются в одном направлении и последовательно, что ограничивает сферу применения каналов.

Вызов POSIX — *pipe*().

18. Разделяемая память и проецирование файлов в межпроцессном взаимодействии. Особенности обмена данными посредством разделяемой памяти

Разделяемая память (shared memory)

Этот способ взаимодействия реализуется не совсем напрямую, а через технологию File Mapping - отображения файлов на оперативную память. Этот механизм позволяет осуществлять доступ к файлу таким образом, как будто это обыкновенный массив, хранящийся в памяти (не загружая файл в память явно). Отсюда появляется возможность работать с таким отображенным файлом сразу нескольким процессам. Таким образом, можно создать объект **file mapping**, но не ассоциировать его с каким-то конкретным файлом. Получаемая область памяти как раз и будет общей между процессами.

После создания разделяемого сегмента памяти любой из пользовательских процессов может подсоединить его к своему собственному виртуальному пространству и работать с ним, как с обычным сегментом памяти. Недостатком такого обмена информацией является отсутствие каких бы то ни было средств синхронизации, однако для преодоления этого недостатка можно использовать технику семафоров или других объектов синхронизации.

Примерный сценарий использования разделяемой памяти при реализации технологий «клиент—сервер» имеет вид:

- сервер получает доступ к разделяемой памяти, используя семафор;
- сервер производит запись данных в разделяемую память;

- после завершения записи данных сервер освобождает доступ к разделяемой памяти с помощью семафора;
- клиент получает доступ к разделяемой памяти, запирая доступ к этой памяти для других процессов с помощью семафора;
- клиент производит чтение данных из разделяемой памяти, а затем освобождает доступ к памяти с помощью семафора.

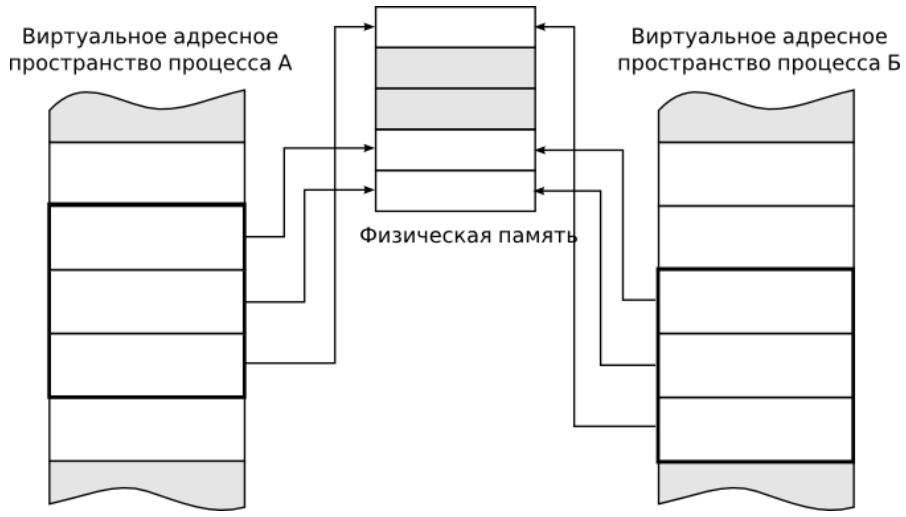
Для работы с разделяемой памятью используются системные вызовы:

- shmget — создание сегмента разделяемой памяти;
- shmctl — установка параметров;
- shmat — подсоединение сегмента памяти;
- shmdt — отсоединение сегмента.

В схеме обмена данными между двумя процессами — клиентом и сервером, использующими разделяемую память, — должна функционировать группа из двух семафоров.

Первый семафор служит для блокирования доступа к разделяемой памяти, его разрешающий сигнал — 1, а запрещающий — 0.

Второй семафор служит для сигнализации сервера о том, что клиент начал работу, при этом доступ к разделяемой памяти блокируется, и клиент читает данные из памяти. Теперь при вызове операции сервером его работа будет приостановлена до освобождения памяти клиентом.



19. Функции ожидания, объекты ожидания, их использование их для синхронизации

Для более сложных методов синхронизации, в том числе между потоками различных процессов, используются т.н. ***функции ожидания***:

WaitForSingleObject (для ожидания одного объекта синхронизации), WaitForSingleObjectEx, SignalObjectAndWait, WaitForMultipleObjects (несколько объектов синхронизации – ожидаются все или хотя бы один объект, в зависимости от параметра), WaitForMultipleObjectsEx, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx.

Основные параметры функций – дескриптор объекта ожидания и таймаут, максимальное время ожидания.

Их действие основано на использовании **глобальных (контролируемых ядром системы) объектах**, состояния которых могут интерпретироваться как “signaled” (приблизительно можно переводить как "установлено", "открыто", фактически означает "свободность", т.е. "не-занятость" объекта) и “non-signaled” (означает "занятость").

Вызов функции ожидания применительно к объекту в состоянии “signaled” дает немедленное успешное завершение, причем состояние объекта может быть изменено в зависимости от его типа, а в состоянии “non-signaled” – перевод вызвавшего процесса в состояние ожидания (блокировки) до "освобождения" объекта.

Практически любые объекты ядра (процессы, файлы, каналы и т.д.) могут выступать в роли синхронизирующих, однако существует набор объектов, специально предназначенных для синхронизации потоков – т.н. **Interprocess Synchronization Objects (ISO)**.

Отметим, что файловые объекты, выступая в качестве синхронизирующих, выполняют важную функцию – обеспечиваю работу асинхронного ввода-вывода.

Все объекты ISO идентифицируются их именами, причем из единого пространства имен, не пересекающегося с именами объектов файловой системы. Для доступа к ним служат дескрипторы (тип HANDLE).

Создание объекта выполняется вызовом Create... (можно выполнять его несколько раз, получая доступ к одному и тому же объекту), доступ к существующему – Open..., дескриптор объекта можно дублировать и наследовать, он удаляется явно функцией CloseHandle, либо автоматически – при завершении процесса-владельца.

Объект существует, пока действителен хотя бы один его дескриптор.

20. Семафоры, мьютексы, барьеры, события (event). Особенности, использование для синхронизации процессов и потоков

С помощью объектов **семафор** реализуется классический механизм семафоров. При этом в качестве примитива P(S) выступает одна из функций ожидания (уменьшает счетчик на единицу, но не меньше нуля, или ждет такой возможности), а в качестве примитива V(S) – функция ReleaseSemaphore (увеличивает счетчик семафора на заданную величину). При создании семафора функцией CreateSemaphore можно определить максимально допустимое значение переменной S.

POSIX – sem_init(), sem_wait(), sem_post(), sem_destroy().

Windows – ReleaseSemaphore, CreateSemaphore, WaitForSingleObject

Mutex – глобальный именованный объект ядра, который одновременно может принадлежать (быть используемым) только одному потоку: если объект Mutex захвачен каким-либо потоком, то остальные потоки не могут захватить его. С помощью объектов Mutex реализуется классический алгоритм критической секции. Доступ к объектам Mutex осуществляется посредством описателей. Общая схема такова: один из потоков порождает объект Mutex и присваивает ему глобальное имя при помощи функции CreateMutex. Остальные потоки могут использовать для получения описателя объекта OpenMutex или CreateMutex с тем же именем. Перед входом в критическую секцию поток вызывает одну из функций ожидания, передавая ей в качестве параметра описатель объекта Mutex. Если объект Mutex уже захвачен, то поток блокируется до освобождения объекта. Если не захвачен, то исполнение потока продолжается, а объект Mutex захватывается. Для освобождения объекта Mutex используется функция ReleaseMutex.

