**Лекция №1 02.09.2016**

**Обзор операционных систем**

**ОС** – это ПО, которое, с одной стороны, обеспечивает удобный интерфейс взаимодействия пользователей с аппаратным обеспечением компьютера, а с другой стороны – управляет этим аппаратным обеспечением и вычислительными процессами, распределяя между ними ресурсы.

Основные функции ОС:

1. Исполнение программ;
2. Управление памятью;
3. Управление устройствами ввода/вывода;
4. Управление файлами;
5. Обеспечение пользователя интерфейсом.

Как правило, каждая из приведенных функции реализована в виде отдельной подсистемы, являющейся компонентом ОС.

В различных ОС эти функции реализуют по-своему и в различном объеме.

**Исполнение программ**

Исполнение программ заключается в выполнении различных команд на центральном процессоре. Программа в момент выполнения – процесс. У каждого процесса своё адресное пространство.

Адресное пространство – диапазон адресов в памяти, который процесс может использовать. Оно содержит саму программу и данные к ней;

В большинстве ОС процессам относят определенное время для работы с центральным процессором. Когда процесс израсходует отведенное ему время, ОС приостанавливает данный процесс и запускает другой, но позже приостановленный процесс будет запущен заново из того же состояния, в каком был остановлен.

**Управление памятью**

Под управлением памятью имеется в виду управление оперативной памяти, используемой для хранения выполняющихся программ.

В простых ОС в один момент времени в памяти может находиться только одна программа, а для запуска новой нужно удалить из записи предыдущую.

Более сложные ОС позволяют одновременно находится в памяти сразу нескольким программам.

Под управлением памятью также подразумевается управление адресным пространством процессов. В простейшем случае максимальная величина адресного пространства процесса ограничено размером оперативной памяти.

Если адресное пространство больше размера оперативной памяти, то процесс не будет выполнен.

Виртуальная память- при использовании которой она хранит часть адресного пространства процесса в оперативной памяти, а часть на жестком диске и меняет их местами при необходимости. При использовании виртуальной памяти адресное пространство может быть больше оперативной памяти.

**Управление устройствами ввода/вывода**

Различные устройства компьютера управляются ОС, для этого у каждого устройства имеет контроллер – управляющее устройство, которое принимает различные программы и выполняет их. Программа, которая дает команды контроллеру – драйвер. С помощью драйверов ОС может управлять различными устройствами.

Обычно вместе с ОС поставляются драйверы для всех устройств, без которых система не может работать.

**Управление файлами**

Файл – поименованная совокупность данных, размещенная на каком-либо запоминающемся устройстве и обрабатываемое, как единое целое.  
При работе с файлами ОС предоставляет пользователю некую абстрактную модель – файловую систему. Файловая система скрывает от пользователей очень сложную структуру физического хранения файла на дисках и других устройствах, представляя более удобную, логическую структуру в виде дерева каталогов, содержащих дерева файлов.

**Обеспечение пользовательского интерфейса**

Два подхода к организации пользовательского интерфейса:

1. текстовый интерфейс (не требует больших ресурсов и обладает большой скоростью отображения информации, использует исключительно набор буквенно-цифровых символов и псевдографики; разновидностью текстового интерфейса является интерфейс командной строки, в которой взаимодействие осуществляется путем ввода команд);
2. графический (имеет более функциональные элементы интерфейса, удобен для непрофессиональных пользователей, графический интерфейс строится на технологии WIMP (окно, пиктограмма, меню, всплывающие окна), мышь является устройством для манипуляции элементами интерфейса).

**Обеспечение безопасности**

Современная ОС содержит различные механизмы защиты, обеспечивающие безопасность ресурсов и объектов системы. Эти механизмы можно разделить:

1. идентификация (номер, имя) и аутентификация (личность) (ни один пользователь не может начать работу, не инфицировав себя и не предоставив аутентифицирующую информацию, которая подтверждает, что пользователь действительно является тем, кем за себя выдает);
2. разграничение доступа (каждый пользователь имеет доступ только к тем ресурсам и объектам ОС, к которым ему предоставлен доступ в соответствии с заданной политикой безопасности);
3. криптографические средства защиты (шифрование используется при хранении и передачи паролей и других данных критичных для безопасности системы).

**Классификация ОС:**

Единой не существует, но их можно разделить по определенным критериям:

1. **по типу аппаратного обеспечения:**

* ОС мейнфреймнов и суперкомпьютеров (мейнфреймны и суперкомпьютеры отличаются от обычных компьютеров своими возможностями; они предназначены для сверхбольших объемов данных, а для их управления используются специальные ОС, которые ориентированы на выполнение множества программ, большинству из которых требуется большое количество операции ввода/вывода;
* серверные ОС (могут не иметь графического отображения и устанавливаются на компьютеры, предоставляющие различные сетевые ресурсы, кроме управления занимается различными сетевыми приложениями:
* системами управления БД;
* средства обмена сообщениями и групповой работой;
* веб-серверы;
* серверы-приложений и другое.
* ОС персональных компьютеров (обычно на них работают «рядовые» пользователи, которые слабо представляют, что такое компьютер, поэтому основная задача – удобный и понятный интерфейс пользователя; они используются также для ноутбуков, планшетов и нетбуков);
* встраиваемые ОС (предназначены для управления таких устройств, как банкоматы, ТВ, телефоны, игровые консоли, платежные терминалы и.т.д.).

1. **по числу выполняемых задач:**

* однозадачные (в один момент времени может выполняться только одна программа);
* многозадачные (в один момент времени может выполняться несколько программ).
* ОС пакетной обработки (поочередно обрабатываются пакеты заданий, состоящие из различных программ, подлежащих выполнению; предназначены для решения задач, не требующих быстрого результата);
* ОС с разделением времени (каждой программе, подлежащей выполнению, дан небольшой интервал времени, если выполнение программы не закончилось к концу выполнения интервала, то она останавливается и помещается в очередь выполнения, затем начинается другая программа; очередность и продолжительность определяются специальными алгоритмами планирования);
* ОС реального времени (так же как и в ОС с разделением времени очередность выполнения программ определяется специальными алгоритмами планирования, однако каждая программа должна заканчиваться в фиксированное время; такие ОС применяются для управления целым технологическим процессом или техническим объектом).

1. **по числу одновременно работающих пользователей:**

* однопользовательские;
* многопользовательские.

Их различие в наличие механизмов защиты данных от несанкционированного доступа.

1. **По количеству поддерживаемых процессоров:**

* однопроцессорные;
* многопроцессорные.
* симметричные системы, в которых программа может быть выполнена на любом имеющимся процессоре;
* ассиметричные системы, в которых каждому процессору дается своё специфическое задание (например, одни процессоры – арифметически-логические операции, а другие – операции ввода/вывода и.т.д.).

**Лекция №2 16.09.2016**

**Архитектура ОС**

Под архитектурой ОС понимают её структурную и функциональную организацию на основе некоторых совокупностей программных модулей. Эти программные модули можно разделить на два типа:

1. ядро (это модуль, выполняющий основные функции операционной системы);
2. исполняемые службы (это модули, выполняющие вспомогательные функции ОС).

При рассмотрении архитектур ОС также используют два понятия:

* режим ядра (привилегированный защищенный режим, работа центрального процессора, при котором программе предоставляется прямой доступ ко всему аппаратному обеспечению компьютеру и разрешается выполнять привилегированные команды центрального процессора; в режиме ядра выполняются такие программы, как драйве устройств и менеджер виртуальной памяти);
* режим пользователя (непривилегированный режим работы центрального процессора, при котором программе запрещается выполнять привилегированные команды, а доступ к аппаратному обеспечению осуществляется только посредством взаимодействия программами, работающими в режиме ядра; в режиме пользователя выполняются прикладные программы; режим пользователя призван защищать ОС от сбоев в работе прикладных программ).

Прикладные программы взаимодействуют с ОС с помощью набора системных вызовов, предоставляемых ОС. Различные ОС обладают различным набором системных вызовов.

**Монолитные операционные системы**

Монолитная ОС представляет собой одну большую программу, которая выполняется в привилегированном режиме ядра. Компоненты такой ОС являются несамостоятельными модулями, а частями одной программы. Для монолитной ОС ядро совпадает со всей системой. В монолитной ОС сборка (компиляция ядра) осуществляется для каждого компьютера, на котором устанавливается ОС. При этом можно выбрать список компонентов, которые будут включены в ядро, так как ядро является единой программой, то единственный способ добавить в него новые компоненты или исключить неиспользованные – это пересобрать ядро заново. Отметим, что присутствие в ядре лишних компонентов крайне нежелательно, так как ядро всегда полностью помещается в оперативную память.

**Микроядерные операционные системы**

В микроядерных ОС ядро более компактное за счет того, что часть функций ядра делегируются различным исполняемым службам, которые работают в режиме пользователя. Примерами исполняемых служб в микроядерной ОС являются: сетевые службы, службы файловой системы, драйверы различных устройств и тому подобное. Взаимодействие между исполняемыми службами обеспечивают микроядро. Кроме этого, микроядро выполняет функции взаимодействия между процессами, планирование задач, управления операциями ввода/вывода и управление памятью.   
В микроядерной ОС добавление новых компонентов или исключение неиспользуемых возможно не прерывая работу системы. Кроме того, такие ОС являются более надежными, поскольку ошибка в режиме пользователя менее опасна, чем ошибка в режиме ядра. В некоторых случаях использование архитектуры микроядерных ОС приводит к некоторому замедлению выполнения системных вызовов по сравнению с монолитными ОС.

**Гибридные операционные системы**

В большинстве случае ОС используют различные комбинации архитектур монолитных и микроядерных ОС. Например, можно перенести в режим ядра драйверы устройств, что позволит им работать быстрее, обращаясь напрямую к устройствам. При этом, драйверы устройств остаются самостоятельными модулями, как в микроядерных ОС, но выполняются в режиме ядра, как в монолитный ОС. Отметим, что это не единственный пример архитектуры гибридной ОС.

**Современные операционные системы**

Развитие информационных технологий привело к появлению огромному количеству различных ОС.

**Linux** - это семейство ОС основано на одноименном ядре. В настоящее время разработка ядра продолжается различными компаниями и разработчиками- энтузиастами. Так как ядро Linux распространяется, как свободное и открытое ПО, большинство ОС, использующих это ядро, как правило, тоже являются свободными (то есть, доступны по свободным лицензиям).

**Debian** – является свободной ОС с отрытым программным кодом. Как правило, эта ОС используется для ПК или ноутбуков. Но может также использоваться в качестве серверной ОС и ОС для различных устройств.

