1. Операционная система (ОС): архитектура компьютера с точки зрения программиста, определение, классификация, операционные системы реального времени, программные интерфейсы, BIOS, драйверы, утилиты, ядро, основные объекты ядра, POSIX, фреймворк, программные и аппаратные прерывания, системные вызовы.

Архитектура включает в себя набор команд, типы данных, операции ввода-вывода и другие характеристики. Включает в себя все, что нужно знать программисту для создания корректно работающих программ.

Операционная система - комплекс взаимосвязанных программ, предназначенных для управления ресурсами компьютера и организации взаимодействия с пользователем. Классификацируются на:

- **для мейнфреймов**: много одновременных заданий, пакетная обработка данных;
- **серверные:** линейка Windows Server 2008/2012/2016, Solaris, Linux Server;
- **сетевые**: Novel NetWare
- **персональные**: Windows, Mac OS, Ubuntu;
- **мобильные**: iOS, Android;
- **встроенные**: Embedded Linux, QNX;
- **реального времени**: мягкое и жесткое реальное время, FreeRTOS;
- **смарт-карт:** MULTOS

Операционная система реального времени — операционная система, реагирующая на внешние события в определенный промежуток времени. Успешность работы любой программы зависит не только от ее логической правильности, но и от времени, за которое она получила результат.
Это операционная система, предназначенная для обслуживания приложений реального времени, которые обрабатывают данные по мере их поступления, как правило, без задержек в буфере.

Programs Interface: интерфейс между приложениями и ядром OS (OS API; OS Framework, .NET Framework, .NET CORE Framework; Windows API)

Basic Input/Output System(BIOS) – набор микропрограмм, реализующих API для работы с аппаратурой компьютера и подключёнными к нему устройствами:

- тестирование оборудования (при включении, POST -Power-On Self-Test);
- поиск и подключение устройств типа plug and play;
- первоначальная загрузка операционной системы;
- API для работы с оборудованием (на этапе загрузки ОС, потом используются драйверы);
- активация в offline (SLIC — software licensing description table);
- пользовательский интерфейс для настройки оборудования (устройство для загрузки, частота, лимиты, перечень устройств и т.п.).

Драйверы – программы, предназначенные для унифицированного доступа программного обеспечения к аппаратуре. Для его разработки исспользуется DDK

Утили́та — вспомогательная компьютерная программа в составе общего программного обеспечения для выполнения специализированных типовых задач.

- **Независимые утилиты** - не требующие для работы операционной системы;
- **Системные утилиты** - входящие в поставку ОС и требующие её наличия

Ядро́ — центральная часть операционной системы, обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешнее устройство ввода и вывода информации.

Объекты ядра Windows:

- Процесс, поток.
- Файловый объект, открытый файл.
- Проекция файла на память.
- Событие, Семафор, Мьютекс.
- Почтовый ящик, канал, сокет и т.д

POSIX (переносимый интерфейс операционных систем) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API), библиотеку языка C и набор приложений и их интерфейсов.

Фре́ймворк— программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

Программная платфо́рма — это среда выполнения, в которой должен выполняться фрагмент программного обеспечения или объектный модуль с учётом накладываемых этой средой ограничений и предоставляемых возможностей.

Прерывание — сигнал от программного или аппаратного обеспечения, сообщающий процессору о наступлении какого-либо события, требующего немедленного внимания.

- Программные - вызываются искусственно с помощью соответствующей команды из программы, предназначены для выполнения некоторых действий операционной системы, являются синхронными (инициируются исполнением специальной инструкции в коде программы. Программные прерывания, как правило, используются для обращения к функциям встроенного программного обеспечения (firmware), драйверов и операционной системы);
- Аппаратные прерывания возникают как реакция микропроцессора на физический сигнал от некоторого устройства (клавиатура, системные часы, клавиатура, жесткий диск и т.д.), по времени возникновения эти прерывания асинхронны, т.е. происходят в случайные моменты времени.

Системный вызов - механизм вызова прикладной программой функции ядра OS (отключается защита памяти, возможно выполнение полного набора команд процессора). Системный вызов осуществляется с помощью программного прерывания.

2. Процесс ОС: определение, ресурсы процесса, контекст процесса, идентификатор процесса, IPC, стандартные потоки ввода/вывода, системные процесса, создание процесса в Windows, HANDEL Windows-процесса, дерево процессов, Idle-процесс, назначение процессоров процессам.

Процесс OS – единица работы OS (объект ядра OS + контекст процесса). основные свойства процесса:

- процессу соответствует исполняемый программный файл;
- у процесса есть PID;
- у процесса есть Parent PID;
- в Windows: HANDLE – идентификатор объекта OS;
- в OS есть процесс инициализации (родитель для всех);
- запуск и управление процессом осуществляется с помощью системных вызовов;
- процессы изолированы друг от друга;
- процессу выделяется линейное адресное пространство (размер зависит от разрядности), сегменты: code, static, data, heap, stack;
- **контекст процесса** – данные, которые сохраняются при переключении процессов и предназначенные для продолжения работы;
- процессу автоматически доступны три потока: ввода, вывода, вывод ошибок.
- при запуске OS некоторые процессы загружаются и стартуют автоматически, как правило используются для внутреннего назначения;
- в составе OS есть таблица, содержащая объекты ядра процессов (состояние, приоритет, указатели на другие объекты); есть средства OS позволяющие ее просматривать;

Ресурсы: регистры, открытые файлы, родительский процесс, перечень связанных (дочерних) процессов, реальные страницы памяти, виртуальное адресное пространство, маркеры доступа (безопасность)

Контекст процесса: адресное пространство, содержимое регистров (общего назначения, счетчик команд, состояния процессора, вершина стека, …), объекты ядра OS (объекты процессов, потоков, безопасности, файлов и пр.), стек ядра (для ее этого процесса). Переключение контекстов. В ядре специальный стек для переключения контекстов

Process IDentifier, PID — уникальный номер (идентификатор) процесса в многозадачной операционной системе (ОС). В ОС семейства Windows PID хранится в переменной целочисленного типа.

inter-process communication, IPC(механизм межпроцессного взаимодействия) — обмен данными между потоками одного или разных процессов. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.