POSIX – mutex_init(), mutex_lock(), mutex_unlock(), mutex_destroy();

Windows – CreateMutex, ReleaseMutex, WaitForSingleObject

События – самая примитивная разновидность синхронизирующих объектов. Они порождаются функцией CreateEvetnt и бывают двух типов с “ручным” (manual) и с “автоматическим” сбросом. Объект "событие" может находиться в двух состояниях: “занят” (non-signaled) и “свободен” (signaled). Для перевода объекта событие в свободное состояние используется функция SetEvent, а для перевода в занятое – ResetEvent. С помощью любой из функций ожидания можно перевести вызвавший поток

в состояние блокировки до освобождения объекта событие. Если объект событие является объектом “с автоматическим сбросом”, то функция ожидания автоматически переведет его в состояние занятого.

Объекты "событие" активно используются при асинхронном файловом вводе/выводе.

Барьер – это механизм синхронизации, который позволяет логически интегрировать результаты работы нескольких потоков, заставляя их ждать на определенной точке до тех пор, пока число заблокированных потоков не достигнет установленного значения. Только после этого все потоки деблокируются и могут продолжить свое исполнение. Когда специфицированный набор потоков достигнет барьера, все потоки деблокируются и продолжают исполнение.

Создание: int pthread_barrier_init(указатель к барьеру, атрибуты барьера, счетчик барьера).

Счетчик определяет число потоков, которые должны вызывать функцию int pthread_barrier_wait(pthread_barrier_t *barrier);

После создания барьера каждый поток будет вызывать эту функцию для указания на завершение этапа вычислений. При вызове данной функции поток блокируется до тех пор, пока число блокированных потоков не достигнет значения счетчика. После этого все потоки будут деблокированы. При создании барьера формируется указатель на объект атрибутов, который далее должен быть инициализирован.

Атрибут разделения барьера может иметь 2 значения:

- PTHREAD_PROCESS_SHARED – разрешает барьеру синхронизировать потоки различных процессов;
- PTHREAD_PROCESS_PRIVATE (default) – разрешает барьеру синхронизировать потоки в пределах одного процесса.

21. Виды организации ввода-вывода. Асинхронный ввод-вывод особенности, применение

Операция ввода-вывода может выполняться по отношению к программному модулю, запросившему операцию, в синхронном или асинхронном режимах. Смысл этих режимов в следующем:

- **синхронный** режим означает, что программный модуль приостанавливает свою работу до тех пор, пока операция ввода-вывода не будет завершена

- **асинхронный** режим означает, что программный модуль продолжает выполняться в мультипрограммном режиме одновременно с операцией ввода-вывода.

Отличие же заключается в том, что операция ввода-вывода может быть инициирована не только пользовательским процессом — в этом случае операция выполняется в рамках системного вызова, но и кодом ядра, например кодом подсистемы виртуальной памяти для считывания отсутствующей в памяти страницы.

Подсистема ввода-вывода должна предоставлять своим клиентам (пользовательским процессам и кодам ядра) возможность выполнять как синхронные, так и асинхронные операции ввода-вывода, в зависимости от потребностей вызывающей стороны.

Системные вызовы ввода-вывода чаще оформляются как **синхронные** процедуры в связи с тем, что такие операции длятся долго и пользовательскому процессу или потоку все равно придется ждать получения результатов операции для того, чтобы продолжить свою работу.

Внутренние же вызовы операций ввода-вывода из модулей ядра обычно выполняются в виде **асинхронных** процедур, так как кодам ядра нужна свобода в выборе дальнейшего поведения после запроса операции ввода-вывода. Использование асинхронных процедур приводит к более гибким решениям, так как на основе асинхронного вызова всегда можно построить синхронный, создав дополнительную промежуточную процедуру, блокирующую выполнение вызвавшей процедуры до момента завершения ввода-вывода. Иногда и прикладному процессу требуется выполнить асинхронную операцию ввода-вывода, например при микроядерной архитектуре, когда часть кода работает в пользовательском режиме как прикладной процесс, но выполняет функции операционной системы, требующие полной свободы действий и после вызова операции ввода-вывода.

22. Сокеты. Программирование с использованием сокетов стека TCPIP

Сокеты предоставляют весьма мощный и гибкий механизм межпроцессного взаимодействия. **Сокеты обеспечивают двухстороннюю связь типа ``точка-точка'' между двумя процессами**. Они являются основными компонентами межсистемной и межпроцессной связи. Каждый

сокет представляет собой конечную точку связи, с которой может быть совмещено некоторое имя. Он имеет определенный тип, и один процесс или несколько, связанных с ним процессов.

С каждым сокетом связываются три атрибута: **домен, тип и протокол**. Эти атрибуты задаются при создании сокета и остаются неизменными на протяжении всего времени его существования.

Для создания сокета используется функция socket, имеющая следующий прототип: **int socket(int domain, int type, int protocol);**

Домен определяет пространство адресов, в котором располагается сокет, и множество протоколов, которые используются для передачи данных. Тип сокета определяет способ передачи данных по сети. Последний атрибут определяет протокол, используемый для передачи данных. Часто протокол однозначно определяется по домену и типу сокета. В этом случае в качестве третьего параметра функции socket можно передать 0, что соответствует протоколу по умолчанию. Тем не менее, иногда требуется задать протокол явно. Прежде чем передавать данные через сокет, его необходимо связать с адресом в выбранном домене.

Для явного связывания сокета с некоторым адресом используется функция **bind.**

Основные **типы** сокетов

1. **Поточный** - обеспечивает двухсторонний, последовательный, надежный, и недублированный поток данных без определенных границ. Тип сокета - SOCK_STREAM, в домене Интернета он использует протокол TCP.
2. **Датаграммный** - поддерживает двухсторонний поток сообщений. Приложение, использующее такие сокеты, может получать сообщения в порядке, отличном от последовательности, в которой эти сообщения посылались. Тип сокета - SOCK_DGRAM, в домене Интернета он использует протокол UDP.
3. **Сокет последовательных пакетов** - обеспечивает двухсторонний, последовательный, надежный обмен датаграммами фиксированной максимальной длины. Тип сокета - SOCK_SEQPACKET. Для этого типа сокета не существует специального протокола.

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

Установка соединения на стороне сервера состоит из **четырёх этапов.**

Сначала сокет создаётся и привязывается к локальному адресу. На следующем шаге создаётся очередь запросов на соединение. При этом сокет переводится в режим ожидания запросов со стороны клиентов. Всё это выполняет функция **listen**. Функция accept создаёт для общения с клиентом новый сокет и возвращает его дескриптор. Один из параметров, **sockfd** задаёт слушающий сокет. После вызова он остаётся в слушающем состоянии и может принимать другие соединения. На стороне клиента для установления соединения используется функция connect. После того как соединение установлено, можно начинать обмен данными. Для этого используются функции **send** и **recv**. В Unix для работы с сокетами можно использовать также файловые функции read и write, но они обладают меньшими возможностями, а кроме того не будут работать на других платформах.

После окончания обмена данными, сокет закрывают с помощью функции **close**. Это приводит к разрыву соединения. Также можно запретить передачу данных в каком-то одном направлении, используя функцию **shutdown.** В зависимости от переданного параметра можно запретить запись в сокет, чтение из него или и то и другое.

23. Модель взаимодействия клиент-сервер назначение, особенности, использование

Клиент-сервер – это модель взаимодействия процессов в вычислительной системе, при которой один процесс (клиент) делает запрос, другой процесс (сервер) его обрабатывает и возвращает первому ответ или предоставляет определенную услугу в виде вычислений, каких-либо данных и т.п.