**Ubuntu** – основана на Debian, но отличается тем, что в её состав включены не только проверенные временем компоненты, но и новые менее стабильные компоненты. Отметим, что эта ОС также является свободной с открытым программным кодом. По утверждению разработчиков этой ОС она является самой популярной ОС среди ОС на ядре Linux. Используется на ноутбуках, ТВ и телефонах.

**Fedora** – является свободной ОС с открытым программным кодом. Может использоваться на ПК и ноутбуках. Разработкой ОС Fedora занимаются разработчики, которых спонсирует компания Red Hat. Все новые технологии и компоненты, которые в дальнейшем включают программных продукт Red Hat сначала тестируются в ОС Fedora.

**Red Hat Enterprise Linux** – является коммерческой и предназначена для корпоративного использования.

**Mandriva Linux** – разрабатываются несколько версий: 1) free (бесплатная, свободный доступ, но в ней отсутствуют коммерческие компоненты); 2) One (пробная версия, предоставляемая в свободном доступе); 3) powerpack (полная коммерческая версия).

**openSUSE** – используется для ПК и является свободной ОС с открытым программным кодом.

**Google Android** – предназначена для мобильных устройств (семейство Linux).

**Oracle Solaris** – предназначена для корпоративных решений, является одной из наиболее популярных платформ.

**Apple OS** – OS X – современная версия, созданная компанией Apple для своих компьютеров Макентош компаний MacOS.

**Apple IOS** - операционная система для смартфонов, электронных планшетов и носимых проигрывателей, разрабатываемая и выпускаемая американской компанией Apple

**Microsoft Windows** – семейство ОС, разработанных компанией Microsoft. Первые версии Windows не были полноценными ОС, а являлись лишь настройками к ОС MS DOS. На сегодняшний день является самыми распространенными ОС.

**IBM AIX** – разработана компанией IBM для серверов серии PowerSystem.

**30.09.2016**

**Процессы и потоки**

Во всех многозадачных ОС Центральный процессор переключается между выполняемыми программами, предоставляя каждой из них своё время. При этом, процессор в каждый момент времени занят только одной программой, но создается иллюзия параллельной работы всех выполняемых программ.

**Процессы**

Все многозадачные ОС используют модель процессов. Согласно этой модели, все выполняемые программы организованы в виде процесса.

Процесс – это абстрактное понятие, описывающее работу программы.

Для реализации модели процессов ОС содержит специальную таблицу процессов. Каждый элемент такой таблицы содержит следующую информацию:

* идентификатор процесса (PID);
* идентификатор пользователя (от имени которого запущен процесс);
* значение сегментных регистров процессоров:
* сегмент данных;
* сегмент кода;
* расширенный сегмент;
* сегмент стека.
* значения регистров данных процессора:
* аккумулятор;
* счетчик;
* регистр данных;
* базовы й регистр.
* значения счетчика команд (IP);
* значение указателя стека (SP);
* значение базового указателя (BP);
* состояние процесса;
* использованное процессорное время;
* приоритет процесса.

Кроме этого, таблица процессов может также содержать дополнительную информацию, необходимая для управления процессом.

**Создание процесса**

При загрузке ОС создаются несколько системных процессов, кроме того, уже созданные процессы могут создавать новые процессы. Для этого созданный процесс выполняет специальный системный вызов, который заставляет ОС создать новый процесс.

В различных ОС для создания новых процессов используются различные подходы.

ОС Linux, Oracle Solaris, OSX, IOS, IBM AIX (все кроме Windows) реализуют подход, предложенный в стандартных осях.

POSIX – описывает интерфейс взаимодействия между ОС и прикладной программой. В POSIX создание нового процесса выполняется с помощью вызова функции: fork (создает копию вызывающего процесса); в дальнейшем созданный процесс может продолжить выполнять ту же программу или вызвать функцию exec (для загрузки и выполнения новой программы).

В Microsoft Windows также может использоваться подход POSIX, но преимущества отдаются другому подходу, который подразумевает вызов всего одной функции CreateProcess, которая создает процесс и запускает в нем нужную программу. При этом для создания процесса создается, а не копируется новое адресное пространство, в которое загружается программа.

**Завершение процесса**

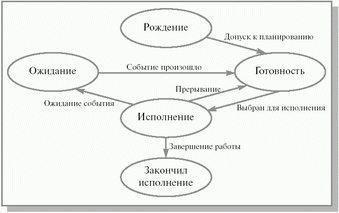
После того, как процесс был завершен, он больше не может выполняться.

Процесс завершается в одном из следующих случаях:

1. Программа завершила свою работу и выполнила специальный системный вызов, который завершает процесс (POSIX: exit или же Windows: ExitProcess);
2. Произошла неустранимая ошибка, в результате которой программа более не может продолжат свою работу;
3. Другой процесс выполнил системный вызов, в результате которого был принудительно завершен указанный процесс (POSIX: kill или же для Windows: TerminateProcess).

При завершении процесса удаляется созданное для него адресное пространство. Однако, ОС не сразу удаляет из таблицы процессов запись о завершенном процессе. Благодаря этому, какое-то время ещё можно узнать необходимую информацию о завершенном процессе. Например, код завершения процесса, сколько процессорного времени было использовано и тому подобное.

**Состояние процесса**

, где

* рождение – процесс создан, но ещё не загружен в основную память;
* готовность – процесс приостановлен, но может быть запущен, как только для этого представится возможность;
* исполнение – процесс выполняется в текущий момент;
* ожидание – процесс приостановлен до тех пор, пока не произойдет определенное событие;
* завершение – процесс уже завершен, но ещё не удален из таблицы процессов.

Возможны различные переходы:

* переход **рождение -> готовность**: происходит, когда ОС готова к обработке созданного процесса;
* переход **готовность -> исполнение**: выполняется компонентов ОС – планировщиком, переход происходит, когда планировщик решил снова предоставить процессор процессу;
* переход **исполнение -> готовность**: происходит, если планировщик решил, что пора приостановить процесс и предоставить процессор другому процессу;
* переход **исполнение -> ожидание**: происходит, когда продолжение работы процесса невозможно, пока не произойдет определенное событие, например, ввод с клавиатуры;
* переход **ожидание -> готовность**: происходит с наступлением того события, которое ожидал процесс;
* переход **исполнение -> завершение**: происходит при достижении точки завершения процесса, а также когда он завершается из-за возникновения неустранимой ошибки, или процесс завершается принудительно другим процессом.

**Потоки**

При создании процесса для создания адресного пространства требуются значительные системные ресурсы. Кроме того, в таблице процессов хранится масса дополнительной информации, на что уходит не мало памяти. Поэтому многозадачные ОС вместе с процессами используют потоки.

Поток исполняет свой вход и манипулирует данными в адресном пространстве процесса, и тем самым, расходует существенно меньше ресурсов, чем процесс.

**Таблица потоков**

Подобно процессам, сведения о потоках хранятся в специальных таблицах потока:

* идентификатор потока;
* значение сегментных регистров процессоров:
* сегмент данных;
* сегмент кода;
* расширенный сегмент;
* сегмент стека.
* значения регистров данных процессора:
* аккумулятор;
* счетчик;
* регистр данных;
* базовы й регистр.
* значения счетчика команд (IP);
* значение указателя стека (SP);
* значение базового указателя (BP);
* состояние потока;
* использованное процессорное время;
* приоритет потока.

**Есть два основных подхода в реализации потока**

* в пользовательском режиме (таблица потоков размещается в адресном пространстве процесса, и управление потоками осуществляется внутри процесса, при этом ядро ОС ничего не знает и управляет только процессами);
* в режиме ядра (таблица потоков размещается в ядре и управление потоками осуществляется там же).

С целью совмещения преимуществ реализации потоков на уровне ядра (простая реализация) и на уровне пользователей (высокая производительность) были разработаны несколько методов: один из этих методов заключается в использовании модели, в которой ядро управляет только своими потоками, но некоторые из этих потоков также могут содержать по несколько своих потоков на пользовательском уровне.

**Состояния потока**

Как и процесс, поток может находится в одном из состояния, но кроме этого для потока предусмотрено состояние приостановка, в котором поток приостанавливается, но не завершается и может быть продолжен (что-то похожее на паузу).

**Планирование**

Ключевым вопросом планирования является момент выбора принятия решения. Во-первых, когда создается новый поток/процесс, необходимо решить какой поток/процесс нужно запустить: родительский или дочерний. Поскольку они оба находится в состоянии готовности, поэтому планировщик может запустить любой из них.

Во-вторых, когда какой-либо процесс или поток завершает свою работу, планировщик должен выбрать из процессов или потоков один, находящийся в состоянии готовности и запустить его. Но, если такого процесса или потока нет, то запускается специальный холостой процесс или поток.

В-третьих, когда процесс или поток блокируется, то есть переходит в состояние ожидания, необходимо выбрать и запустить другой процесс или поток, при этом причина блокировки может повлиять на выбор.

В-четвертых, когда наступает событие, ожидаемое каким-либо процессом или потоком необходимо выбрать какой процесс или поток запустить, тот, который был приостановлен, или какой-либо другой.

**Алгоритмы планирования**

Можно разделить на две категории, согласно их поведению.

1) Алгоритмы кооперативной многозадачности (эти алгоритмы выбирают процесс и позволяют ему выполняться вплоть до завершения, либо до момента, когда процесс сам не освободит процессор);

2) Алгоритмы, вытесняющие многозадачность (эти алгоритмы выбирают процесс и позволяют ему работать некоторое максимально возможное фиксированное время, если к концу заданного интервала времени процесс ещё работает, то он приостанавливается, и управление переходит к другому процессу).

В различных классах многозадачных ОС используются разные категории алгоритмов планирования.

В ОС С ПАКЕТНОЙ ОБРАБОТКОЙ приемлемы алгоритмы кооперативной многозадачности. Однако, могут использоваться и алгоритмы, вытесняющие многозадачность, но с большим временем, отводимым каждому процессу для работы.

В ОС С РАЗДЕЛЕНИЕМ ВРЕМЕНИ необходимы алгоритмы, вытесняющие многозадачность, чтобы предотвратить захват процессора одним процессом.

В ОС С РЕАЛЬНЫМ ВРЕМЕНЕМ приемлемы алгоритмы, вытесняющие многозадачность, поскольку все процессоры должны завершаться в определенное время.

Отметим, что алгоритмы планирования процессов также могут быть применены к планированию потоков.