Стандартные потоки ввода/вывода процесса - потоки имеющие зарезервированные номера - дескрипторы (номера): поток ввода (0), поток вывода (1), поток вывода ошибок (2). Как правило, эти дескрипторы открыты уже в момент запуска задачи.

Системные процессы: процессы, запускаемые автоматически при запуске OS (**сервис**). Функция **CreateProcess (при вызове нужно имя exe)**, которая создает новый процесс с единственным потоком. Функция возвращает два отдельных дескриптора, по одному для процесса и потока, передавая их в структуре. Глобальные идентификаторы остаются уникальными для данного объекта на протяжении всего времени его существования и во всех процессах, тогда дескрипторов процесса может быть несколько и каждый из которых может характеризоваться собственным набором атрибутов, например определенными разрешениями доступа.

Новый процесс получает информацию об окружении, рабочем каталоге и иную информацию в результате вызова функции CreateProcess. По завершении этого вызова любые изменения характеристик родительского процесса никак не отразятся на дочернем процессе. Оба процесса полностью независимы друг от друга.

HANDLE — условно, адрес, по которому хранится информация по процессу, например такая как: время запуска, имя файла, ассоциированного с процессом, и даже тот же самый PID. Будучи однажды получен, HANDLE требует закрытия через CloseHandle().

Процесс-потомок наследует следующие признаки родителя:
- сегменты кода, данных и стека программы;
- таблицу файлов, в которой находятся состояния флагов дескрипторов файла, указывающие, читается ли файл или пишется. Кроме того, в таблице файлов содержится текущая позиция указателя записи-чтения;
- рабочий и корневой каталоги;
- реальный и эффективный номер пользователя и номер группы;
- приоритеты процесса (администратор может изменить их через nice);
- контрольный терминал;
- маску сигналов;
- ограничения по ресурсам;
- сведения о среде выполнения;
- разделяемые сегменты памяти.

Потомок не наследует от родителя следующих признаков:
- идентификатора процесса (PID, PPID);
- израсходованного времени ЦП (оно обнуляется);
- сигналов процесса-родителя, требующих ответа;
- блокированных файлов (record locking).

Бездействие системы (англ. system **idle** process) — процесс, выполняемый процессором в пространстве ядра операционной системы в случае, если нет других процессов, которые процессор мог бы выполнять.

3. Поток ОС: определение, отличие от процесса, контекст потока, идентификатор потока, основной поток процесса, потоковая функция, понятие многопоточности, понятие потокобезовасность, диспетчеризация потоков, модель 12 состояний потока, создание потока в Windows, управление потоками в Windows, HANDEL Windows-потока, особенности создания потоков и работы с ними в Linux, приостановка работы потока, реентерабельность потока, дерево потоков, назначение процессоров понятие fiber.

Поток - объект ядра операционной системы, которому OS выделяет процессорное время. Наименьшая единица работы ядра OS. **поток (управления) OS** – последовательность инструкций, выполняемых процессором в выделенные OS интервалы времени.

Процессы и потоки связаны друг с другом, но при этом имеют существенные различия. **Процесс** — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). Каждый процесс выполняется в отдельном адресном пространстве. Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие (например файлы).
Поток использует то же самое пространства стека, что и процесс, а множество потоков совместно используют данные своих состояний. Как правило, каждый поток может работать с одной и той же областью памяти. У каждого потока есть собственные регистры и собственный стек, но другие потоки могут их использовать. Это определенный способ выполнения процесса. Когда один поток изменяет ресурс процесса, это изменение сразу же становится видно другим потокам этого процесса.
контекст потока:

- программный код;
- набор регистров;
- стек памяти;
- стек ядра операционной системы;
- маркер доступа.

Идентификатор - уникально идентифицирует поток повсюду в системе. Допустимы со времени создания потока, и до тех пор, пока поток не завершит работу. При создании процесса, автоматически создается основной (main) поток (выполняется функция ядра, создающая поток), который является потоком выполняющихся по очереди команд центрального процессора. При необходимости главный поток может создавать другие потоки, пользуясь для этого программным интерфейсом операционной системы. В простейшем – один поток процесса.

Многопото́чность — свойство платформы 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.

Потоковая безопасность — это концепция программирования, применимая к многопоточным программам. Код потокобезопасен, если он функционирует исправно

при использовании его из нескольких потоков одновременно. В частности, он должен обеспечивать правильный доступ нескольких потоков к разделяемым данным.

Диспетчеризация заключается в реализации переключения процессора с одного потока на другой. Прежде чем прервать выполнение потока, ОС запоминает его контекст с тем, чтобы впоследствии использовать эту информацию для последующего возобновления выполнения данного потока.

Диспетчеризация состоит из трех шагов:

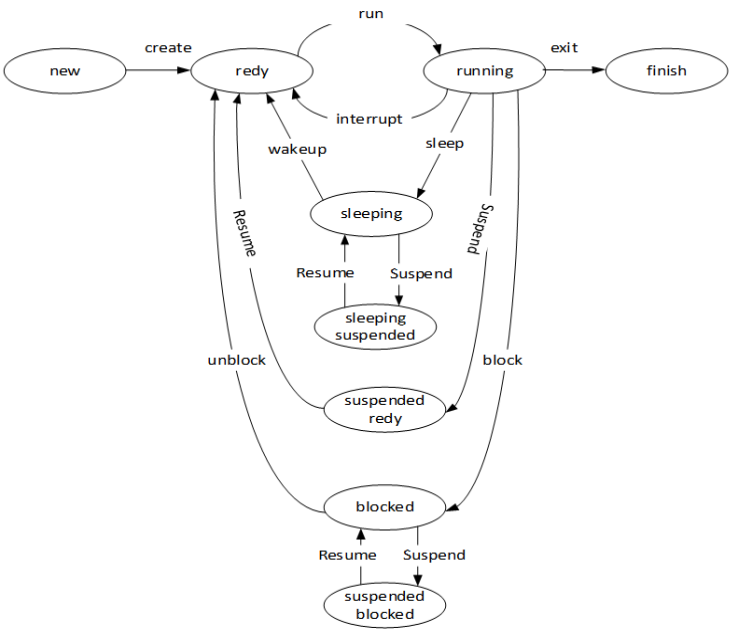
1. Сохранение контекста потока, который требуется сменить.
2. Загрузка контекста нового потока.
3. Запуск нового потока на выполнение.