Примерами **серверов** могут служить:

- сервер телекоммуникаций, обеспечивающий услуги по связи данной локальной сети с внешним миром;
- вычислительный сервер, дающий возможность производить вычисления, которые невозможно выполнить на рабочих станциях;

- дисковый сервер, обладающий расширенными ресурсами внешней памяти и предоставляющий их в использование рабочим станциями и, возможно, другим серверам;
- файловый сервер, поддерживающий общее хранилище файлов для всех рабочих станций;
- сервер баз данных фактически обычная СУБД, принимающая запросы по локальной сети и возвращающая результаты.

Системная архитектура "клиент-сервер"

Система разбивается на две части, которые могут выполняться в разных узлах сети, - клиентскую и серверную части. Прикладная программа или конечный пользователь взаимодействуют с клиентской частью системы, которая в простейшем случае обеспечивает просто надсетевой интерфейс. Клиентская часть системы при потребности обращается по сети к серверной части. Заметим, что в развитых системах сетевое обращение к серверной части может и не понадобиться, если система может предугадывать потребности пользователя, и в клиентской части содержатся данные, способные удовлетворить его следующий запрос.

Интерфейс серверной части определен и фиксирован. Поэтому возможно создание новых клиентских частей существующей системы (пример интероперабельности на системном уровне).

Основной **проблемой** систем, основанных на архитектуре "клиент-сервер", является то, что в соответствии с концепцией открытых систем от них требуется мобильность в как можно более широком классе аппаратно-программных решений открытых систем. Даже если ограничиться UNIX-ориентированными локальными сетями, в разных сетях применяется разная аппаратура и протоколы связи. Попытки создания систем, поддерживающих все возможные протоколы, приводит к их перегрузке сетевыми деталями в ущерб функциональности.

Еще более сложный аспект этой проблемы связан с возможностью использования разных представлений данных в разных узлах неоднородной локальной сети. В разных компьютерах может существовать различная адресация, представление чисел, кодировка символов и т.д. Это особенно существенно для серверов высокого уровня: телекоммуникационных, вычислительных, баз данных.

Общим решением проблемы мобильности систем, основанных на архитектуре "клиент-сервер" является опора на программные пакеты,

реализующие протоколы ***удаленного вызова процедур*** (**RPC** - Remote Procedure Call). При использовании таких средств обращение к сервису в удаленном узле выглядит как обычный вызов процедуры. Средства RPC, в которых, естественно, содержится вся информация о специфике аппаратуры локальной сети и сетевых протоколов, переводит вызов в последовательность сетевых взаимодействий. Тем самым, специфика сетевой среды и протоколов скрыта от прикладного программиста.

При вызове удаленной процедуры программы RPC производят преобразование форматов данных клиента в промежуточные машинно-независимые форматы и затем преобразование в форматы данных сервера. При передаче ответных параметров производятся аналогичные преобразования.

Если система реализована на основе стандартного пакета RPC, она может быть легко перенесена в любую открытую среду.

24. Обработка множественных запросов. Параллельная реализация сервера (многопроцессный, многопоточный)

По сути существуют 3 модели работы сервера:

Простой последовательный сервер

Сервер открывает слушающий сокет и ждет, когда появится соединение (во время ожидания он находится в заблокированном состоянии). Когда приходит соединение, сервер обрабатывает его в том же контексте, закрывает соединение и снова ждет соединения. Очевидно, это далеко не самый лучший способ, особенно когда работа с клиентом ведется достаточно долго и подключений много. Кроме того, у последовательной модели есть еще много недостатков (например, невозможность использования нескольких процессоров), и в реальных условиях она практически не используется.

Это самая простая и понятная форма сервера: приняв запрос он его обрабатывает, и до завершения обработки недоступен для новых запросов. Обычно используется как эталон для сравнения: он имеет минимальное время реакции, так как не затрачивается время на порождение каких-либо механизмов параллелизма.

Многопроцессная (многопоточная).

Сервер открывает слушающий сокет. Когда приходит соединение, он принимает его, после чего создает (или берет из пула заранее созданных) новый процесс или поток, который может сколь угодно долго работать с соединением, а по окончании работы завершиться или вернуться в пул. Главный поток тем временем готов принять новое соединение. Это наиболее популярная модель, потому что она относительно просто реализуется, позволяет выполнять сложные и долгие вычисления для каждого клиента и использовать все доступные процессоры. Однако у этого подхода есть и недостатки: при большом количестве одновременных подключений создается очень много потоков (или, что еще хуже, процессов), и операционная система тратит много ресурсов на переключения контекста. Особенно плохо, когда клиенты очень медленно принимают контент. Получаются сотни потоков или процессов, занятых только отправкой данных медленным клиентам, что создает дополнительную нагрузку на планировщик ОС, увеличивает число прерываний и потребляет достаточно много памяти.

Классический параллельный сервер

Классическая реализация параллельного сервера: по поступлению запроса порождается (вызовом fork()) новый обслуживающий процесс (клон родительского процесса). Родительский процесс при этом возвращается в режим прослушивания следующих поступающих запросов. После получения запроса от клиента и порождения отдельного процесса, он закрывает свою копию прослушивающего сокета, производит ретрансляцию через соединённый сокет, завершает соединение и завершается сам. Родительский же процесс закрывает свою копию соединённого сокета и продолжает прослушивание канала.

Параллельный сервер, создающий потоки по запросам

Классический параллельный сервер, но вместо параллельных клонов процессов теперь будем порождать параллельные потоки в том же адресном пространстве

Аналогично **– сервер с предварительным созданием потоков.**

Неблокируемые сокеты/конечный автомат. Сервер работает в рамках одного потока, но использует неблокируемые сокеты и механизм поллинга. Т.е. сервер на каждой итерации бесконечного цикла выбирает из всех сокетов тот, что готов для приема/отправки данных с помощью вызова

select(). После того, как сокет выбран, сервер отправляет на него данные или читает их, но не ждет подтверждения, а переходит в начальное состояние и ждет события на другом сокете или же обрабатывает следующий, в котором событие произошло во время обработки предыдущего.

Данная модель очень эффективно использует процессор и память, но достаточно сложна в реализации. Кроме того, в рамках этой модели обработка события на сокете должна происходить очень быстро – иначе в очереди будет скапливаться много событий, и в конце концов она переполнится. Именно по такой модели работает nginx. Кроме того, он позволяет запускать несколько рабочих процессов (так называемых workers), т.е. может использовать несколько процессоров.

25. Обработка множественных запросов. Реализация сервера на основе конечного автомата

Неблокируемые сокеты/конечный автомат. Сервер работает в рамках одного потока, но использует неблокируемые сокеты и механизм поллинга. Т.е. сервер на каждой итерации бесконечного цикла выбирает из всех сокетов тот, что готов для приема/отправки данных с помощью вызова select(). После того, как сокет выбран, сервер отправляет на него данные или читает их, но не ждет подтверждения, а переходит в начальное состояние и ждет события на другом сокете или же обрабатывает следующий, в котором событие произошло во время обработки предыдущего.

Данная модель очень эффективно использует процессор и память, но достаточно сложна в реализации. Кроме того, в рамках этой модели обработка события на сокете должна происходить очень быстро – иначе в очереди будет скапливаться много событий, и в конце концов она переполнится. Именно по такой модели работает nginx. Кроме того, он позволяет запускать несколько рабочих процессов (так называемых workers), т.е. может использовать несколько процессоров.