Например, **алгоритм FIFO** – это неприоритетный алгоритм планирования, согласно которому процессор предоставляется процессам в том порядке, в котором они его запрашивают. Согласно этому алгоритму формируется единая очередь ждущих процессов. Как только создается первый процесс, он немедленно получает процессор и работает столько, сколько необходимо. Остальные процессы становятся в очередь, когда текущий процесс блокируется, запускается первый процесс из очереди, а когда ожидание блокированного процесса завершено, он снова попадает в очередь.

**Алгоритм наименьшего времени выполнения** – является версией алгоритма FIFO, согласно которой планировщик каждый раз выбирает процесс с наименьшим временем выполнения. В этом случае заранее необходимо знать время выполнения процесса.

**Алгоритм циклического планирования** – согласно этому алгоритму каждому процессу, в списке процессов выделяется определенный квант времени. Если к концу предоставленного кванта времени процесс всё ещё выполняется, то он блокируется, а процессор передается следующему процессу. Если же процесс блокируется или прекращает свою работу, процессор также передается следующему процессу. Отметим, что слишком малый квант времени приведет к частому переключение процессов и небольшой эффективности. Слишком большой квант может привести к медленному реагированию на короткие интерактивные запросы.

Представим, что переключение процессов занимает 1 миллисекунду, допустим, что квант времени равен 4 миллисекунды. В этом случае 20% процессорного времени уйдет только на переключение процессов.

Можно увеличить квант времени до 100 миллисекунд, в этом случае пропадает только один процент процессорного времени. Тогда представим ситуации, в которой будет 10 процессов, которые будут одновременно ожидать некоторого события. При наступлении этого события все соответствующие процессы перейдут в состояние готовности. Какой-то из этих процессов начинает работать первым, следующему процессу придется ждать 100 миллисекунд, следующему процессу 200 миллисекунд и.т.д. Последнему процессу возможно придется ждать целую секунду, если остальные процессы не блокируются до этого времени.

На практике обычно выбирается квант времени от 20-50 миллисекунд.

**Алгоритм приоритетного планирования** - является версией предыдущего алгоритма, согласно которой каждому процессу присваивается приоритет. И процессор передается готовому процессу с наибольшим приоритетом. Чтобы предотвратить бесконечную работу процессов с высоким приоритетом, планировщик может уменьшать приоритет процесса, также для этого возможно предоставление каждому процессу максимального отрезка времени работы. Как только время закончилось, то управление передается следующему по приоритету процессу. Иногда процессы группируются в классы по приоритетам и используют алгоритм приоритетного планирования среди классов, но внутри каждого класса используется алгоритм циклического планирования. Пока в классе определенного приоритета, есть готовые к работе процессы, то они выполняются один за другим, согласно алгоритму циклического планирования. При этом процессор с более низким классом приоритета выполняться не будут, пока есть готовые к работе процессы с более высоким классом приоритета.

**Алгоритм справедливого планирования** – предполагает, что каждому пользователю достается некоторая доля процессорного времени. И планировщик выбирает процесс в соответствии с этим фактом. Рассмотрим пример, в котором существует пять процессов A, B, C, D, E, которые принадлежат одному пользователю.

Если использовать циклический алгоритм каждому пользователю относится по 50% процессорного времени. В этом случае цепочка переключения процессов, будет следующая A-E-B-E-C-E-D-E и обратно в начало.

Но в алгоритме справедливого планирования цепочка переключения процессов, будет следующая A-B-E-C-D-E и в начало. И процессорное время будет разделено между пользователями на 30% и 70%.

**Синхронизация**

В многозадачных ОС процессы могут совместно использовать общее хранилище данных, которые представляет собой структуру данных ядра или файлов. Каждый процесс может считывать из общего хранилища данных и записывать туда информацию.

Рассмотрим пример синхронизации на буфере печати.

Если процессу требуется вывести на печать некоторый документ, то он помещает нужные страницы в буфер печати. Другой процесс периодически проверяет наличие страниц, которые нужно напечатать. Печатает страницу и удаляет её из буфера печати. Представим, что буфер печати представляет собой массив, в элементах которого будут храниться страницы, которые необходима напечатать.

Пусть в данный момент времени в буфере печати уже находятся три страницы, в процессе A и B решают добавить ещё под одной страницы каждый, в этом случае возможно ситуация, когда процесс А определяет индекс свободного элемента, в который можно сохранить страницу. И сохраняет его (индекс) в некоторой своей локальной переменной, после этого процессор передается процессу В, который, в свою очередь, определяет индекс свободного элемента (это тоже тройка) и сохраняет его в своей локальной переменной. В данный момент времени оба процесса считают, что следующий свободный элемент третий; процесс В сохраняет в третий элемент буфера печати свою страницу и занимается своими делами, наконец, процессор снова переходит процессу А и он продолжает на месте, на котором остановился, то есть он обращается к своей локальной переменной, в которой сохранен индекс свободного элемента, считывает её значение и сохраняет в третий элемент буфера печати страницу, удаляя при этом страницу, записанную туда процессором В. Так как структура буфера печати не нарушена, то процесс, отвечающий за печать страниц, ничего не заподозрит, но страница процесса В будет утрачена.

Ситуация, в которой два или более процесса считывают и записывают данные одновременно и конечный результат зависит от того, кто из них был первым, **называется гонкой.**

**1 вариант: критические секции и взаимное исключение:** часть программы, в которой есть обращение к разделяемым данным, или выполняется какое-то другое действие, которое может привести к гонке, называется критической секцией, если удастся предотвратить одновременное нахождение более одного процесса критической секции можно избежать гонок; основным способом предотвращения гонок является **взаимное исключение**, то есть запрет одновременной записи и чтения разделяемых данных более, чем одним процессом, а именно в тот момент, когда один процесс будет использовать разделяемые данные, другому процессу это делать запрещено.



**Запрещение прерываний:** самое простое решение состоит в запрещении прерываний, поскольку процессор переключается с одного процесса на другой только по прерыванию, а исключение к прерыванию исключает передачу процессора другому процессу.  
Такое решение неприемлемым в качестве механизма взаимного исключения для пользовательских процессов. Во-первых, что, если процесс отключил прерывание и в результате какого-либо сбоя не включил их обратно.

В многопроцессорных ОС запрещение прерываний повлияет только на тот процессор, который выполняется текущим процессом, а остальные процессоры продолжат работу и сохранят доступ к разделяемым ресурсам.

Для ядра ОС запрещение прерываний бывает полезным при работе с объектами ядра.

**Переменные блокировки**

Ещё одно решение заключается в использовании процессами совместной переменной блокировки (изначально равной нулю). Если процесс хочет попасть в критическую секцию, он предварительно проверяет значение переменной блокировки. Если переменная блокировки содержит значение равное нулю, то процесс присваивает ей значение равное единице и входит в критическую секцию. Если же переменная блокировки содержит значение равное единицы, то процесс ждет, пока значение не смениться, но ноль.

У данного метода так же имеются проблемы. Может произойти состояние гонок.

Один процесс считывает значение из переменной блокировки, обнаруживает, что там значение 0, но прежде чем он успевает изменить его процессор переходит к другому процессу, который так же смотрит, что значение =0, изменяет его на 1 и заходит в критическую секцию. Когда первый процесс снова получит процессор, он тоже заменит значение переменной на 1 и два процесса одновременно оказываются в критической секции.

**Поочередный доступ к критической секции**

for (;;)

{

..

while (turn!=0);

…//критическая секция

turn = 1;

…//вне критической секции

}

for(;;)

{

while (turn !=1);

…// критическая секция

turn =0;

…//вне критической секции

}

Целочисленная переменная turn изначально равна нулю и отслеживает чья очередь входить в критическую секцию.

В начале первый процесс проверяет значение turn, которое равно нулю, и входит в критическую секцию.

Второй процесс проверяет значение turn, которое не равно единице, и после этого входит в цикл активного ожидания, при этом непрерывно проверяя, когда значение turn будет равно единице.

Блокировка, использующая активное ожидание, называется циклической блокировкой. В общем случае, циклических блокировок следует избегать, поскольку это бесцельная трата процессорного времени.

Когда первый процесс покидает критическую секцию, то он изменяет значение переменной turn на единицу, позволяя второму процессу попасть в критическую секцию.

Предположим, что второй процесс быстро покидает свою критическую секцию и значение переменной turn будет равно нулю. Если теперь первый процесс выполнит весь цикл быстрее, чем второй процесс вне критической секции, то тогда первый процесс после выхода из критической секции снова изменит значение переменной turn на 1, и оба процесса будут находиться вне критической секции. При этом, если первый процесс завершает работу вне критической секции, то он не сможет пойти в критическую секцию, так как значение переменной turn равно единице, и второй процесс всё ещё находится вне критической секции. Таким образом, первый процесс вынужден ждать, пока второй процесс изменит свое значение turn на 0.

Из рассмотренного примера можно сделать вывод: метод поочередного доступа к критической секции не слишком эффективен, если один процесс существенно медленнее другого.

**Алгоритм Петерсона**

Был разработан для синхронизации двух процессов, но может быть обобщен для произвольного количества процессов.

Алгоритм заключается в использовании двух функции: void EnterCS, void LeaveCS.

bool interested[2] = { false, false }; //индикатор интереса

int turn = 0; //определяет очередность процессов

void EnterCS(int process)

{

int other= 1-process; //определим номер второго процесса

interested[process] = true; //установим индикатор процесса

turn = other;//установим очередность

//спин-блокировка

while (turn == other && interested[other]);

}

void LeaveCS9int process)

{

interested[process]=false;

}

Прежде, чем войти в критическую секцию, процесс должен вызвать функцию EnterCS со своим номером в качестве параметра. Поэтому процессу при необходимости придется подождать прежде, чем входить в критическую секцию. После выхода из критической секции процесс должен вызвать функцию LeaveCS, чтобы обозначить свой выход и разрешить другому процессу вход в критическую секцию.

Исходно оба процесса находятся вне критической секции. Первый процесс вызывает функцию EnterCS с параметром 0, в котором задает значение идентификатора интереса и значение переменной turn=1. Поскольку второй процесс пока не заинтересован в попадании в критическую секцию, функция EnterCS возвращается управление. Теперь, если второй процесс вызовет функции EnterCS с параметром 1, ему придется подождать, пока индикатор интереса первого процесса примет значение false. А это произойдет только в тот момент, когда первый процесс вызовет функцию LeaveCS с параметром 0, чтобы посетит критическую секцию.