Поток может находиться в одном из трех состояний:

1. Выполнение: процессор выполняет команды данного потока.
2. Ожидание: поток заблокирован по своим внутренним причинам
3. Готовность: поток готов, но процессор занят выполнением другого потока.

модель 12 состояний

1. Create (создать поток) и Exit (выход из потока)
2. Run (запустить поток) и Interrupt (прерывание по окончанию кванта)
3. Block (заблокировать до наступления события) и Unblock (событие наступило)
4. Suspend (приостановить поток) и Resume (возобновить поток)
5. Sleep (остановить поток на заданное время) и Wakeup (возобновить работу).



Функция **CreateThread** создает для процесса поток (определит начальный адрес (название функции) кода, с которого новый поток должен исполняться). Эта функция получает единственный параметр и возвращает значение типа DWORD. Процесс может иметь одновременно несколько потоков, выполняющих ту же самую функцию.

CREATE_SUSPENDED приведет к запуску потока в приостановленном состоянии, из которого поток может быть переведен в состояние готовности путем вызова функции ResumeThread.

По завершении выполнения потока память, занимаемая ее стеком, освобождается. В случае если поток был создан в библиотеке DLL, будет вызвана соответствующая точка входа DllMain с указанием флага **DLL_THREAD_DETACH** в качестве "причины" этого вызова.

Любой поток процесса может сам завершить свое выполнение, вызвав функцию **ExitThread**, однако более обычным способом самостоятельного завершения потока является возврат из функции потока с использованием кода завершения в качестве возвращаемого значения. Когда завершается выполнение последнего потока, завершается и выполнение самого процесса.

Выполнение потока также может быть завершено другим потоком с помощью функции **TerminateThread**, однако освобождения ресурсов потока при этом не происходит, обработчики завершения не выполняются и уведомления библиотекам DLL не посылаются.

Поток, выполнение которого было завершено, продолжает существовать до тех пор, пока посредством функции **CloseHandle** не будет закрыт ее последний дескриптор. Любой другой поток, возможно и такой, который ожидает завершения другого потока, может получить код завершения потока.

Поток может увеличивать или уменьшать значение счетчика приостановок другого потока с помощью функций **SuspendThread** и **Resume-Thread**.

Тип **HANDLE** обозначает 32-разрядное целое, используемое в качестве дескриптора (описателя). Есть несколько похожих типов данных, но все они имеют ту же длину, что и HANDLE, и начинаются с литеры H.

Дескриптор – это просто число, определяющее некоторый ресурс. Субъект системы безопасности, проверяющий доступ. Дескриптор может быть продублирован функцией DuplicateHandle, которая разрешает Вам создать дескриптор потока с подмножеством прав доступа. Дескриптор допустим до тех пор, пока не закрыт, даже после того, когда поток, который он представляет, завершит свою работу. Если у нас 4 процесса будет дескриптор, указывающий на поток, который указывает на процессы. У Lunux дескриптор указывает сразу на 4 процесса.

Реентерабельность кода (программы) – свойство одной копии программного кода работать в нескольких потоках одновременно. Реентерабельный код всегда потокобезопасен. Реентерабельный код не использует статическую память и не изменяет сам себя, все данные сохраняются в динамической памяти.

Назначение процессора – это автоматическое выполнение программы. Другими словами, он является основным компонентом любого компьютера.

Fibers (Файберы, Фибра, волокна): Windows, Linux. механизм для ручного планирования выполнения кода в рамках потока.

4. Диспетчеризация потоков: циклическое планирование, приоритетное планирование, кооперативное планирование, понятие приоритета потока и процесса, базовый приоритет потока в Windows, особенности диспетчеризации потоков в системах реального времени, назначение приоритета процессу и потоку в Windows, динамическое (автоматическое) изменение приоритета потоков, распределение процессоров потокам.

Диспетчеризация – это переключение процессора с одного потока на другой.

Циклическое планирование. Каждому процессу предоставляется квант времени процессора. Когда квант заканчивается, процесс переводится планировщиком в конец очереди, а управление передается следующему за ним процессу.

Преимущества: простота и справедливость (как в очереди покупателей, каждому только по килограмму).
Недостатки:

- слишком малый квант времени приводит к частому переключению процессов и снижению производительности;
- слишком большой квант может привести к увеличению времени ответа на интерактивный запрос.

Приоритетное планирование. Каждому процессу присваивается приоритет, и управление передается процессу с самым высоким приоритетом. Обычно процессы объединяют по приоритетам в группы, и применяют приоритетное планирование среди групп, а внутри группы используют циклическое планирование.

Гарантированное планирование. ОС гарантирует существующим потокам, что они получат гарантированную справедливую часть процессорного времени.

КООПЕРАТИВНЫЕ (невытесняющие) алгоритмы основаны на том, что активному потоку позволяется выполняться, пока он сам, по собственной инициативе, не отдаст управление операционной системе.

Вытесняющие алгоритмы. решение о переключении процессора с выполнения одного потока на выполнение другого потока принимается операционной системой.

Приоритет потока — величина, складывающаяся из двух составных частей: приоритета породившего поток процесса и собственно приоритета потока. Когда поток создается, ему назначается приоритет, соответствующий приоритету породившего его процесса. В свою очередь, процессы могут иметь следующие классы приоритетов.

- Real time; 24
- Normal; 8
- High; 13
- Below normal; 6
- Above normal; 10
- Idle.

Приоритет процесса - число, ориентируясь на значение, которого планировщик процессов может выдавать процессу больше или меньше процессорного времени.