26. Основные средства IPC ОС Unix (System V IPC, POSIX IPC)

3 механизма, появились в Unix System V и были описаны в System V Interface Definition (SVID) – **сообщения, разделяемая память и семафоры**.

Интерфейсы трех механизмов SVID IPC основаны на общих принципах. Для того, чтобы разные процессы могли получить доступ к одному объекту системы, они должны «договориться» об идентификации этого объекта. Роль идентификатора для всех объектов System V IPC выполняет ключ – число, уникальное в пределах подсистемы System V IPC. Для того, чтобы использовать один и тот же объект, программы должны использовать один и тот же ключ. Для каждого объекта IPC предусмотрены специальные функции чтения и записи, а также управляющая функция.

Сообщения накапливаются в очередях и могут изыматься из очередей последовательно или в произвольном порядке. Каждая группа процессов может создать одну или несколько очередей для обмена сообщениями. Сообщение определяется как «последовательность байтов, передаваемая от одного процесса другому». Посылающий процесс присваивает каждому сообщению тип. Получатель может избирательно читать сообщения из очереди, основываясь на этом типе. В частности, он может получить первое сообщение в очереди (независимо от типа), первое сообщение данного типа или первое сообщение любого типа из нескольких.

Функции System V IPC: msgrcv, msgsnd

Функции POSIX: mq_open(), mq_close(), mq_send(), mq_receive()

Разделяемая память – наиболее быстрый способ IPC. Разделяемая память может принадлежать более чем одному процессу. С момента присоединения, разделяемая память становится частью области данных процесса. Как только этот сегмент памяти модифицируется, новые данные становятся доступны всем процессам, присоединённым к нему. Разделяемая память может использоваться для хранения общей для нескольких процессов информации, такой как таблицы поиска или критерии правильности данных.

В POSIX это реализовано через технологию File Mapping - отображения файлов на оперативную память. Этот механизм позволяет осуществлять доступ к файлу таким образом, как будто это обыкновенный массив, хранящийся в памяти (не загружая файл в память явно). Отсюда появляется возможность работать с таким отображенным файлом сразу нескольким процессам. Таким образом, можно создать объект **file mapping**, но не ассоциировать его с каким-то конкретным файлом. Получаемая область памяти как раз и будет общей между процессами.

Функции System V IPC: shmget, shmctl
Функции POSIX: shm_open(), shm_unlink()

Семафоры используются для синхронизации процессов и управления ресурсами. Например, семафор может быть использован для управления доступом к устройству, такому как принтер. Семафор может гарантировать, что с принтером в данный момент работает только один процесс. Это защитит от перемешивания вывода нескольких процессов на печать.

Структуру POSIX семофоров определяет всего один семафор, в отличие от System V IPC, где структура определяет массивом до 25 эдементов.

Функции System V IPC: semget, semctl, semop
Функции POSIX: sem_open(), sem_close(), sem_getvalue() и др.

ipcs –q (очередь) –m (разделяемая память) –s (семафоры)

27. Основные средства IPC ОС MS Windows

Буфер обмена - одна из самых примитивных и хорошо известных форм IPC. Он появился еще в самых ранних версиях Windows. Основная его задача - обеспечивать обмен данными между программами по желанию и под контролем пользователя. Не рекомендуется использовать его для внутренних нужд приложения, и не стоит помещать туда то, что не предназначено для прямого просмотра пользователем.

Сообщение WM_COPYDATA

Стандартное сообщение для передачи участка памяти другому процессу. Работает однонаправленно, принимающий процесс должен расценивать полученные данные как read only. Посылать это сообщение необходимо только с помощью SendMessage, которая в отличие от PostMessage ждет завершения операции. Это не имеет значения для небольших кусков данных, но для больших объемов данных или для real-time приложений этот способ вряд ли подходит.

Разделяемая память (shared memory)

Этот способ взаимодействия реализуется не совсем напрямую, а через технологию File Mapping - отображения файлов на оперативную память. Вкраце, этот механизм позволяет осуществлять доступ к файлу таким образом, как будто это обыкновенный массив, хранящийся в памяти (не загружая файл в память явно). "Побочным эффектом" этой технологии является возможность работать с таким отображенным файлом сразу нескольким процессам. Таким образом, можно создать объект file mapping, но не ассоциировать его с каким-то конкретным файлом. Получаемая область памяти как раз и будет общей между процессами. Работая с этой памятью, потоки обязательно должны согласовывать свои действия с помощью объектов синхронизации.

Каналы (pipes)

Каналы - это очень мощная технология обмена данными. Чаще всего каналы используются для передачи непрерывного потока данных. Каналы делятся на анонимные (anonymous pipes) и именованные (named pipes). Анонимные каналы используются достаточно редко, они просто передают поток вывода одного процесса на поток ввода другого. Именованные каналы передают произвольные данные и могут работать через сеть. (Именованные каналы поддерживаются только в WinNT/2000.)

Почтовые слоты (mailslots)

Почтовые слоты - это механизм однонаправленного IPC. Если приложению известно имя слота, оно может помещать туда сообщения, а приложение-хозяин этого слота (приемник) может их оттуда извлекать и соответствующим образом обрабатывать. Основное преимущество этого способа - возможность передавать сообщения по локальной сети сразу нескольким компьютерам за одну операцию. Для этого приложения-приемники создают почтовые слоты с одним и тем же именем. Когда в дальнейшем какое-либо приложение помещает сообщение в этот слот, приложения-приемники получают его одновременно.

Объекты синхронизации

Microsoft Message Queue (MSMQ) обеспечивает посылку сообщений между приложениями с помощью очереди сообщений. Основное его отличие от стандартной очереди сообщений Windows в том, что он может работать с удаленными процессами и даже с процессами, которые на данный момент недоступны (например, не запущены). Доставка сообщения

по адресу гарантируется. Оно ставится в специальную очередь сообщений и находится там до тех пор, пока не появляется возможность его доставить.

28. Управление памятью. Виды памяти, модели представления памяти, методы распределения памяти

Программы вместе с данными, к которым они имеют доступ, в процессе выполнения должны (по крайней мере частично) находиться в оперативной памяти. ОС решает задачу распределения памяти между пользовательскими процессами и компонентами ОС. Эта деятельность называется **управлением памятью**. Таким образом, память (storage, memory) является важнейшим ресурсом, требующим тщательного управления. В недавнем прошлом память была самым дорогим ресурсом.

Часть ОС, которая отвечает за управление памятью, называется **менеджером памяти.**

Физическая организация памяти компьютера

ЗУ компьютера разделяют, как минимум, на два уровня: основную (главную, оперативную, физическую) и **вторичную** (внешнюю) память.

Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно основная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичная память (это главным образом диски) - также одномерное линейное адресное пространство, состоящее из последовательности байтов, однако, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

Для эффективного контроля использования памяти, ОС должна выполнять следующие **функции**:

1. отображение адресного пространства процесса на конкретные области физической памяти;
2. распределение памяти между конкурирующими процессами;

3. контроль доступа к адресным пространствам процессов;
4. выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;
5. учет свободной и занятой памяти.

Методы распределения памяти

- без использования внешней памяти: фиксированными разделами, динамическими разделами или перемещаемыми разделами
- с использованием внешней памяти: страничное распределение, сегментное распределение, сегментно-страничное распределение

Схема с фиксированными разделами

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько разделов фиксированной величины. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического адресного пространства. Связывание логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда – на этапе компиляции.