Представим, что оба процесса практически одновременно вызвали функцию EnterCS, в переменную turn сохранится номер того процесса, который был вторым. Предположим, что вторым был процесс с номером 1, а значение переменной turn было равно 1. Когда оба процесса дойдут до оператора while процесс с номером 0 войдет в критическую секцию, а другой процесс останется в цикле и будет ждать, когда процесс с номером 0 выйдет из критической секции.

**Блокировка шины памяти**

void \_fastcall InterLockedIncrement(long volatile \*addend)

{

\_asm

{

lock inc dword ptr [ECX]

}

}

void \_fastcall InterLockedDecrement(long volatile \*addend)

{

\_asm

{

lock dec dword ptr[ECX]

}

}

Современные центральные процессоры способны блокировать шину памяти, запрещая другим процессорам доступ к памяти до тех пор, пока не будет выполнена определенная команда.

Процессор архитектуры x86 используют специальную команду lock, которая функционирует со следующими командами, которые используются операндом памяти:

* ADD;
* ADC;
* AND;
* BTC;
* BTR;
* BTS;
* CMPXCHG;
* CMPXCHG8B;
* DEC;
* INC.

void \_fastcall InterLockedExchange(long volatile \*target, long value)

{

\_asm

{

lock xchg dword ptr [ECX], EDX

}

}

Блокировка шины памяти существенно отличается от запрета на прерывание, если при запрете прерываний ничто не помешает другому процессору подключится к шине памяти и получить к ней доступ, то в блокировке шины памяти этого не произойдет.

**Приостановка и активизация**

Основным недостатком методов взаимного исключения является использование циклических блокировок. По сути, они все сводятся к следующему: когда процессу требуется войти в критическую секцию, он проверяет разрешенный вход. Если вход запрещен, процесс просто начинает цикл ожидания разрешения. Такой подход приводит не только к трате процессорного времени, но также приводит к тупиковым ситуациям, называемых **взаимной блокировкой** (dead-block).

Например, если процесс А должен выполняться с высоким приоритетом, а процесс В с низким, при этом правило планирования их выполнения предусматривает, что процесс А выполняется сразу же после входа в состояние готовности. В определенный момент времени, когда процесс В находится в критической секции – процесс А переходит в состояние готовности. Например, после завершения операции ввода/вывода. Теперь процесс А при попытке попасть в критическую секцию начинает цикл активного ожидания, но, поскольку, пока выполняется процесс А, выполнение процесса В не планируется. И поэтому он не может выйти из критической секции. В сложившейся ситуации процессы А и В обречены на бесконечное ожидание.

Вместо того, чтобы использовать циклическую блокировку, можно блокировать процесс, приостанавливая его до тех пор, пока он не будет активизирован другим процессом.

**Задача производителя и потребителя**

Два процесса используют общий буфер фиксированного размера, но один из них производитель, который помещает информацию в буфер, а другой потребитель, который извлекает её оттуда.

Когда производителю требуется поместить в буфер очередную запись, если при этом буфер уже заполнен, процесс блокируется до тех пор, пока потребитель не извлечет из буфера, как минимум, одну запись.

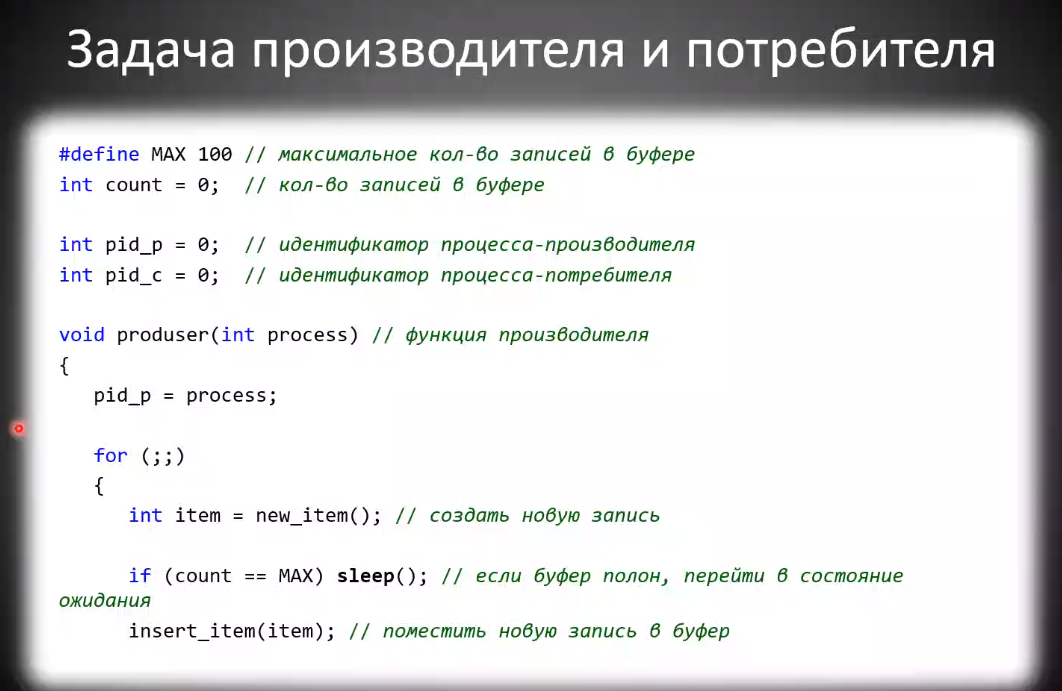
Если потребителю нужно извлечь из буфера информацию, но при этом буфер пуст, то потребитель блокирует процесс до тех пор, пока производитель не поместит запись в буфер или активизирует его.

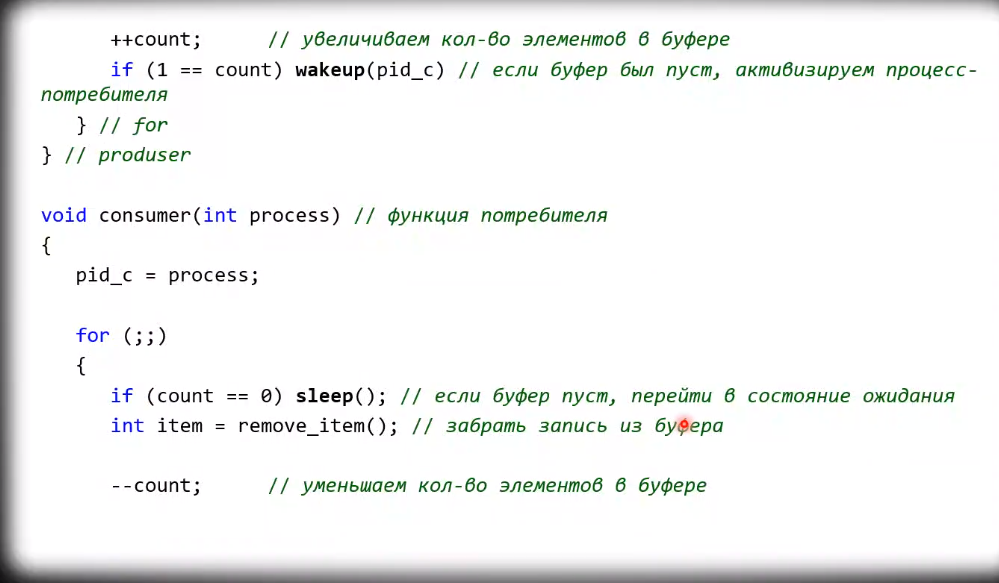
Используются специальные системные вызовы sleep() и wakeup(). Системный вызов sleep блокирует вызывающий процесс, а wakeup – активирует указанный адрес. В различных ОС эти системные вызовы могут называться по-разному и могут иметь различные параметры.

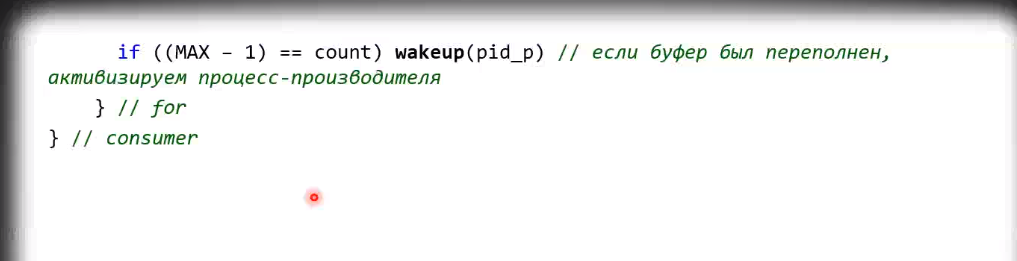
В приведенном примере возможно состояние 00, поскольку доступ к переменной count. Может возникнуть ситуация, в которой буфер пуст и потребитель только что получит значение переменной count, чтобы проверить не равно ли оно 0. В этот момент планировщик передал процессор производителю, который поместил новую запись в буфер и увеличил значение переменной count на 1. Зная, что перед этим оно было равно 0, и потребитель находится в состоянии ожидания, производитель активизирует его с помощью команды wakeup.

Однако, потребитель не был в состоянии ожидания, так что сигнал активации прошел впустую. Когда потребитель снова получит процессорное время, он вернется к полученному значению переменной cout и обнаружит, что оно равно 0, и уйдет в состояние ожидания. Рано или поздно производитель заполнит буфер и также уйдет в состояние ожидания. Оба процесса так и останутся в этом состоянии, то есть возникла взаимоблокировка.

Проблема заключается в том, что сигнал активации, отправленный процессом, не состоящий в состоянии ожидания, пропал, если бы этого не было, то проблемы тоже бы не было.