Базовый приоритет потока – сочетание класса приоритета процесса и приоритета потока, изменяется в пределах [1,31], по умолчанию – 8, приоритеты возрастающие

процессы

потоки: Класс приоритета / относительный приоритет	Real-Time	High	Above-Normal	Normal	Below-Normal	Idle
Time Critical (+насыщение)	31	15	15	15	15	15
Highest (+2)	26	15	12	10	8	6
Above Normal (+1)	25	14	11	9	7	5
Normal (0)	**24**	**13**	**10**	**8**	**6**	**4**
Below Normal (-1)	23	12	9	7	5	3
Lowest (-2)	22	11	8	6	4	2
Idle (-насыщение)	16	1	1	1	1	1

Особенности диспетчеризации потоков в системах реального времени:

- Любая система реального времени должна реагировать на сигналы управляемого объекта в течение заданных временных ограничений.
- Необходимость тщательного планирования работ облегчается тем, что в системах реального времени весь набор выполняемых задач известен заранее.
- Кроме того, часто в системе имеется информация о временах выполнения задач, моментах активизации, предельных допустимых сроках ожидания ответа и т. д.

Назначение потоку приоритета происходит в два этапа.

- каждому процессу в момент создания присваивается класс приоритета. GetPriorityClass и SetPriorityClass.
- При создании уровень приоритета потока по умолчанию устанавливается равным уровню класса приоритета процесса, создавшего данный поток. SetThreadPriority, чтобы изменить приоритет потока.

!! класс приоритета процесса может быть задан при создании процесса, может быть получен или изменен с помощью системного вызова;

Динамическое (автоматическое) изменение приоритета потоков - Иногда система изменяет уровень приоритета потока. Это происходит в ответ на некоторые события, связанные с вводом выводом. Система повышает приоритет только тех потоков, базовый уровень которых находится в пределах 1-15(область динамического приоритета). Система не допускает динамического повышения приоритета потока до уровней реального времени (более 15), поскольку потоки с такими уровнями обслуживают системные функции, это ограничение не дает приложению нарушить работу операционной системы.

Распределение процессоров потокам - при работе на многоядерных процессорах, операционная система переносит поток с более загруженных на менее загруженные ядра, что обеспечивает равномерную загрузку всех ядер системы.

!! OS может динамически изменять приоритет потока, этот режим работы может быть включен/отключен с помощью системного вызова

5. Управление памятью: упрощенные схемы чтения и записи данных в оперативную память, адресное пространство, виртуальная память, страничный свопинг, страничная память, таблица страниц, Memory Management Unit, Translation Lookaside Buffer, трансляция виртуальных адресов в реальные, смена контекста процесса, инвертированные таблицы, понятие рабочего набора, алгоритм замещения страниц LRU, структура адресного пространства Windows/32, распределение памяти для стека приложения в Windows, распределение heap-памяти в Windows, выделение виртуальной и освобождение памяти в Windows, управление рабочим множеством страниц в Windows, запрет свопинга в Windows, распределение и управление Heap-памятью.

Управление памятью: упрощенные схемы чтения и записи данных в оперативную память.
Упрощенная схема чтения данных:

1. Адресная шина ↓
2. Регистр адреса ↓
3. Оперативная память ↓
4. Регистр данных ↓
5. Шина данных ↓

упрощенная схема записи данных:

1. Адресная шина ↓
2. Регистр адреса ↓
3. Оперативная память
4. Регистр данных ↑
5. Шина данных↑

Адресное пространство - абстракция ядра OS (реальной памяти). Непрерывный диапазон адресов выделяемый OS процессу; у каждого процесса свое адресное пространство.
Виртуальная память – метод управления памятью процессора, предназначенный для выполнения программ, которым выделяется адресное пространство, превышающее доступный физический объем памяти компьютера.
Страничный свопинг - механизм OS обмена (вытеснения и загрузки) содержимым блоков оперативной физической памяти компьютера с устройством хранения данных с целью расширения адресуемого объема оперативной памяти компьютера. Механизм является аппаратно-программным.
Страничная память - реализации виртуальной памяти, при которой физическая память и адресное пространство разбивается на блоки (страницы), а также осуществляется страничный свопинг.
Таблица страниц - структура данных, используемая системой виртуальной памяти в операционной системе компьютера для хранения сопоставления между виртуальным адресом и физическим адресом.

Таблица страниц является ключевым компонентом преобразования виртуальных адресов, который необходим для доступа к данным в памяти.
Обращение к таблице страниц:

- при выборе процессором инструкции из памяти для исполнения;
- при выполнении инструкции, если в ней используется адрес памяти (регистр/регистр -нет);
- перегрузка контекста процессора.

многоуровневые таблицы страниц применяются для больших адресных пространств; позволяют не хранить информацию о страницах, не распределенных процессу;
Memory Management Unit (MMU диспетчер памяти) – аппаратное (программируемое) устройство, входящее в состав процессора и предназначенное для трансляции виртуальных адресов оперативной памяти в реальные.
Translation Lookaside Buffer (TLB буфер быстрого преобразования адреса) – ассоциативная память (параллельный поиск). Компонент MMU, предназначенный для вычисления реальных адресов, хранит 64 строки таблицы страниц, полностью таблица хранится во вторичной (диск) памяти без свопинга;
Трансляция виртуальных адресов в реальные - блок управления памятью.
Смена контекста процесса - процесс прекращения выполнения процессором одной задачи (процесса, потока, нити) с сохранением всей необходимой информации и состояния, необходимых для последующего продолжения с прерванного места, и восстановления, и загрузки состояния задачи, к выполнению которой переходит процессор.
Инвертированные таблицы - таблица для физических страниц; применение хэш-таблиц; специальный TLB для инвертированной таблицы;
Понятие рабочего набора — это набор страниц, которые физически находятся в памяти в любой момент времени
Алгоритм замещения страниц LRU - выбор страницы, которую можно заместить страницей с дискового устройства: «рабочее множество», LRU

- hiberfile.sys - файл для сохранения памяти в режиме «сон» (гибернация);
- pagefile.sys - файл подкачки;
- swapfile.sys - файл подкачки отдельных (предварительно скаченных из магазина приложений UWP) для быстрого применения (в случае надобности).

Распределение памяти для стека приложения в Windows - Начальный размер стека зависит от o/s. Начальный размер heap логически равен нулю, но имеет тенденцию расти почти сразу. стек и куча будут назначены сегментам, которые могут быть расширены. Однако многие микропроцессоры не имеют аппаратного обеспечения для отображения памяти, и размеры должны быть предварительно зарезервированы.