Оверлейная структура

Так как размер логического адресного пространства процесса может быть больше, чем размер выделенного ему раздела (или больше, чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay) или организация структуры с перекрытием. Основная идея – держать в памяти только те инструкции программы, которые нужны в данный момент.

Динамическое распределение. Свопинг

Имея дело с пакетными системами, можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping) – перемещению процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (paging).

Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом

связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. Очевидно, что свопинг увеличивает время переключения контекста. Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, то есть быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

Система свопинга может базироваться на **фиксированных** разделах, однако более эффективно **динамическое распределение** (вначале вся память свободна и не разделена заранее на разделы): First fit – Best fit – Worst fit стратегии

Страничная память

В самом простом и наиболее распространенном случае страничной организации памяти (или paging) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы (page), а соответствующие единицы в физической памяти называют физическими страницами или страничными кадрами (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое число страниц, ограничены частью последней страницы процесса.

Сегментная и сегментно-страничная организация памяти

Существуют две другие схемы организации управления памятью: сегментная и сегментно-страничная. Сегменты, в отличие от страниц, могут иметь переменный размер. При сегментной организации виртуальный адрес является двумерным как для программиста, так и для операционной системы, и состоит из двух полей – номера сегмента и смещения внутри сегмента.

29. Подсистема памяти. Планировщик (менеджер памяти). Уровни доступа к распределению памяти

Физическая организация памяти компьютера

ЗУ компьютера разделяют, как минимум, на два уровня: основную (главную, оперативную, физическую) и **вторичную** (внешнюю) память.

Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно основная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичная память (это главным образом диски) - также одномерное линейное адресное пространство, состоящее из последовательности байтов, однако, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

Менеджер памяти – часть компьютерной программы (как прикладной, так и операционной системы), обрабатывающая запросы на выделение и освобождение оперативной памяти или (для некоторых архитектур ЭВМ) запросы на включение заданной области памяти в адресное пространство процессора.

Основное назначение менеджера памяти в первом смысле – реализация динамической памяти.

Менеджеры памяти часто образуют иерархию: нижестоящие менеджеры задействуют какие-либо закономерности выделения-освобождения памяти и этим снижают нагрузку на вышестоящие.

Например:

1. **Системный**. Сверху находится менеджер памяти, встроенный в ОС. Он вносит ту или иную страницу в адресное пространство процесса — а значит, работает с дискретностью в 4 килобайта и очень медленный.
2. **Принадлежащий процессу.** Менеджер памяти, встроенный в стандартную библиотеку языка программирования, берёт у ОС блоки памяти «оптом» и раздаёт их сообразно с нуждами программиста. При этом он знает, что память отдаётся только одному процессу — а значит, синхронизация потоков производится не мютексами, а фьютексами. И переключение в

режим ядра происходит в двух случаях: либо когда «оперативного запаса» памяти не хватает и нужно обратиться к ОС, либо когда один из потоков «натыкается» на занятый фьютекс.

3. **Специализированные**. Некоторые динамические структуры данных, например, std::vector, также берут память у стандартной библиотеки с запасом (например, блоками по 16 элементов). Таким образом, элементы добавляются по одному, но обращение к вышестоящему менеджеру происходит один раз за 16 элементов. Объектный пул выделяет память под объекты конкретного типа и удобен, если они выделяются-освобождаются в больших количествах, и т. д.

30. Файловая система. Назначение, основные функции и требования, подходы к организации.

Назначение файловой системы

Цели и задачи файловой системы:

- Хранение информации: любое компьютерное приложение получает, хранит и выводит данные. Во время работы процесс может хранить ограниченное количество данных в собственном адресном пространстве, поскольку его емкость ограничена рамками виртуального адресного пространства.

- Сохранение информации при аварийном завершении процесса: после завершения работы процесса информация, хранящаяся в его адресном пространстве, теряется. Долговременность обеспечивается запоминающими устройствами, не зависящими от электропитания

- Доступ к данным: часто необходимо разным процессам одновременно получать доступ к одним и тем же данным (или части данных). Для решения этой проблемы необходимо отделить информацию от процесса.

Решение этих проблем состоит в хранении информации, организованной в файлы.

Файл – это именованная совокупность данных, хранящаяся на каком-либо носителе информации. При рассмотрении файлов и их совокупностей используются понятия:

Поле (Field) – основной элемент данных. Содержит единственное значение, такое как имя служащего, дату, значение некоторого показателя и т.п. Поле характеризуется длиной и типом данных и может быть фиксированной или переменной длины, т.е. состоять из нескольких подполей: имя поля, значение, длина поля.

Запись (Record) – набор связанных между собой полей, которые могут быть обработаны как единое целое некоторой прикладной программой (например, запись о сотруднике, содержащая такие поля, как имя, должность, оклад и т.д.). В зависимости от структуры записи могут быть фиксированной или переменной длины.

Обычно единственным способом работы с файлами является применение системы управления файлами или иначе – файловой системы (ФС).

Файловая система – это часть операционной системы, включающая:

- совокупность всех файлов на носителе информации (магнитном или оптическом диске, магнитной ленте и др.);
- наборы структур данных, используемых для управления файлами, каталоги и дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске и др.);
- комплекс системных программных средств, реализующих различные операции над файлами (создание, уничтожение, чтение, запись и др.).

Основные функции и требования файловой системы

Задачи, решаемые файловой системой, во многом определяются способом организации вычислительного процесса (наиболее простые – в однопрограммных и однопользовательских ОС, наиболее сложные – в сетевых ОС.).

В мультипрограммных, многопользовательских ОС **задачами (функциями)** файловой системы являются

- соответствие требованиям управления данными и требованиям со стороны пользователей, включающим возможность хранения данных и выполнения операций с ними;
- гарантирование корректности данных, содержащихся в файле;

оптимизация производительности, как с точки зрения системы (пропускная способность), так и с точки зрения пользователя (время отклика);

- поддержка ввода-вывода для различных типов устройств хранения информации;
- минимизация или полное исключение возможных потерь или повреждений данных;
- защита файлов от несанкционированного доступа;
- обеспечение поддержки совместного использования файлов несколькими пользователями (в т.ч. средства блокировки файла и его частей, исключение тупиков, согласование копий);
- обеспечение стандартизированного набора подпрограмм интерфейса ввода-вывода.

Минимальным набором **требований** к файлам системы **со стороны пользователя** диалоговой системы общего назначения можно считать следующую совокупность возможностей, предоставляемую пользователю:
- создание, удаление, чтение и изменение файлов;
- контролируемый доступ к файлам других пользователей;
- управление доступом к своим файлам;
- реструктурирование файлов в соответствии с решаемой задачей;
- перемещение данных между файлами;
- резервирование и восстановление файлов в случае повреждения;
- доступ к файлам по символьным именам.

Подходы к организации файловой системы

ФС распределяет дисковую память, поддерживает именование файлов, отображает имена файлов в соответствующие адреса во внешней памяти, обеспечивает доступ к данным, поддерживает разделение, защиту и восстановление данных.

ФС играет роль промежуточного слоя, экранизирующего все сложности физической организации долговременного хранилища данных и создающего для программ более простую логическую модель этого хранилища, а затем предоставляет им набор удобных в использовании команд для манипулирования файлами.

На нижнем уровне драйверы устройств непосредственно связаны с периферийными устройствами или их котроллерами либо каналами. Драйвер устройства отвечает за начальные операции ввода-вывода устройства и за обработку завершения запроса ввода-вывода. При файловых операциях контролируемыми устройствами являются дисководы

и стримеры (накопители на МЛ). Драйверы устройств рассматриваются как часть операционной системы.