****

****

**Семафоры**

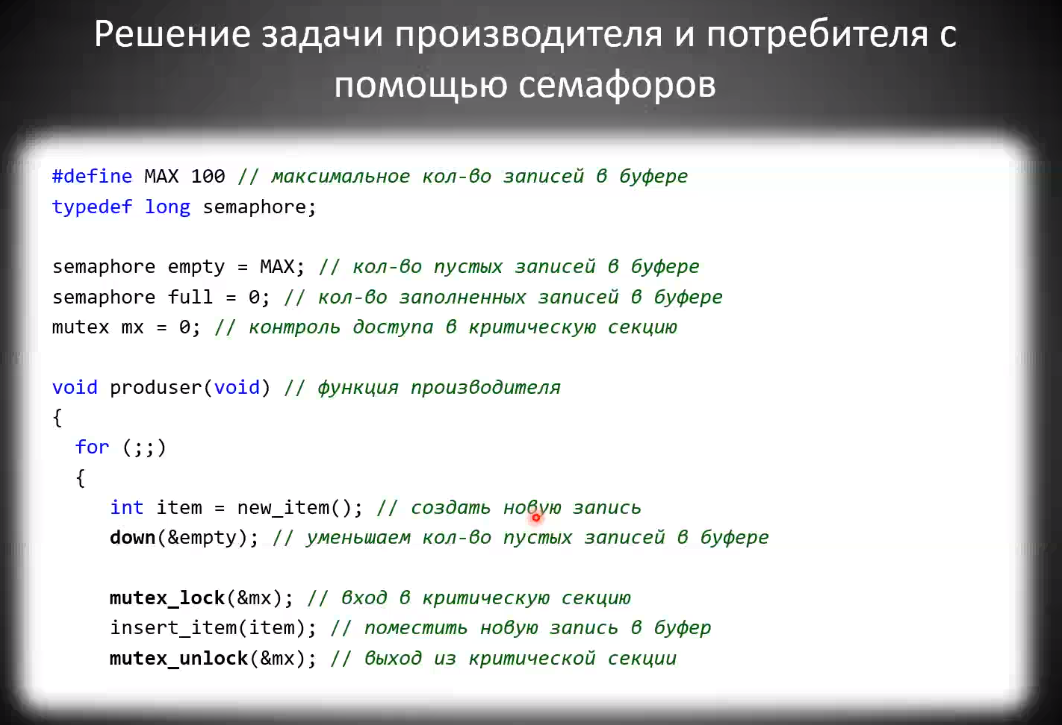
Семафоры – это тип данных, предназначенные для подсчета сигналов активации процессов.

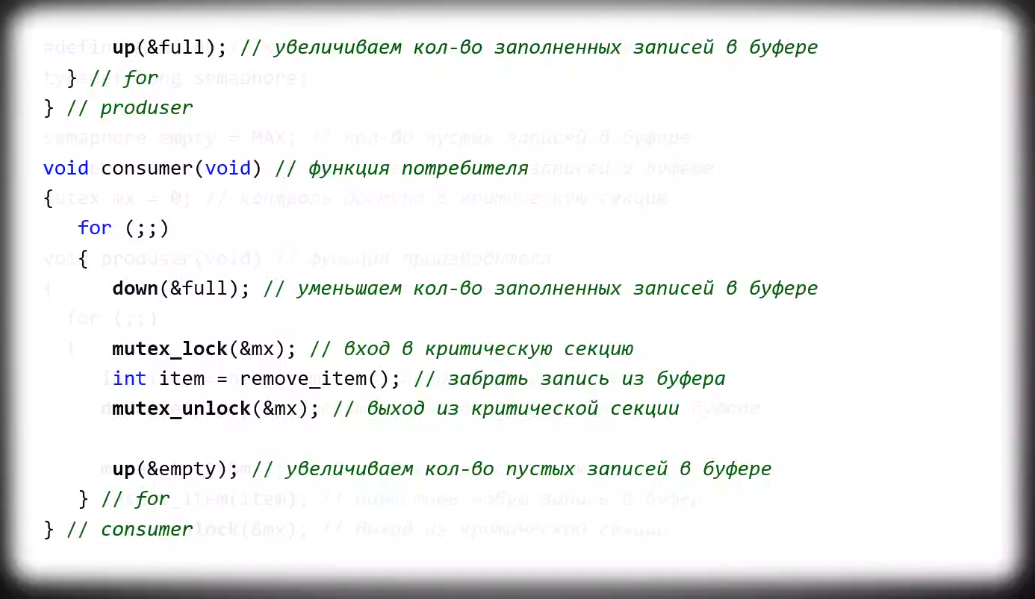
Семафор может иметь нулевое значение, в случае отсутствия положительных сигналов, или некоторое положительное целое число, соответствующее количеству отложенных сигналов активации.

Было предложено две операции:

* down (сравнивает значение семафора с нулем, если значение больше 0, то оно уменьшается, то есть расходуется один из сохраненных сигналов активации; если значение семафора равно 0, то процесс переходит в состояние ожидания);
* up (увеличивает значение семафора; если с этим семафором связано один или несколько ожидающих процессов, один из них активируется, при этом значение семафора остается равным нулю, но число ожидающих процессов уменьшается на 1).

Все операции проверки значения семафора, его изменение, приостановка или активация процесса должно выполняться, как единое и неделимое целое, тем самым гарантируется, что после начала операции ни один процесс не получит доступа к семафору до окончания операции или блокирования процесса.





**Мьютексы**

Мьютекс может находиться в одном из двух состояний: заблокирован и не заблокирован, которые соответственно устанавливается функциями mutex\_lock и mutex\_unlock.

Когда процесс собирается войти в критическую секцию, он вызывает функцию mutex\_lock, если мьютекс не заблокирован, то вызывающий процесс войдет в критическую секцию, если же мьютекс заблокирован, то этот процесс блокируется до тех пор, пока другой процесс находится в критической секции и не выйдет из неё, вызвав функцию mutex\_unlock.

Если мьютекс блокирует сразу несколько процессов, то из них выбирается один случайным образом.

Классические задачи синхронизации

Существует много задач синхронизации, изучение которых полезно для моделирования процессов, соревнующихся за монопольный доступ к разделяемым ресурсам.

* **Задача обедающих философов**

Была сформулирована и решена в 1995 г.

Формулировка: пять философов сидят за круглым столом и у каждого есть тарелка со спагетти. Между каждыми двумя тарелками лежит одна вилка. Каждый философ может либо есть, либо размышлять. Философ может есть только тогда, когда держит две вилки: взятую справа и слева. Когда философ голоден, он пытается получить две вилки (левую и правую) в любом порядке. Если ему удалось получить две вилки, то он некоторое время ест, а затем кладет вилки обратно и продолжает размышлять.



Очевидным решением является постоянное выполнение каждым философов конкретных действия:

1. Размышлять, пока не освободится левая вилка, и, когда она освободится, взять её;
2. Размышлять, пока не освободится правая вилка, и, когда она освободится взять её;
3. Есть;
4. Положить левую вилку;
5. Положить правую вилку.

Это решение не верно, так как приводит к блокировке процесса. Например, если пять философов возьмут одновременно свои левые вилки, то каждый из философов останется без правой вилки и произойдет взаимоблокировка.

Тогда можно изменить алгоритм: после получения левой вилки проверять доступность правой. Если правая вилка недоступна, то философ отдает левую вилку обратно, ждет некоторое время и повторяет процесс.

Это решение также является неверным, но по другой причине. Если все пять философов возьмут одновременно свои левые вилки, то обнаружат отсутствие правой вилки, положат левую вилку обратно, подождет некоторое время и снова возьмет левую вилку и так до бесконечности.

Ситуация, в которой процесс продолжает работать сколь угодно долго, но не может добиться хоть какого-то прогресса, называется **голоданием процесса**.

* **Задача читателей и писателей**

Формулировка: модулирует доступ к БД, к которой пытается получить доступ сразу несколько процессов. Можно разрешить одновременно считывать данные из базы, но если процесс записывает информацию в базу, то доступ остальных процессов должен быть прекращен (даже доступ для чтения).

Первый читатель блокирует доступ к БД с помощью мьютекса, так что последующие читатели просто увеличивают значение счетчика. По мере ухода читателей из БД, счетчик уменьшается, и последний читатель освобождает доступ к БД.

Возможна ситуация, когда при наличии читателей, которые работают с БД, процесс, которому необходим монопольный доступ к базе, будет заблокирован. Пока с базой работает хотя бы один активный читатель, доступ остальным читателям разрешается, а писатели вынуждены ждать.

Если предположить, что новый читатель запрашивает доступ каждые 2 секунды, а работать с базой ему нужно 5 секунд, то писатель не получит доступа к базе никогда. Чтобы избежать подобной ситуации, необходимо внести изменения: если писатель ждет доступа к БД, то новый читатель доступа не получает и становится в очередь за писателей. Теперь писателю нужно подождать, пока доступ освободят, имеющие его читатели, но не нужно пропускать вперед читателей, пришедших после него. Недостатком данного решения является понижение производительности.

* **Задача спящего брадобрея**

Формулировка: в парикмахерской есть один брадобрей. Его кресло и несколько стульев для клиента. Если клиентов нет, то брадобрей сидит в своем кресле и спит. Если в парикмахерскую приходит клиент, то он должен разбудить брадобрея. Если клиент приходит и видит, что брадобрей занят, он либо садится на стул, если место есть, либо уходит, если места нет. Необходимо избежать состояние голода или взаимоблокировки.

Придя в парикмахерскую, клиент запрашивает доступ для входа в критическую секцию. Если вслед за ним придет ещё один клиент, то ему не удастся ничего-либо сделать, пока первый клиент не выйдет из критической секции. Клиент проверяет наличие свободных стульев, а случае неудачи – уходит. Если свободный стул есть, то клиент увеличивает значение семафора, тем самым, активизируя брадобрея. В этот момент клиент и брадобрей активен. Когда клиент выйдет из критической секции, в неё войдет брадобрей, проделывает некоторые операции и начинает стричь клиента. По окончанию стрижки клиент покидает парикмахерскую.

**Виртуальная память**

В случае нехватки оперативной памяти, необходимой для запуска и работы процесса, используется виртуальная память, чтобы восполнить эту нехватку.

Виртуальная память – это технология управления памятью, сочетающая в себе одновременное использование оперативной памяти и временное хранилище на жестком диске или другом внешнем накопителе.

Если оперативной памяти недостаточно, то данные из оперативной памяти помещаются в хранилище, которое называется файлом подкачки. Перемещение данных в файл подкачки освобождает достаточное оперативной памяти. При этом чтение из оперативной памяти выполняется гораздо быстрее, чем из файла подкачки.

Виртуальная память может использоваться как в однозадачных, так и в многозадачных операционных системах. В памяти одновременно могут находиться различные части нескольких процессов. Когда процесс ждет перемещения в оперативную память очередной своей части, то он находится в состоянии ожидания и не может исполняться.

Модуль операционной системы, ответственный за управление виртуальной памятью, называется менеджер виртуальной памяти.

**Страничная организация виртуальной памяти**

В большинстве ОС используется страничная организация виртуальной памяти, в которой все пространство виртуальной памяти разбивается на фиксированные блоки, называемые страницами. Блоки оперативной памяти, которые хранят страницы виртуальной памяти, имеют такой же размер и называются страничными кадрами.