области адресного пространства (от младших к старшим адресам) и поясните их назначения:

- код- код программы
- static - автоматические переменные

- heap – фрагмент памяти адресного пространства (по умолчанию 1MB), предназначенный для динамического использования
- data - глобальные и статические переменные, которые программист создал(инициализировал)
- stack – область стека для потоков (по умолчанию 1MB).

распределение heap-памяти в Windows - Heap – область памяти адресного пространства, предназначенного для использования программной фрагментов динамически выделяемой памяти. По умолчанию – 1MB, из них 4K сразу забирает процесс. По мере new(malloc) размер HEAP прирастает. Память выделяется с учетом минимизации фрагментации.

выделение виртуальной и освобождение памяти в Windows - virtualAlloc VirtualFree

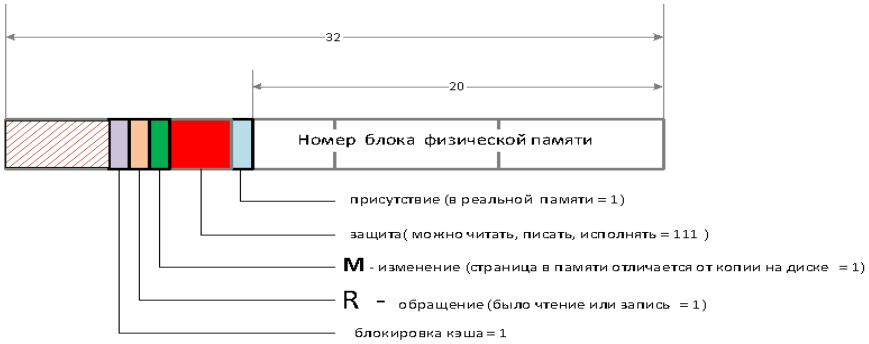
управление рабочим множеством страниц в Windows - SetProcessWorkingSEt - устанавливает минимальный и максимальный размеры рабочего комплекта памяти для заданного процесса.

запрет свопинга в Windows - файл, представляющий собой виртуальную память, которая позволяет одновременно выполняться большому количеству процессов, которые все сразу не смогли бы поместиться в физической памяти.

распределение и управление Heap-памятью

```
#include <Windows.h>
#include <iostream>
#include <iomanip>
#include <locale>

void sayHeapInfo(HANDLE pheap);
int main()
{

    HANDLE heap = HeapCreate(HEAP_NO_SERIALIZE|HEAP_ZERO_MEMORY, 2048,4096);
    sayHeapInfo(heap);

    int* x1 = (int*)HeapAlloc(heap, HEAP_NO_SERIALIZE | HEAP_ZERO_MEMORY, 500 * sizeof(int));
    sayHeapInfo(heap);

    int* x2 = (int*)HeapAlloc(heap, HEAP_NO_SERIALIZE | HEAP_ZERO_MEMORY, 220 * sizeof(int));
    sayHeapInfo(heap);

    HeapFree(heap, HEAP_NO_SERIALIZE, x2);
    sayHeapInfo(heap);

    HeapFree(heap, HEAP_NO_SERIALIZE, x1);
    sayHeapInfo(heap);

    HeapDestroy(heap);

    system("pause");
    return 0;
}
```

Информация, содержащаяся в строке таблицы страниц



Рабочее множество - количество памяти, требующееся процессу в заданный интервал времени.

Рабочий набор процесса — это набор страниц в виртуальном адресном пространстве процесса, которые в настоящее время находятся в физической памяти. Процесс уменьшает или опустошает рабочий набор, вызывая функцию SetProcessWorkingSetSize (Устанавливает минимальный и максимальный размеры рабочего набора для указанного процесса.)

Согласно модели рабочего множества, **процесс может находиться в ОЗУ** тогда и только тогда, когда множество всех его страниц, используемых в настоящее время могут находиться в ОЗУ. Модель работает по принципу «всё или ничего», то есть, если число нужных процессу страниц памяти растёт и в ОЗУ нет свободного места, то процесс выгружается из памяти целиком, чтобы освободить память для использования другими процессами.

Часто сильно загруженный компьютер может иметь столько процессов в очереди, что, если позволить им запуститься в один и тот же квант времени, то объём памяти, на который они будут ссылаться, превысит объём ОЗУ, вследствие чего возникнет пробуксовка виртуальной памяти.

VirtualLock и VirtualUnlock Блокирует указанную область виртуального адресного пространства процесса в физической памяти, гарантируя, что последующий доступ к области не приведет к ошибке страницы. Страницы, заблокированные процессом, остаются в физической памяти до тех пор, пока процесс не разблокирует их или не завершит работу. Эти страницы гарантированно не будут записаны в файл подкачки, пока они заблокированы. Максимальное количество страниц, которые может заблокировать процесс, равно количеству страниц в его минимальном рабочем наборе за вычетом небольших накладных расходов.

6. Синхронизация потоков: определение, взаимная блокировка потоков, принцип реализации процесса синхронизации, механизмы синхронизации потоков в Windows, атомарные операции в Windows.

При одновременном доступе нескольких процессов к какому-либо ресурсу возникает проблема синхронизации. Поток может быть остановлен в любой момент времени, при этом один из потоков не успел завершить модификацию ресурса, но был остановлен, а другой поток попытался обратиться к тому же ресурсу. В этот момент ресурс находится в несогласованном состоянии. Основой синхронизации потоков в Win32, является использование объектов синхронизации и функций ожидания. Объекты могут находиться в одном из двух состояний — **Signaled** или **Not Signaled**. Таким образом, поток, которому необходим эксклюзивный доступ к ресурсу, должен выставить какой-либо объект синхронизации в несигнальное состояние, а по окончании — сбросить его в сигнальное. Остальные потоки должны перед доступом к этому ресурсу вызвать функцию ожидания, которая позволит им дождаться освобождения ресурса.

Синхронизация - механизм упорядочивания выполнения программных блоков двух или более потоков.

Взаи́мная блокиро́вка — ситуация в многозадачной среде, при которой несколько процессов находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать свое выполнение.
Механизмы синхронизации потоков в Windows:

- Critical section;
- Mutex;
- Semaphore;
- Atomic operation (interlocking function)
- Event;
- Writable timer.