Следующий уровень называется **базовой файловой системой**, или уровнем физического ввода-вывода. Это первичный интерфейс с окружением (периферией) компьютерной системы. Он оперирует блоками данных, которыми обменивается с дисками, магнитной лентой и другими устройствами. Поэтому он связан с размещением и буферизацией блоков в оперативной памяти. На этом уровне не выполняется работа с содержимым блоков данных или структурой файлов. Базовая файловая система обычно рассматривается как часть операционной системы (в MS-DOS эти функции выполняет BIOS, не относящийся к ОС).

Диспетчер базового ввода-вывода отвечает за начало и завершение файлового ввода-вывода. На этом уровне поддерживаются управляющие структуры, связанные с устройством ввода-вывода, планированием и статусом файлов. Диспетчер осуществляет выбор устройства, на котором будет выполняться операция файлового ввода-вывода, планирование обращения к устройству (дискам, лентам), назначение буферов ввода-вывода и распределение внешней памяти. Диспетчер базового ввода-вывода является частью ОС.

Логический ввод-вывод предоставляет приложениям и пользователям доступ к записям. Он обеспечивает возможности общего назначения по вводу-выводу записей и поддерживает информацию о файлах.

Наиболее близкий к пользователю уровень ФС часто называется **методом доступа**. Он обеспечивает стандартный интерфейс между приложениями и файловыми системами и устройствами, содержащими данные. Различные методы доступа отражают различные структуры файлов и различные пути доступа и обработки данных.

ПРОВЕРЯЮЩЕМУ: вопрос расписан для Unix. Изначально, когда сдавали ОСиС, он тоже был для юникс. Думаю, если проверять будет Сиротко, проблем не возникнет.

31. Объекты файловой системы, файлы, директории, логические устройства. Атрибуты файлов.

Объекты файловой системы

Типы объектов ФС (для Unix):

В UNIX существует шесть типов файлов, различающихся по строение и поведению при выполнении операций над ними:

- **Обычный файл** (regular file): наиболее общий тип файлов, содержащий данные в некотором формате. Для ОС это просто последовательность байт. Интерпретация содержимого производится прикладной задачей. Пример: текстовый файл, двоичные данные, исполняемый файл. Команды: cat имя и less имя.

- **Каталог** (directory): Это файл, содержащий имена находящихся в нем файлов, а также указатели на метаданные этих файлов, позволяющие ОС производить операции над ними.
Каталоги определяют положение файла в дереве файловой системы, так как сам файл не содержит информации о своем местонахождении. Каталоги образуют дерево. Первые два байта в каждой строке каталога являются единственной связью между именем файла и его содержимым. Именно поэтому *имя файла в каталоге называют связью*. Оно связывает имя в иерархии каталогов с индексным дескриптором и, тем самым, с информацией.

- **Специальный файл устройства** (special device file): Обеспечивает доступ к физическому устройству. Различают символьные и блочные файлы устройств. Доступ к устройствам происходит путем открытия, чтения/записи в специальный файл устройства. *Символьные файлы* позволяют небуферизованный обмен данными (посимвольно), а *блочные* - обмен пакетами определенной длины - блоками. К некоторым устройствам доступ возможен как через символьные, так и через блочные файлы. Для создания - mknod.

- **FIFO или именованный канал** (named pipe). Используется для связи между процессами.

- **Связь** (ссылка)

- **Жесткая ссылка:** Связь имени файла с его данными называется *жесткой ссылкой* (hard link). Имена жестко связаны с метаданными и, соответственно, с данными файла, в то время, как файл существует независимо от того, как его называют в файловой системе. Такая система позволяет одному файлу иметь несколько имен в файловой системе.

Пример: $ pwd /home/stud1 $ln first /home/stud2 second # создание жесткой ссылки.

Все жесткие ссылки на файл абсолютно равноправны. Файл существует в системе до тех пор, пока существует хотя бы одна жесткая связь, указывающая на него, то есть пока у него есть хотя бы одно имя.

- **Символическая ссылка:** Особый тип связи - символическая связь, позволяющая косвенно адресовать файл, в отличие от жесткой, обращающейся напрямую. Символическая ссылка содержит в себе имя файла, на который ссылается, а не его данные.

- **Сокет** (socket): Используются для межпроцессного взаимодействия.

Структура файловой системы

Все Unix-системы имеют сходную систему расположения и именования файлов и каталогов. Использование общепринятых имен файлов и структуры каталогов в UNIX-подобных ОС облегчает работу и перенос. Нарушение структуры ведет к нарушениям в работе.

Корневой каталог "/" является основой FS. Все остальные файлы и каталоги располагаются в рамках структуры, порождаемой корневым каталогом. *Абсолютное или полное имя* файла определяет точное местонахождение файла в структуре файловой системы. Начинается с "/" (в корневом каталоге) и содержит полный путь подкаталогов, которые нужно пройти, чтобы достичь файла.
Относительное имя определяет местонахождение файла через текущий каталог. Никогда не начинается с "/". *Каталог-предок* - это тот, который содержит другой каталог (..). Корневой каталог не имеет предка. Каталог, находящийся в другом каталоге, называется *каталогом-потомком или подкаталогом*. К текущему каталогу можно обратиться по имени ".": ./file1. *Домашним или начальным каталогом* называется область, котрая выделяется каждому пользователю и в которой он может хранить свои файлы и программы “~”: ~/file.txt.

Основные каталоги

/bin - наиболее часто употребляемые файлы и утилиты.
/dev - содержит специальные файлы устройств, являющиеся интерфейсом доступа к периферийным устройствам. Может содержать подкаталоги, группирующие устройства по типам. Например, /dev/dsk - доступ к дискам.
/etc - системные конфигурационные файлы и утилиты. Иногда утилиты отсюда выносятся в /sbin и /usr/sbin.
/lib - библиотеки Си и других языков программирования. Часть библиотек - в /usr/lib.

/usr, /usr/bin - утилиты;/usr/include - заголовочные файлы Си; /usr/man - справочная система; /usr/local - дополнительные программы; /usr/share - файлы, разделяемые между различными программами.
/var - временные файлы сервисных подсистем (печати, почты, новостей).
/tmp - каталог временных файлов. Обычно открыт на запись для всех пользователей системы.

Атрибуты файлов

Группой называется определенный список пользователей системы. Пользователь может быть членом нескольких групп, одна из которых является первичной, а остальные - дополнительными. В UNIX любой файл имеет двух владельцев: владельца-пользователя и владельца-группу. При этом владелец-пользователь не обязательно принадлежит владельцу-группе.
Сменить владельца-пользователя может либо текущий владелец, либо администратор (root). Сменить владельца-группу может либо владелец-пользователь для группу, к которой он сам принадлежит (POSIX), либо администратор.

Права доступа к файлам

У каждого файла существуют атрибуты, называемые правами доступа. В UNIX существует три базовых типа доступа:
u (user), g (group), o (other), a (all).
В каждом из этих классов установлены три основных права доступа:
r (read), w (write), x (execute)
Права может изменять владелец-пользователь и(или) администратор. Для изменения прав доступа используется команда chmod:

Последовательность проверки прав

-если вы администратор (root), доступ разрешен. Права не проверяются.
-если операция запрашивается владельцем, идет проверка его прав. В соответствии с ними ему разрешается выполнение операции или нет.
-если операция запрашивается пользователем, входящим в группу, владеющую файлом, идет проверка его прав. Соответственно, он либо получает разрешение, либо нет.
аналогично для всех остальных пользователей.