Каждой странице виртуальной памяти присваивается номер. При обращении процесса к памяти, процессор указывает виртуальный адрес, который состоит из номера виртуальной страницы и смещения относительно начала этой страницы. Затем менеджер виртуальной памяти преобразует виртуальный адрес в физический адрес, который состоит из номера страницы в оперативной памяти и смещения относительно её начала. Причем преобразованию подвергается только номер страницы, так как смещения будут одинаковыми.

Если менеджер памяти обнаруживает, что необходимая виртуальная страница отсутствует в оперативной памяти, выполняется прерывание центрального процессора, называемая страничным прерыванием. После этого согласно алгоритму замещения страниц, выбирается страничный кадр в оперативной памяти и его содержимое записывается в файл подкачки, затем требуемые страницы считываются из файла подкачки в только что освободившейся страничный кадр.

**Таблица страниц**

Преобразование виртуального адреса в физический, осуществляется с помощью таблицы страниц.

Таблица страниц определяет какие виртуальные страницы загружены в оперативную память и в каких страничных кадрах они находятся.

Каждый процесс имеют свою собственную таблицу страниц, благодаря чему у него есть свое собственное виртуальное адресное пространство.

Структура записи таблицы страниц зависит от конкретных особенностей аппаратной архитектуры, как правило, каждая такая запись содержит номер виртуальной страницы, номер страничного кадра, а также набор следующих атрибутов:

1. Признак присутствия (который определяет загружены ли виртуальные страницы в оперативную память);
2. Признак использования (который используется при замещении страниц оперативной памяти);
3. Признак модификации (который определяет была ли изменена страница в процессе работы);
4. Право доступа (который используется с целю защиты виртуальной страницы и определяет какой вид доступа к ней разрешен (чтение, запись и исполнение)).

Так как таблица страниц содержит только ту информацию, которая необходима для перевода из виртуального в физический, то нет необходимости включать информацию о том, где хранится виртуальная страница.

**Многоуровневые таблицы страниц**

Для того, чтобы избежать проблему постоянно размещения, в оперативной памяти огромных таблиц страниц, их разбивают на фрагменты. В оперативной памяти хранятся лишь некоторые необходимые фрагменты таблицы страниц.

Наиболее распространённым способом разбиения является организация, так называемой, **многоуровневой таблицы страниц**. Для примера рассмотрим двухуровневую таблицу страниц: таблицы первого уровня состоят из записей, содержащих информацию о виртуальных страницах, в которых хранятся таблицы второго уровня; в свою очередь, таблицы второго уровня содержат информацию о виртуальных страницах, хранящих данные процесса.

Таким образом, для размещения процесса с большим объемом занимаемой памяти, достаточно иметь в оперативной памяти одну таблицу первого уровня и несколько таблиц второго уровня.

В этом случае виртуальный адрес в двухуровневые таблицы страниц будет содержать номер виртуальной страницы первого уровня, номер виртуальной таблицы второго уровня и смещение относительно начала этой страницы.

Отметим, что наличие нескольких уровней снижает производительность менеджера виртуальной памяти. Несмотря на то, что размеры таблиц на каждом уровне подобраны так, чтобы таблица помещалась целиком внутри одной страницы, обращение к каждому уровню – это отдельное обращение к памяти.

Таким образом, для преобразования виртуального адреса в физический может потребоваться несколько обращений к памяти.

Количество уровней таблицы страниц зависит от конкретных особенностей аппаратной архитектуры. Например, в архитектуре x86 используется именно двухуровневая архитектура страниц.

**Буфер ассоциативной трансляции**

Поиск номера страничного кадра, соответствующего нужной виртуальной страницы, занимает много времени, так как требует обращение к оперативной памяти, в которой хранится таблица страниц.

Проблему поиска можно решить путем снабжения компьютеров специальным аппаратным устройством для преобразования виртуального адреса в физический без обращения к таблице страниц. Это устройство называется буфер ассоциативной трансляции (TLB), который представляет собой быструю кэш-память, хранящую только необходимую на данный момент часть таблицы страниц.

Записи в буфере ассоциативной трансляции точно соответствуют записям в таблице страниц. Когда виртуальный адрес передается для преобразования, диспетчер виртуальной памяти сначала убеждается в том, что номер виртуальной страницы присутствует в буфере ассоциативной трансляции. Если номер виртуальной страницы был найден, то вся необходимая информация берется из буфера без перехода к таблице страниц.

Если номер виртуальной страницы не находится в буфере ассоциативности трансляции, то менеджер виртуальной памяти выполняет поиск в таблице страниц, затем заменяет одну из записей в буфере ассоциативной трансляции, найденной записью. Когда запись удаляется из буфера ассоциативной трансляции, признак модификации копируется в таблицу страниц, а другие атрибуты уже находятся там.

**Размеры страниц**

Отдельной проблемой менеджера виртуальной памяти со страничной организацией памяти является размер страниц. С одной стороны, страница не должна быть слишком большая, так как это может привести к неэффективному использованию памяти и копированию слишком больших объемов данных, а с другой стороны страницы не должны быть слишком маленькими, так как это приведет к чрезмерному увеличению таблицы страниц.

Как правило, размер страницы зависит от аппаратной архитектуры. В архитектуре х86 размер страницы равен 4 Кбайт.

**Алгоритмы замещения страниц**

Когда происходит страничное прерывание, ОС должна выбрать страницу для удаления из оперативной памяти. Чтобы освободить место для страницы, которую нужно загрузить в оперативную память. Если необходимо удалить страницу, которая была изменена за время своего нахождения в оперативной памяти, её необходимо переписать в файл подкачки, чтобы обновить копию, находящуюся там.

**Алгоритм NRU** (неиспользовавшийся в последнее время) – алгоритм заключается в применении признаков использования и модификации в таблице страниц. Когда процесс запускается, оба этих признака для всех таблиц процесса установлены в ноль. Периодически признак использования сбрасывается в ноль, чтобы выявить страницы, к которым давно не было обращения. Когда произойдет страничное прерывание, ОС проверяет все страницы и делит их на 4 категории, на основании текущих значений, признаков использования и модификации.

1 категория: страница не изменялась и не используется;

2 категория: страница изменялась и не используется;

3 категория: страница не изменялась и используется;

4 категория: страница изменялась и используется.

Алгоритм NRU удаляет из памяти страницы с помощью случайного выбора страницы в категории с наименьшим номером, кроме категории 1. Подразумевается, что лучше выгрузить измененную страницу, но не используемую, чем удалить часто используемую страницу.

**Алгоритм FIFO –** заключается в использовании списка виртуальных страниц, находящихся в оперативной памяти. Когда произойдет страничное прерывание, ОС удаляет из оперативной памяти первую страницу из списка. После этого, удаленная страница покидает список, а новая страница будет добавлена в его конец. Применение алгоритма FIFO при обработке страничного прерывания может привести к ситуации, когда из оперативной памяти будут удалены часто используемые страницы.

**Алгоритм вторая попытка –** представляет собой модификацию алгоритма FIFO, которая позволяет избежать проблему удаления из оперативной памяти часто используемых страниц. То есть, данный алгоритм заключается в следующем: если признак использования равен нулю, то страница не только находится в памяти долго, но и не используется, поэтому должна быть удалена при обработке страничного прерывания. Если же признак использования равен единице, то ему присваивается ноль, страница переносится в конец списка, то есть считается, что она была только что загружена в оперативную память. Затем процедура повторяется. Таким образом, алгоритм вторая попытка ищет в списке самую старую страницу, к которой не было обращений. Если признак использования для всех = 0, то данный алгоритм работает, как алгоритм FIFO.

**Алгоритм часы –** алгоритм вторая попытка является корректным, но не слишком эффективным, потому что постоянно передвигает страницы по списку. Поэтому лучше хранить все страничные блоки в кольцевом списке в форме часов (стрелка указывает на старейшую страницу). Когда происходит страничное прерывание, проверяется та страница, на которую направлена стрелка, если её признак использования равен 0, то страница удаляется из оперативной памяти, а на её место встает новая страница. А стрелка сдвигается на одну позицию. Если же признак использования равен 1, то он сбрасывается, а стрелка перемещается к следующей странице. Такой процесс повторяется до тех пор, пока не найдется признак использования равный 0.

**Сегментная организация виртуальной памяти**

Все пространство виртуальной памяти и оперативной памяти разбивается на блоки произвольного размера, называемые сегментами. При запуске процесса ОС выдается таблице сегментов, аналогичную таблице страниц (для каждого сегмента указывая номер сегмента, начальный и физический адрес сегмента в оперативной памяти, размер сегмента и различные атрибуты. Вроде тех, что используется в таблице страниц).

Виртуальный адрес при сегментной реализации памяти, состоит из номера сегмента и смещения в сегменте.

Менеджер виртуальной памяти с сегментной организацией функционирует аналогично менеджеру виртуальной памяти со страничной организацией. То есть, время от времени происходят прерывания, связанные с отсутствием нужных сегментов в оперативной памяти. При необходимости некоторые сегменты удаляются из оперативной памяти. Отметим, что при загрузке сегментов менеджер отыскивает подходящий по размеру участок свободной оперативной памяти.

Недостатком сегментной организации виртуальной памяти является фрагментация оперативной памяти, а также более медленная по сравнению со страничной организацией преобразование адреса.

**Сегментно-страничная организация виртуальной памяти**

При сегментно-страничной организации виртуальной памяти все пространство виртуальной оперативной памяти разбивается на сегменте, которые в свою очередь разбиваются на страницы. В этом случае, виртуальный адрес состоит из номера сегмента, номера страницы внутри сегмента и смещение внутри страницы. Соответственно для преобразования виртуального адреса в физический используется две таблицы: таблица сегментов (связывающего номер сегмента с таблицей страниц) и отдельная таблица страниц для каждого сегмента.

**Файл. Файловые системы**.

Во время работы процесс может хранить данные в собственном адресном пространстве. Однако, после завершения работы процесса, данные, хранящиеся в его адресном пространстве, теряются. Для многих приложений исчезновение данных после завершения работы процесса – неприемлемо.

Кроме того, часто возникает необходимость получить доступ к одним и тем же данным. Если эти данные будут хранится в адресном пространстве одного процесса, то допуск к ним будет только у этого процесса.

Решением всех этих проблем обычно состоит в хранении данных в файлах.

Файл – поименованная область диска или другой области данных, которая обрабатывается, как единое целое.

Для управления этими данными создаются соответствующие файловые системы. Также именование файла, их защита и доступ к ним также является важными задачами файловой системы.