Critical section — механизм синхронизации нескольких потоков одного процесса. НЕ ЯВЛЯЕТСЯ объектом ядра OS и может быть использован несколькими процессами одновременно.
Mutex – механизм синхронизации нескольких потоков разных процессов, является объектом ядра OS. Как только хотя бы один процесс запрашивает владение мьютексом, он переходит в несигнальное состояние и остается таким до тех пор, пока не будет освобожден владельцем. принадлежит процессу и служит для синхронизации только его потоков.
Semaphore - представляет собой счетчик, содержащий целое число в диапазоне от 0 до максимальной величины, заданной при его создании. Счетчик уменьшается каждый раз, когда поток успешно завершает функцию ожидания, использующую семафор, и увеличивается путем вызова функции ReleaseSemaphore. При достижении семафором значения 0 он переходит в несигнальное состояние, при любых других значениях счетчика его состояние — сигнальное.

Event - объекты-события используются для уведомления ожидающих нитей о наступлении какого-либо события. Различают два вида событий - с ручным и

автоматическим сбросом. Ручной сброс осуществляется функцией ResetEvent. События с ручным сбросом используются для уведомления сразу нескольких нитей. При использовании события с автосбросом уведомление получит и продолжит свое выполнение только одна ожидающая нить, остальные будут ожидать дальше. Event, SetEvent, WaitForSingleObject

Waitable timer - объекты ядра, которые предназначены для отсчета промежутков времени. Окончание временного интервала определяется по переходу таймера в свободное состояние (signaled). Момент перехода таймера в свободное состояние определяется одной из ожидающих функций. Таймер ожидания переходит в сигнальное состояние по завершении заданного интервала времени.

Таймер ожидания может быть либо синхронизирующим (synchronization timer), либо сбрасываемым вручную уведомляющим (manual-reset notification timer) таймером. Синхронизирующий таймер связывается с функцией косвенного вызова, аналогичной процедуре завершения расширенного ввода/вывода, тогда как для синхронизации по сбрасываемому вручную уведомляющему таймеру используется функция ожидания.

Атомарная операция — операция, которая либо выполняется целиком, либо не выполняется вовсе

Атомарные операции в Windows — это группа особых функций, названия которых начинаются с префикса Interlocked. Суть их в том, что каждая из них позволяет выполнить пару простых операций, но так, что они выполняются атомарно, то есть как бы «одним махом», так что их выполнение не может быть прервано другим потоком. Каждый, кто работает с объектом, вызывает в начале AddRef, увеличивая счётчик, а по окончании работы – Release, уменьшая его. Если счётчик при очередном вызове Release стал равен нулю, значит объект никому больше не нужен и Release удаляет его.

IntrlockedIncrement и IntrlockedDecrement выполняет атомарно сразу две операции: изменяют счётчик на единичку и сравнивают результат с нулём. Это общее свойство всех Interlocked функций: все они позволяют атомарно выполнить не одну, а как минимум две или даже три простых операции.

7. Компьютерное время: принцип вычисления компьютерного времени, социальное и компьютерное время, единица измерения компьютерного времени, Universal Coordinated Time, POSIX-время, ожидающий таймер, состояния ожидающего таймера, периодические и непериодические таймеры, работа с ожидающими таймерами в Windows.

вычисление времени в компьютере

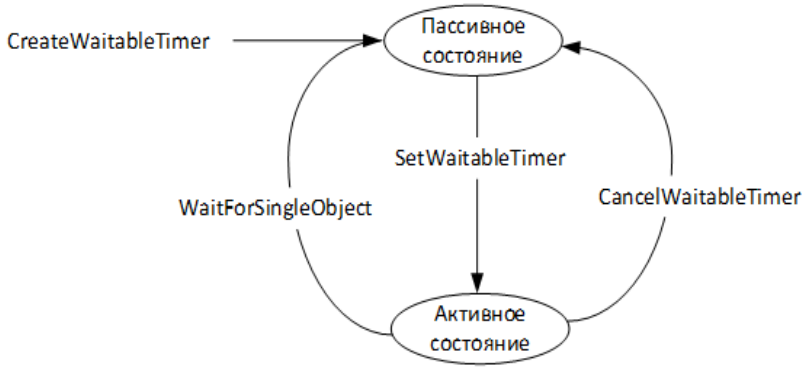
Другая разновидность часов создается из трех компонентов: кварцевого генератора, счетчика и регистра хранения. Если из кристалла кварца правильно вырезать пластину и подвести к ней напряжение, то ее можно заставить генерировать очень стабильный периодический сигнал. С помощью электронных схем этот опорный сигнал можно умножить на небольшое целое число, чтобы получить частоты до нескольких гигагерц или даже выше. В любом компьютере можно найти как минимум одну такую схему, которая обеспечивает различные компьютерные электронные схемы синхросигналом. Этот сигнал поступает в счетчик, заставляя его производить обратный отсчет до нуля. Когда значение счетчика становится нулевым, он выдает прерывание на центральный процессор.

Unix-время(Posix) — система описания моментов во времени, принятая в Unix и других POSIX-совместимых операционных системах. Определяется как количество секунд, прошедших с полуночи (00:00:00 UTC) 1 января 1970 года (четверг); этот момент называют «эпохой Unix» .

Современное **Unix-время** согласуется с UTC — отсчет происходит в секундах СИ. В Unix-времени соответствующие номера секунд повторяются, то есть високосные секунды не учитываются.

Всеми́рное координи́рованное вре́мя (англ. Coordinated Universal Time, фр. Temps Universel Coordonné; UTC) — стандарт, по которому общество регулирует часы и время. Отличается на целое количество секунд от атомного времени и на дробное количество секунд от всемирного времени UT1.

Ожидающий таймер: объект синхронизации, два состояния: сигнальное – наступление заданного момента времени; несигнальное (активное и пассивное состояние) - ждет наступления заданного момента времени.



Таймер ожидания может быть либо синхронизирующим (synchronization timer), либо сбрасываемым вручную, уведомляющим (manual-reset notification timer) таймером. Синхронизирующий таймер связывается с функцией косвенного вызова, аналогичной процедуре завершения расширенного ввода/вывода, тогда как для синхронизации по сбрасываемому вручную уведомляющему таймеру используется функция ожидания.