Дополнительные атрибуты файла (также устанавливаются с помощью chmod)
Для обычных файлов:

t - *"sticky bit"* (бит липучка)- сохранить образ выполняемого файла в памяти после выполнения (устаревший аттрибут)
s - set UID, *SUID* - установить права у процесса, как у запущенного файла, а не как у пользователя, запустившего программу (по умолчанию)
s - set GID, *SGID* - то же для группы
1 - блокирование - в каждый момент времени с файлом может работать только одна задача
Для каталогов:t - пользователь может удалять только те файлы и каталоги, которыми владеет или имеет право на запись;

32. Защищаемые объекты, угрозы, средства обеспечения безопасности в операционных системах.

Объекты, нуждающиеся в защите: оборудование (например, центральные процессоры, сегменты памяти, дисковые приводы или принтеры) или программное обеспечение (например, процессы, файлы, базы данных или семафоры).

Угрозы безопасности ОС можно классифицировать по различным аспектам их реализации:

1. По цели атаки:

• несанкционированное чтение информации;
• несанкционированное изменение информации;
• несанкционированное уничтожение информации;
• полное или частичное разрушение ОС.

2. По принципу воздействия на операционную систему.

• использование известных (легальных) каналов получения информации; например угроза несанкционированного чтения файла, доступ пользователей к которому определен некорректно, т. е. разрешен доступ пользователю, которому согласно политике безопасности доступ должен быть запрещен;

• использование скрытых каналов получения информации; например угроза использования злоумышленником недокументированных возможностей ОС;

• создание новых каналов получения информации с помощью программных закладок.

3. По типу используемой злоумышленником уязвимости защиты:

• неадекватная политика безопасности, в том числе и ошибки администратора системы;

• ошибки и недокументированные возможности программного обеспечения ОС, в том числе и так называемые люки — случайно или преднамеренно встроенные в систему «служебные входы», позволяющие обходить систему защиты;

• ранее внедренная программная закладка.

4. По характеру воздействия на операционную систему:

• активное воздействие — несанкционированные действия злоумышленника в системе;

• пассивное воздействие — несанкционированное наблюдение злоумышленника за процессами, происходящими в системе.

Угрозы безопасности ОС можно также классифицировать по таким признакам, как: способ действий злоумышленника, используемые средства атаки, объект атаки, способ воздействия на объект атаки, состояние атакуемого объекта ОС на момент атаки. ОС может подвергнуться следующим **типичным атакам:**

• сканированию файловой системы. Злоумышленник просматривает файловую систему компьютера и пытается прочесть (или скопировать) все файлы подряд. Рано или поздно обнаруживается хотя бы одна ошибка администратора. В результате злоумышленник получает доступ к информации, который должен быть ему запрещен;

• подбору пароля. Существуют несколько методов подбора паролей пользователей:

— тотальный перебор;

— тотальный перебор, оптимизированный по статистике встречаемости символов или с помощью словарей;

— подбор пароля с использованием знаний о пользователе (его имени, фамилии, даты рождения, номера телефона и т. д.);

• краже ключевой информации. Злоумышленник может подсмотреть пароль, набираемый пользователем, или восстановить набираемый пользователем пароль по движениям его рук на клавиатуре. Носитель с ключевой информацией (смарт-карта, Touch Memory и т. д.) может быть просто украден;

• сборке мусора. Во многих ОС информация, уничтоженная пользователем, не уничтожается физически, а помечается как уничтоженная (так называемый мусор). Злоумышленник восстанавливает эту информацию, просматривает ее и копирует интересующие его фрагменты;

• превышению полномочий. Злоумышленник, используя ошибки в программном обеспечении ОС или политике безопасности, получает полномочия, превышающие те, которые ему предоставлены в соответствии с политикой безопасности. Обычно это достигается путем запуска программы от имени другого пользователя;

• программным закладкам. Программные закладки, внедряемые в ОС, не имеют существенных отличий от других классов программных закладок;

• жадным программам — это программы, преднамеренно захватывающие значительную часть ресурсов компьютера, в результате чего другие программы не могут выполняться или выполняются крайне медленно. Запуск жадной программы может привести к краху ОС.

С точки зрения безопасности у компьютерных систем есть четыре основные задачи, соответствующие угрозам:

- **конфиденциальность** данных — сохранение секретности соответствующих данных. Система должна гарантировать невозможность допуска к данным лиц, не имеющих на это права. Как минимум, владелец должен иметь возможность определить, кто и что может просматривать, а система должна обеспечить выполнение этих требований, касающихся в идеале отдельных файлов.

- **целостность** данных (data integrity) - пользователи, не обладающие соответствующими правами, не должны иметь возможности изменять какие-либо данные без разрешения их владельцев (внесение в них изменений, удаление или добавление ложных данных). Если система не может гарантировать, что заложенные в нее данные не будут подвергаться изменениям, пока владелец не решит их изменить, то она потеряет свою роль информационной системы.

- **работоспособность** системы (system availability) —никто не может нарушить работу системы и вывести ее из строя. Атаки, вызывающие отказ от обслуживания (denial of service, DOS), приобретают все более распространенный характер. Например, если компьютер работает в роли интернет-сервера, то постоянное забрасывание его запросами может лишить его работоспособности, отвлекая все рабочее время его центрального процессора на изучение и отклонение входящих запросов.

- **исключение постороннего доступа** - посторонние лица могут иногда взять на себя управление чьими-нибудь домашними компьютерами (используя вирусы и другие средства) и превратить их в зомби (zombies), моментально выполняющих приказания посторонних лиц. Часто зомби

используются для рассылки спама, поэтому истинный инициатор спам-атаки не может быть отслежен.

Средства обеспечения безопасности в ОС

Большинство ОС позволяют отдельным пользователям определять, кто может осуществлять операции чтения и записи в отношении их файлов и других объектов - политика **разграничительного управления доступом** (discretional access controlI). Для более жестких мер безопасности (военные организации, корпоративные патентные отделы, лечебные учреждения) – **принудительное управление доступом** (mandatory access control).

Понятие защищенной ОС

Операционную систему называют **защищенной**, если она предусматривает средства защиты от основных классов угроз. Защищенная ОС обязательно должна содержать средства разграничения доступа пользователей к своим ресурсам, а также средства проверки подлинности пользователя, начинающего работу с ОС. Кроме того, защищенная ОС должна содержать средства противодействия случайному или преднамеренному выводу ОС из строя.

Если ОС предусматривает защиту не от всех основных классов угроз, а только от некоторых, такую ОС называют **частично защищенной.**

Подходы к построению защищенных ОС

Существуют два основных подхода к созданию защищенных ОС — **фрагментарный** и **комплексный**. При **фрагментарном** подходе вначале организуется защита от одной угрозы, затем от другой и т. д. Примером фрагментарного подхода может служить ситуация, когда за основу берется незащищенная ОС, на нее устанавливаются антивирусный пакет, система шифрования, система регистрации действий пользователей и т. д. При применении фрагментарного подхода подсистема защиты ОС представляет собой набор разрозненных программных продуктов, как правило, от разных производителей. Эти программные средства работают независимо друг от друга, при этом практически невозможно организовать их тесное взаимодействие. Кроме того, отдельные элементы такой подсистемы защиты могут некорректно работать в присутствии друг друга, что приводит к резкому снижению надежности системы.