**Именование файла**

Важной характеристикой любого файла является то, как он именуется. Имя файла дается при его создании, после этого различные процессы могут получить доступ к файлу по его имени. В современных файловых системах правило именования файла различаются. Например, в некоторых системах различают прописные и строчные символы в имени файлов, когда другие нет.

Обычно имя файла состоит из двух частей, разделенных точкой. Часть имени файла после точки является расширением имени файла. Расширение файла предназначено, чтобы пользователь или программа могли определить тип хранящихся в файле данных. В некоторых ОС, например Linux, расширение файлов является просто соглашением, и ОС не принуждает пользователя их строго придерживаться. В других ОС, например Windows, каждому расширению файла предназначена программа, которая будет запускаться каждый раз, когда пользователь будет обращаться к соответствующему файлу.

**Доступ к файлам**

В ранних ОС предоставляется только последовательный доступ к файлам, поскольку данные можно было хранить только на магнитной ленте. Последовательный доступ означает, что чтение и запись данных осуществляется по порядку: от начала и до конца файла.

С появлением дисков стало возможно обращаться к файлам произвольным образом. Под произвольным доступом понимается, что чтение и запись данных может осуществляться в произвольном порядке. При произвольном доступе обращение к любому фрагменту файла выполняется за равные промежутки времени, не зависящие от размера файла. В отличии от последовательного доступа, при котором, чем дальше фрагмент, тем больше требуется времени для доступа.

**Файловые системы**

На сегодняшний день существует большое количество файловых систем, созданных для различных устройств внешней памяти и различных ОС.

В них используются соответственно различные принципы размещения данных. Файловая система определяет способ организации данных на диске или каком-либо ином внешнем накопители. Компонент ОС реализующий работу с файлами, согласно соответствующим спецификациям файловых систем, называют системой управления файлами. Назначением системы управления файлами является предоставление наиболее удобного доступа к данным, организованным, как файлы. То есть, вместо низкоуровневого доступа к данным с указанием конкретных физических адресов, используется логический доступ с указанием имени файла. Как правило, все современные ОС имеют возможность работать с несколькими файловыми системами, либо с одной из нескольких, либо сразу с несколькими одновременно. Пользователь может образно представить себе жесткий диск как блокнот в клеточку. Одна клеточка на странице – это один кластер. Файловая система – это содержание блокнота, а файл – слово.

**Файловая система FAT**

Кластер — определение минимального размера данных на диске.

Файловая система FAT представляет собой простую файловую систему, разработанную для небольших дисков и простых структур каталогов. Название этой файловой системы происходит от метода, применяемого для организации файлов, - таблица размещения файлов. Использовалась в качестве основной файловой системы в операционных системах DOS и Microsoft Windows. Существует четыре версии FAT — FAT12, FAT16, FAT32 и exFAT. Они отличаются количеством бит, отведённых для хранения номера кластера. FAT12 применяется в основном для дискет, FAT16 — для дисков малого объёма, а новая exFAT преимущественно для флэш-накопителей. Структура файловой системы FAT





**Загрузочный сектор**- Он необходим для начальной загрузки компьютера. Так же в нем располагается информация о параметрах данного раздела.

**Таблица размещения файлов**- Вся область данных диска разделена на кластеры – блоки, размер которых задается при форматировании диска. Каждый файл и каталог занимает один или несколько кластеров. Таким образом, образуются цепочки кластеров. В таблице размещения файлов каждый кластер помечается специальным образом. Размер метки в битах для каждого кластера указывается в названии файловой системы. Т.е. для файловой системы FAT16 размер метки будет равен 16-ти байтам, для FAT32 – 32-м и т.д.

Всего существует три типа меток для кластеров:

**Свободный кластер** – кластер, в который будут записываться новые файлы и каталоги.

**Занятый кластер** – в метке указывается следующий кластер в цепочке. Если цепочка кластеров заканчивается, то кластер помечается особой меткой.

**BAD-блок** – кластер с ошибками доступа. Помечается при форматировании диска, что бы исключить в последующем доступ к нему.

Повреждение таблицы размещения файлов полностью уничтожает структуру файловой системы, поэтому на диске всегда хранится две копии таблицы.

**Корневой каталог**- Область диска, в котором располагается информация о корневом каталоге. Размер ее ограничен, поэтому в корневом каталоге диска может находиться не более 512-ти файлов и подкаталогов.

**Область данных**-Оставшаяся часть раздела, на которой размещается содержимое файлов и каталогов.





**Файловая система NTFS**

Файловая система NTFS разрабатывалась Microsoft как основная файловая система для серверных версий операционных систем Windows. В настоящее время NTFS рассматривается в качестве предпочтительной файловой системы как для серверных, так и для клиентских версий Windows. В NTFS используются 64 разрядные идентификаторы кластеров.

Перечислим некоторые возможности NTFS

Восстанавливаемость – способность файловой системы возвращаться к работоспособному состоянию после возникновения сбоя. Реализуется такая возможность, во первых, за счет поддержки атомарных транзакций, во вторых, за счет избыточности хранения информации. Атомарная транзакция – операция с файловой системой, приводящая к её изменению, которая либо полностью успешно выполняется, либо не выполняется вообще (т. е. в случае сбоя во время атомарной транзакции все изменения откатываются). Избыточность используется при хранении важнейших данных файловой системы, критически необходимых для её корректной работы;

безопасность (security) – защищенность файлов от несанкционированного доступа.

шифрование (encryption) – преобразование файла в зашифрованный код, который невозможно прочесть без ключа.

возможность использования для хранения информации нескольких дисков; данные с одного диска автоматически копируются на другие, обеспечивая тем самым повышенную надежность;

**50.** Для создания приложений (программ) работающих под управлением операционной системы Microsoft Windows используются разнообразные системные функции, которые предоставляют **интерфейс прикладного программирования**  (API). Windows API включает в себя совокупность функций, принадлежащих ядру или службам Windows, а также соглашения об использовании этих функций.

**51.** Разработчики Win32 API определяют собственные типы данных,

чтобы в некоторой степени изолировать себя от языка C/C++. На самом деле все они определены посредством директив typedef или #define в заголовочных файлах Win32 API.

Пример CHAR=char= символ АНСИ

FLOAT, DOUBLE, DWORD(целое 32 разрядное число без знака), LPCSTR(АНСИ строка).

**52**. Если по какой-либо причине функция Win32 API не может выполнить свою работу, она возвращает значение, свидетельствующее о том, что произошла ошибка.

Типы данных:

VOID- Функция почти всегда выполняется успешно.

BOOl(EAN)- Если вызов функции завершается неудачно, возвращается значение FALSE;

HANDLE- Если вызов функции завершается успешно, то возвращается дескриптор объекта, которым можно манипулировать; в случае ошибки – NULL или INVALID\_HANDLE\_VALUE (все зависит от конкретной функции).

PVOID-Если вызов функции завершается успешно, то возвращается адрес блока данных в памяти; в случае ошибки – NULL

LONG(DWORD) -Если вызов функции завершается неудачно, то обычно возвращается 0 или -1

Узнать какая именно это ошибка можно, вызвав функцию GetLastError, которая вернет код последней ошибки в текущем потоке: DWORD GetLastError(); Если нет ошибки вернет код ERROR\_SUCCESS.

**53.** Компилятор Visual C++ поддерживает, как стандартные 8-битные символы ANSI, так и 16-битные символы Unicode (UTF-16). Для представления ANSI-символов используется тип char, а для Unicode-символов – wchar\_t:

В заголовочном файле WinNT.h, определены следующие типы данных для

работы с символами, указателями на символы и строками:

typedef char CHAR; // 8-битный ANSI-символ

typedef CHAR \*LPSTR, \*PSTR; // ANSI-строка

typedef WCHAR \*LPWSTR, \*PWSTR; // Unicode-строка

**Любая функция,** изменяющая строку, несет потенциальную угрозу безопасности: если результирующая строка больше, чем предназначенный для нее буфер, содержимое памяти будет испорчено.

Поэтому для каждой из существующих строковых функций, таких как \_tcscpy, в библиотеку C была добавлена новая безопасная версия, которая включает имя старой функции и суффикс \_s . Теперь если функции передается в качестве параметра буфер, доступный для записи, то одновременно необходимо передать и его размер. Помимо безопасных строковых функций (функций с суффиксом \_s в именах) библиотека C поддерживает ряд новых безопасных функций для манипулирования со строками, определенных в заголовочном файле StrSafe.h. Пример StringCchCopy(принимают в виде параметра размер, выраженный в символах,), StringCbCopy(принимают в виде параметра размер, выраженный в байтах).

Функция MultiByteToWideChar преобразует мультибайтовые символы строки в UTF-16.

**54.**

* Для получения Имя компьютера: **GetComputerName и GetComputerNameEx.**

Для того чтобы изменить имя локального компьютера используются функции **SetComputerName и SetComputerNameEx.**

* Для получения имени пользователя в текущем сеансе используются функции **GetUserName и GetUserNameEx**.
* **SHGetFolderPath**- получения пути к различным системным каталогам.
* Для получения информации о версии Windows используется функция **GetVersionEx**: Параметр этой функции указатель на структуру OSVERSIONINFO([хранится размер струкутры в байтах, идентификатор ОС,идентификатор версии, идентификатор сборки, платформа, доп инфа об ОС)
* В Win32 API имеется функция **GetSystemMetrics**, позволяющая получить значения различных системных метрик.
* Для получения или изменения системных параметров используется функция **SystemParametersInfo**:
* Чтобы получить системную дату и время следует использовать функцию **GetSystemTime**, а чтобы извлечь системную дату и время по местному времени – **GetLocalTime**

**55.** Консольное приложение (console application) представляет собой программу, использующую интерфейс командной строки. Консольные приложения создаются, как правило, в тех случаях, когда требуется создать простейшую программу, обладающую самым простым интерфейсом. Информация консольному приложению передается в аргументах командной строки и через стандартный поток ввода. Для вывода информации консольное приложение использует стандартный поток вывода.

Главная функция –main. Точка входа в консольное приложение MainCRTStartup.

**56.** Вместо того чтобы каждый раз реализовывать одни и те же функции в каждом создаваемом приложении, можно создать библиотеку, реализующую эти функции, и затем использовать ее в приложениях. **Статической библиотекой (static library)** называется файл с раширением .lib . Компоновщик копирует данные, функции, классы и методы классов, содержащиеся в статической библиотеке, в исполнимый файл (.exe ) приложения при его создании. После этого содержимое статической библиотеки становиться частью приложения. Такой подход называется статическим подключением (static linking). Основным недостатком такого подхода является то, что при внесении изменений в библиотеку придется пересоздать все зависимые от нее файлы.