Для начала потребуется создать дескриптор таймера, используя для этого функцию **CreateWaitableTimer.** Второй параметр, **bManualReset**, определяет, таймер какого типа должен быть создан — синхронизирующий или уведомляющий.

Первоначально таймер создается в неактивном состоянии, но с помощью функции **SetWaitableTimer** его можно активизировать и указать начальную временную задержку, а также длительность промежутка времени между периодически вырабатываемыми сигналами

Установив синхронизирующий таймер, вы можете перевести поток в состояние дежурного ожидания путем вызова функции SleepEx, чтобы обеспечить возможность вызова процедуры завершения. В случае сбрасываемого вручную уведомляющего таймера следует организовать ожидание перехода дескриптора таймера в сигнальное состояние. Дескриптор будет оставаться в сигнальном состоянии до следующего вызова функции SetWaitableTimer..

Функция CancelWaitableTimer используется для отмены действия вызванной перед этим функции SetWaitableTimer, но при этом не изменяет сигнальное состояние таймера. Чтобы это сделать, необходимо в очередной раз вызвать функцию SetWaitableTimer.

8. Файловая система: логическая и физическая организация данных, определение файловой системы, отличие файловых систем, оглавление файловой системы, файлы, каталоги, основные функции файловой системы, буферы ввода/вывода, кеширование ввода/вывода, основные функции API файловой системы, маркер файла, текущая позиция файла, блокировка файлов, наблюдение за изменением в каталоге.

HDD (он же «винчестер») - это жесткий диск, который состоит из одного или нескольких магнитных дисков и считывающих головок. Магнитные диски вращаются, а считывающая головка (закрепленная на рычаге) перемещается над поверхностью диска и распознает сохраненные данные.
В SSD нет движущихся механических частей, он состоит из большого количества отдельных flash-накопителей, которые встроены в диск по тому же принципу, что и в USB-флешках.
Кеш — это промежуточный буфер с быстрым доступом, содержащий информацию, которая может быть запрошена с наибольшей вероятностью**.** Многие периферийные устройства хранения данных используют внутренний кэш для ускорения работы, в частности, жёсткие диски используют кэш-память от 1 до 256 Мбайт.

При разбиении диска на разделы диск разбивается на области, которые также называются разделами или логическими дисками. В каждом разделе может быть установлена своя файловая система. В настоящее время операционные системы Windows используют файловые системы FAT16, FAT32 и NTFS.

Логической записью или структурой называется упорядоченное множество данных разных типов.

На уровне прикладной программы файл представляет собой множество логических записей. На физическом уровне файл представляет собой поименованное множество секторов и кластеров, хранящихся на диске.

Маркер — это именованная точка в файле ASF. Каждый маркер состоит из имени и связанного с ним времени, измеряемого как смещение от начала файла. Приложение может использовать маркеры, чтобы назначать имена различным точкам в содержимом, отображать эти имена пользователю, а затем искать позиции маркеров. Приложение может добавлять или удалять маркеры из существующего файла ASF.

HHD: разбиение на разделы: логическое разбиение диска на разделы. В каждом разделе устанавливается своя файловая система. в каждом разделе: Boot Sector (загрузочная запись), таблицы размещения файлов и Root Directory (коревой директорий).

Файловая система: система управления файлами, часть операционной системы, обеспечивающая доступ к файлам. Устанавливает связь между логическим представлением и физическим расположением данных (абстракция над данными).

FAT – ФС: таблица размещения файлов, которая помещена в самом начале тома.
Нельзя отменить удаление, быстрый только до 200 Мб дальше медленный.

FAT 32 представляет собой цепь данных, которые связывают между собой кластеры дискового пространства и файлы. В базе данных кластеров существует только один элемент. Из них, первые два элемента представляют собой информацию о системе FAT – 32, а третий и последующий элементы ставятся в соответствии с кластерами дискового пространства.

Файловая система Windows NT (**NTFS**) обеспечивает производительность, надежность и совместимость. NTFS разрабатывалась с целью обеспечения скоростного выполнения стандартных операций над файлами (включая чтение, запись, поиск) и предоставления продвинутых возможностей. Обладает характеристиками защищенности, которые необходимы на мощных файловых серверах и высокопроизводительных компьютерах в корпоративных средах. Файловая система NTFS поддерживает контроль доступа к данным и привилегии владельца. NTFS - единственная файловая система в Windows NT, которая позволяет назначать права доступа к отдельным файлам.

Файловая система NTFS является простой, и одновременно чрезвычайно мощной. Практически все, что имеется на томе, представляет собой файл, а все, что имеется в файле представляет собой атрибут, включая атрибуты данных, атрибуты системы безопасности, атрибуты имени файла. Каждый занятый сектор на томе NTFS принадлежит какому-нибудь файлу.

логическое представление данных: файл и запись;

файл: набор логических записей;

длина имени файла: зависит от типа FS;

логическая запись: последовательность байт;

каталог – файл содержащий информацию о месте расположения других файлов; специальные каталоги: .(точка. каталог с которым работает приложение), .. (две точки родительский каталог),

специальные имена: con, lpt1, prn, aux, com... (не могут быть именами файлов).

основные функции файловой системы:

- создание/удаление каталогов,
- включение/исключение подкаталогов,
- включение/исключение файла в каталог,
- создание/удаление файла,
- открытие/закрытие доступа к файлу,
- чтение/запись логических записей файла,
- установка (поддержка) указателя файла.

указатель файла – объект файловой системы, позиционирующий логическую запись.

буферы ввода/вывода: области памяти для хранения физически считанных данных; необходимы для устранения несоответствия между физическим и логическим чтением/записью.

кэширование ввода/вывода: перемещение в быстродействующую память, наиболее часто используемых данных (обычно упреждающее чтение).