При **комплексном** подходе защитные функции вносятся в ОС на этапе проектирования архитектуры ОС и являются ее неотъемлемой частью. Отдельные элементы подсистемы защиты, созданной на основе комплексного подхода, тесно взаимодействуют друг с другом при решении различных задач, связанных с организацией защиты информации, поэтому

конфликты между ее отдельными компонентами практически невозможны. Подсистема защиты, созданная на основе комплексного подхода, может быть устроена так, что при фатальных сбоях в функционировании ее ключевых элементов она вызывает крах ОС, что не позволяет злоумышленнику отключать защитные функции системы. При фрагментарном подходе такая организация подсистемы защиты невозможна.

Административные меры защиты

1. Постоянный контроль корректности функционирования ОС, особенно ее подсистемы защиты. Такой контроль удобно организовать, если ОС поддерживает автоматическую регистрацию наиболее важных событий (event logging) в специальном журнале.

2. Организация и поддержание адекватной политики безопасности. Политики безопасности ОС должна постоянно корректироваться, оперативно реагируя на попытки злоумышленников преодолеть защиту ОС, а также на изменения в конфигурации ОС, установку и удаление прикладных программ.

3. Инструктирование пользователей операционной системы о необходимости соблюдения мер безопасности при работе с ОС и контроль за соблюдением этих мер.

4. Регулярное создание и обновление резервных копий программ и данных ОС.

5. Постоянный контроль изменений в конфигурационных данных и политике безопасности ОС. Информацию об этих изменениях целесообразно хранить на неэлектронных носителях информации, для того чтобы злоумышленнику, преодолевшему защиту ОС, было труднее замаскировать свои несанкционированные действия.

Адекватная политика безопасности

Оптимальная адекватная политика безопасности — это такая политика безопасности, которая не только не позволяет злоумышленникам выполнять несанкционированные действия, но и не приводит к описанным выше негативным эффектам.

Адекватная политика безопасности определяется не только архитектурой ОС, но и ее конфигурацией, установленными прикладными программами и т. д. Формирование и поддержание адекватной политики безопасности ОС можно разделить на ряд этапов.

1. **Анализ угроз**. Администратор ОС рассматривает возможные угрозы безопасности данного экземпляра ОС. Среди возможных угроз

выделяются наиболее опасные, защите от которых нужно уделять максимум средств.

2. **Формирование требований к политике безопасности.** Администратор определяет, какие средства и методы будут применяться для защиты от тех или иных угроз. Например, защиту от НСД к некоторому объекту ОС можно решать либо средствами разграничения доступа, либо криптографическими средствами, либо используя некоторую комбинацию этих средств.

3. **Формальное определение политики безопасности.** Администратор определяет, как конкретно должны выполняться требования, сформулированные на предыдущем этапе. Формулируются необходимые требования к конфигурации ОС, а также требования к конфигурации дополнительных пакетов защиты, если установка таких пакетов необходима. Результатом данного этапа является развернутый перечень настроек конфигурации ОС и дополнительных пакетов защиты с указанием того, в каких ситуациях, какие настройки должны быть установлены.

4. **Претворение в жизнь политики безопасности**. Задачей данного этапа является приведение конфигурации ОС и дополнительных пакетов защиты в соответствие с политикой безопасности, формально определенной на предыдущем этапе.

5. **Поддержание и коррекция политики безопасности**. В задачу администратора на данном этапе входит контроль соблюдения политики безопасности и внесение в нее необходимых изменений по мере появления изменений в функционировании ОС.

33. Особенности управляющих систем и систем реального времени.

Под **реальным временем (РВ)** понимается количественная характеристика, которая может быть измерена реальными физическими часами, в отличие от ***логического времени***, определяющего лишь качественную характеристику, выражаемую относительным порядком следования событий. Говорят, что система работает в ***режиме реального времени***, если для описания работы этой системы требуются количественные временные характеристики.

Система реального времени (СРВ) — это система, которая должна реагировать на события во внешней по отношению к системе среде или воздействовать на среду в рамках требуемых временных ограничений. Другое определение: **СРВ** – система, в которой успешность работы любой программы зависит не только от ее логической правильности, но и от времени, за которое она получила р-т. Если временные ограничения не удовлетворены, то фиксируется сбой в работе систем. Т.о., временные ограничения должны быть гарантированно удовлетворены. Это требует от системы быть предсказуемой, то есть вне зависимости от своего текущего состояния и загруженности выдавать нужный р-т за требуемое время. При этом желательно, чтобы система обеспечивала как можно больший % использования имеющихся ресурсов.

Примером задачи, где требуется СРВ, является управление роботом, берущим деталь с ленты конвейера. Деталь движется, и робот имеет лишь небольшое временное окно, когда он может ее взять.

Стандарт POSIX: «РВ в ОС - это способность ОС обеспечить требуемый уровень сервиса в заданный промежуток времени».

Требования:

- своевременная реакция. После того как произошло событие, реакция должна последовать не позднее, чем через требуемое время. Превышение этого времени рассматривается как серьезная ошибка.
- одновременная обработка информации, которая характеризует изме¬нение процесса нескольких событий. Даже если одновременно происходит несколько событий, реакция ни на одно из них не должна запаздывать. Это означает, что система реального времени должна иметь встроенный параллелизм. Параллелизм достигается использованием нескольких процессоров в системе и/или многозадачным подходом.

Характеристики

Системы реального времени могут иметь следующие **характеристики** и связанные с ними **ограничения**:

- дедлайн (англ. deadline) — критический срок обслуживания, предельный срок завершения какой-либо работы;
- латентность (англ. latency) — время отклика (время задержки) системы на внешние события;
- джиттер (англ. jitter) — разброс значений времени отклика.

В зависимости от допустимых нарушений временных ограничений системы реального времени можно поделить на системы жёсткого реального времени, для которых нарушения равнозначны отказу системы, и системы мягкого реального времени, нарушения характеристик которых приводят лишь к снижению качества работы системы

Признаки систем жесткого реального времени:
. недопустимость никаких задержек, ни при каких условиях;
. бесполезность результатов при опоздании;
. катастрофа при задержке реакции;
. цена опоздания бесконечно велика.
Пример: системы жесткого реального времени - бортовая система управления самолетом.

Признаки систем мягкого реального времени:
. за опоздание результатов приходится платить;
. снижение производительности системы, вызванное запаздыванием реакции на происходящие события.

События реального времени могут относится к одной из трёх категорий:
- **Асинхронные** события — полностью непредсказуемые события. Например, вызов абонента телефонной станции.
- **Синхронные** события — предсказуемые события, случающиеся с определённой регулярностью. Например, вывод аудио и видео.
- **Изохронные** события — регулярные события (разновидность асинхронных), случающиеся в течение интервала времени. Например, в мультимедийном приложении данные аудиопотока должны прийти за время прихода соответствующей части потока видео.

Применение СРВ
СРВ применяются в промышленности, включая системы управления технологическими процессами, системы промышленной автоматики, испытательное и измерительное оборудование, робототехнику. Применения в медицине включают в себя томографию, оборудование для радиотерапии, прикроватный мониторинг. СРВ встроены в периферийные устройства компьютеры, телекоммуникационное оборудование и бытовую технику, такую как лазерные принтеры, сканеры, цифровые камеры, кабельные

модемы, маршрутизаторы, системы для видеоконференций и интернет-телефонии, мобильные телефоны, микроволновые печи, музыкальные центры, кондиционеры, системы безопасности. На транспорте СРВ применяются в бортовых компьютерах, системах регулирования уличного движения, управлении воздушного движения, аэрокосмической технике, системе бронирования билетов и т. п. СРВ находят применения и в военной технике: системах наведения ракет, противоракетных системах, системах спутникового слежения.