**Динамически подключаемой библиотекой** (dynamic link library, DLL) является библиотека, содержащая данные, функции, классы и методы классов, которые не копируются в исполнимый файл при создании приложения, а загружаются во время его запуска или в процессе выполнения. Подход, при котором библиотека DLL загружается в уже исполняемое приложение, называется динамическим подключением (dynamic linking). При внесении изменений в библиотеку DLL нет

необходимости пересоздавать все зависимые от нее файлы.В общем случае, файл, являющийся динамически подключаемой библиотекой, имеет расширение . dll.

**57. Окно в Windows** – это прямоугольная область экрана, где приложение отображает выводимую и принимает вводимую информацию от пользователя.

**Главное окно**- служит основным окном программы.

**Диалоговые окна**, или окна диалога (dialog box), – это всплывающие окна, которые используются для получения от пользователя дополнительной информации, а также вывода результатов работы приложения.

**Элементами управления** являются дочерние окна , предназначенные для отображения и редактирования различной информации, а также для выполнения пользователем определенных действий.

**Оконный класс** (window class) – это структура, определяющая основные характеристики окна. К ним относятся стиль класса и связанные с ним ресурсы, такие как меню (menu), пиктограмма (icon), курсор (cursor) и кисть (brush) для закрашивания фона окна. Кроме того, эта структура содержит адрес функции, предназначенной для обработки сообщений любого окна данного класса. Для создания нового оконного класса необходимо заполнить структуру WNDCLASSEX, а затем передать адрес этой структуры в виде аргумента функции RegisterClassEx.

**58.** Оконное сообщение- сообщение, которое адресовано окнам приложения. Все сообщения, которые помещает ОС, она помещает в очередь сообщения.

Цикл обработки сообщений-извлекает сообщения из очереди, и передает полученное сообщение оконной процедуре того окна, которому оно было адресовано.

Обработкой оконных сообщений в приложении занимается специальная функция, называемая оконной процедурой.

**59.** Приложение Win32 (Win32 application) основано на графическом интерфейсе пользователя. Такие приложения создают окна, имеют меню, взаимодействуют с пользователем через диалоговые окна и т.п.

Главная функция-WinMain, точка входа-WinMainCRTStartup

**60.** Объект ядра (kernel object) представляет собой структуру данных, созданную ядром операционной системы, и в которой содержится информация об объекте операционной системы (дескриптор безопасности, счетчик использования и др.). Windows позволяет создавать, открывать и выполнять различные операции с несколькими типами объектов ядра, к которым в том числе относятся процессы и потоки,job, mutant

В Win32 API имеется множество функций, создающих объекты ядра. Какую именно из этих функций следует использовать, зависит от типа объекта ядра.

Закрытие объекта ядра-BOOL CloseHandle(HANDLE hObject);

**61.** Наследование применяется в случае, когда процесс решает, породив дочерний процесс, передать ему по наследству доступ к своим объектам ядра.

Во-первых, для этого нужно, чтобы дескрипторы созданных объектов ядра были наследуемыми.

Во-вторых, процесс должен порождать дочерний процесс с помощью функции CreateProcess, передав в параметре bInheritHandles значение TRUE.

Помимо копирования записей из таблицы дескрипторов, также увеличиваются на 1 значения счетчиков соответствующих объектов ядра, поскольку эти объекты используются еще одним процессом.

**62.** Операция дублирования дескриптора объекта ядра заключается в том, что берется соответствующая запись в таблице дескрипторов одного процесса и создается ее копия в таблице другого.

Используется функция DuplicateHandle.

**63**. Многие объекты ядра (но не все) допускают именование. Имя объекта ядра указывается при вызове функции, создающей этот объект.

При создании именованного объекта ядра, сначала проверяется, существует ли объект с таким же именем. Если такого объекта не существует, создается именованный объект ядра и добавляется соответствующая запись в таблицу дескрипторов вызывающего процесса.

Если объект с таким именем уже существует, выполняется проверка типа объекта и прав доступа вызывающего процесса к этому объекту. Если типы двух объектов с одинаковыми именами не совпадают или вызывающий процесс не имеет прав доступа к существующему объекту, вызов функции, создающей этот объект, завершается ошибкой. Если же все в порядке, счетчик объекта ядра увеличивается на 1 и создается новая запись в таблице дескрипторов вызывающего процесса, а функция, создающая объект ядра, возвращает дескриптор существующего объекта.

64. Процесс-представляет собой экземпляр исполняемого приложения обладающий не зависимим виртуальным адресным пространством, в котором могут размещаться данные недоступные другим процессам.

Новый процесс создается с помощью функции CreateProcess

Процесс завершает работу в следующих случаях:

• Главная функция (WinMain, wWinMain, main или wmain) приложения завершила свою работу и вернула управление.

• Один из потоков процесса вызвал функцию ExitProcess. Функция ExitProcess завершает процесс, указав в качестве параметра uExitCode код завершения

• Поток другого процесса вызвал функцию TerminateProcess.

• Все потоки процесса завершили свою работу.

Приоритеты процесса (Реального времени, высокий. Выше среднего, средний, ниже среднего, низкий)

**65.** Задание (job) – это объект ядра, который позволяет группировать процессы и накладывать на них различные ограничения. CreateJobObject –функция создания задания. Один и тот же процесс может быть включен только в одно задание. Как только процесс был включен в какое-нибудь задание, его уже нельзя исключить из него или включить в другое задание. Если процесс был включен в задание, все порождаемые им процессы автоматически включаются в тоже задание. Но можно исправить это с помощью наложения ограничения.

**66.** Потоки-исполняют код и манипулируют данными в адресном пространстве процесса. Любой поток состоит из двух компонентов объекта ядра и стека потока.

\_beginthreadex и \_endthreadex - функция создания потока и его завершение.

1)Функция потока завершила работу и вернула управление.

2)Поток вызвал функцию ExitThread.

3)Один из потоков процесса вызвал функцию TerminateThread.

4)Был завершен процесс, содержащий данный поток.

**67.**  Приоритеты процесса. Чем выше приоритет, тем больше процессорного времени ему выделяется. Приоритеты процесса (Реального времени, высокий. Выше среднего, средний, ниже среднего, низкий)

SetPriorityClass, GetPriorityClacc –функции для работы с приоритетом процесса.

Относительный приоритет потока(критичный по времени, максимальный, выше среднего, средний, ниже среднего, минимальный, приостанавливающий).

GetThreadPriority; SetThreadPriority.

**68.** синхронизацию потоков применяют в следующих случаях:

1. когда несколько потоков должны совместно использовать некоторые общие ресурсы;
2. когда нужно уведомлять другие потоки о завершении или о начале каких-либо операций.

Существуют два подхода к синхронизации потоков: синхронизация в пользовательском режиме и синхронизация с использованием объектов ядра.

Механизмы синхронизации в пользовательском режиме обеспечивают высокое быстродействие, но у них есть ряд ограничений.

**Функции взаимоблокировки**

В Win32 API имеется набор функций под общим названием функции взаимоблокировки . Разграничение доступа к разделяемым ресурсам с помощью функций взаимоблокировки возможно благодаря тому, что каждая такая функция гарантированно выполняется, как атомарная операция (atomic operation), которая не может быть прервана в ходе своего выполнения

**критические секции**

Область программного кода, в которой требуется монопольный доступ к разделяемым ресурсам, называется критической секцией. Для использования критической секции нужно создать переменную типа CRITICAL\_SECTION и проинициализировать ее с помощью функции InitializeCriticalSection. После этого любой участок программного кода, работающий с разделяемым ресурсом, нужно заключить в вызовы функций EnterCriticalSection и LeaveCriticalSection Когда критическая секция более не нужна, ее следует удалить вызовом функции DeleteCriticalSection.

**тонкая блокировка чтения и записи**

Тонкая блокировка позволяет нескольким «читающим» потокам одновременно обращаться к разделяемому ресурсу, поскольку чтение не грозит повреждением такого ресурса. Однако для «записывающих» потоков предоставляется монопольный доступ к разделяемому ресурсу. Чтобы использовать механизм тонкой блокировки чтения и записи, нужно создать переменную типа SRWLOCK и проинициализировать ее с помощью функции InitializeSRWLock. Теперь чтобы получить монопольный доступ к ресурсу, «записывающий» поток должен вызвать функцию AcquireSRWLockExclusive. После модификации ресурса снять блокировку можно вызовом функции ReleaseSRWLockExclusive. В случае «читающего» потока все аналогично, только используется другая пара функций – AcquireSRWLockShared и ReleaseSRWLockShared.

69. **МЬЮТЕКС**

**Мьютекс** – объект ядра, гарантирующий потокам взаимное исключение доступа к разделяемому ресурсу. В каждом объекте ядра «мьютекс» есть переменная, в которой запоминается идентификатор захватившего его потока. Этот идентификатор может быть равен нулю, что означает – мьютекс не захвачен ни одним из потоков. Если мьютекс не захвачен, он находится в свободном состоянии, иначе – в занятом состоянии. Для создания объектов ядра «мьютекс» предназначены две функции – CreateMutex и CreateMutexEx. Мьютекс можно перевести в свободное состояние с помощью функции ReleaseMutex.

**СЕМАФОР**

Семафор – объект ядра, предназначенный для того, чтобы ограничить максимальное число потоков, одновременно работающих с неким разделяемым ресурсом.

В каждом объекте ядра «семафор» есть специальный счетчик. Когда значение этого счетчика больше нуля, семафор находится в свободном состоянии, иначе – в занятом состоянии.

Для создания объектов ядра «семафор» используют одну из двух функций – CreateSemaphore и CreateSemaphoreEx: (параметры безопасности; начальное и максимальное значение счетчика семафора;имя;0)

Открыть уже созданный объект ядра «семафор» можно с помощью функции OpenSemaphore. Увеличить значение счетчика семафора можно с помощью функции ReleaseSemaphore:

**СОБЫТИЯ**

Событие (event) – объект ядра, используемый потоком для того, чтобы сигнализировать другим потокам о наступлении какого-либо события.

Объекты ядра «события» бывают двух типов: сбрасываемые вручную события и автоматически сбрасываемые события.

Для создания объектов ядра «событие» применяют две функции – CreateEvent и CreateEventEx. Создав событие, можно управлять его состоянием с помощью трех функций – SetEvent, ResetEvent и PulseEvent (переводит в свободное состояние, в занятое, и то и другое поочередно).