CreateFile –создать файл(название,generic read/write, атрибуты, open always / create new)
DeleteFile – удалить файл (название)
WiriteFile – запись в файл (файл, что записать(буфер), писать байт, записано байт)
ReadFile – чтение из файла (файл, буфер, читать байт, прочитано байт)
FlushFileBufer – освободить буфер (файл)
CopyFile – копироваиние файла (старый файл, новый файл)
MoveFile – перенос файла (старый и новый файлы)
ReplaceFile – перемещение с атрибутами безопасности в пределах одного логического диска (новый и старый файлы, резервный файл, опции замещения)
SetPointer – сдвинуть указатель позиции файла(файл, младшая и старшая части сдвига, текущая позиция (FILE_BEGIN(CURRENT, END)))
GetFileSizeEx – размер файла (имя файла и размер)
SetEndOfFile – установить EOF в текущую позицию файла
LockFile/UlockFile – монопольный доступ к файлу/части файла (файл, младшая и старшие части смещения, младшие и старшие части количества байт)
GetInformationByHandle – получение информации о файла (файл и информация). Получает атрибуты файла, время создания, время последней записи, размер и тд
GetFileType – получение типа файла
CreateDirectory - создание каталога (имя , атрибуты безопасности)
FindFirstFile/FindNextFile – поиск первого файла (образец для поиска и адрес поиска)
FindClose – закрыть поиск (переменная поиска файла)
RemoveDirectory – удаление каталога
MoveFile – перенос каталога / файла
Get/SetCurrentDirectory – получение/изменение текущего каталога
Наблюдение за изменениями в каталоге:

1. FinfdFirstChengeNotification (установить наблюдение за каталогом)
2. Установка фильтра (собыия которые будут отслеживаться)
3. FindNextChangeNotification (продолжить наблюдение)
4. FindCloseNextChangeNotification (остановить наблюдение)
5. WaitForSingleObject (ожиданеи изменений)

9. Механизм отображение файлов в памяти: последовательность системных вызовов Windows для создания образа файла в оперативной памяти, использование образа файла, как средства межпроцессного взаимодействия, особенности отображения файлов в linux.

File Mapping - отображение файла в виртуальную память; отображенный файл называется File View (представлением файла). File Mapping — это способ работы с файлами в некоторых операционных системах, при котором всему файлу или некоторой непрерывной его части ставится в соответствие определенный участок памяти

Преимущества:

Меньшая нагрузка на операционную систему — дело в том, что при использовании отображений операционная система не загружает в память сразу весь файл, а делает это по мере необходимости. Таким образом, даже имея небольшое количество физической памяти, можно легко отобразить файл размером 100 мегабайт или больше, и при этом для системы это не приведет к большим накладным расходам. Также выигрыш происходит и при записи из памяти на диск: если вы обновили большое количество данных в памяти, они могут быть одновременно записаны на диск.

Достаточно легко менять размер файла и при этом получать в своё распоряжение непрерывный кусок памяти нужного размера. Менеджер памяти автоматически настраивает процессор так, что странички ОЗУ, хранящие соседние фрагменты файла, образуют непрерывный диапазон адресов.

Более одного процесса могут использовать файл на диске как для чтения, так и для записи. Каждый процесс может создать новое представление, не отображая текущее представление файла.

Недостатки:

Обычный ввод-вывод чреват накладными расходами на дополнительные системные вызовы и лишнее копирование данных, использование отображений чревато замедлениями из-за страничных ошибок доступа. Допустим, страница, относящаяся к нужному файлу, уже лежит в кэше, но не ассоциирована с данным отображением. Если она была изменена другим процессом, то попытка ассоциировать её с отображением может закончиться неудачей и привести к необходимости повторно зачитывать данные с диска либо сохранять данные на диск. Таким образом, хотя программа и делает меньше операций для доступа через отображение, в реальности операция записи данных в какое-то место файла может занять больше времени, чем с использованием операций файлового ввода-вывода

Размер отображения зависит от используемой архитектуры. Теоретически, 32-битные архитектуры не могут создавать отображения длиной более 4 Гб.

последовательность системных вызовов Windows для создания образа файла в оперативной памяти

1. Создайте или откройте файловый объект, представляющий файл на диске.

HANDLE hFile = CreateFile (TEXT ("datafile.txt"), GENERIC_READ | GENERIC_WRITE, 0, NULL, CREATE_NEW, FILE_FLAG_DELETE_ON_CLOSE, NULL);

2. Создайте map объект для файла, который содержит информацию о том, как получить доступ к файлу и его размеру. После создания вышеуказанного файла мы используем его дескриптор и создаем его отображение в физической памяти.

HANDLE CreateFileMappingA (HANDLE hFile, NULL, PAGE_READWRITE, 0, bufferSize, filename);

Шаг 3. Отображение всего или части объекта отображения файлов из физической памяти в виртуальное адресное пространство вашего процесса.

LPVOID MapViewOfFile (HANDLE hFileMappingObject, DWORD dwDesiredAccess, DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow, SIZE_T dwNumberOfBytesToMap);

// Может использоваться как

void*p=::MapViewOfFile (hFileMapping, FILE_MAP_ALL_ACCESS, 0, param1, param2);

4. Очистка

4 (A) Отмените отображение объекта отображения файла из адресного пространства процесса.

BOOL UnmapViewOfFile (LPCVOID lpBaseAddress);

4 (B) Закройте объект сопоставления файлов. Этот шаг удаляет отображение файла из физической памяти.

CloseHandle (hFileMapping);

4 (C) Закройте файловый объект. Здесь закройте файл, открытый на диске, и освободите ручку. Поскольку на первом шаге мы устанавливаем флаг FILE_FLAG_DELETE_ON_CLOSE , файл будет удален после этого шага.

CloseHandle (hFile);

Также есть функция, которая “сбрасывает” виртуальную память обратно в файл.

```
//  WINBASEAPI BOOL WINAPI FlushViewOfFile(                 // зафиксировать view в файл
//                       LPCVOID lpBaseAddress,             //  = MapViewOfFile (...)
//                       SIZE_T  dwNumberOfBytesToFlush     // количество байт
//                        );
```

использование образа файла, как средства межпроцессного взаимодействия

Для того, чтобы использовать представление файла в разных процессах, необходимо при создании Маппинга указать его имя, а в другом процессе вызывать функцию OpenFileMapping (с указанием имени Маппинга при создании).

Пример:

Это процесс, который создает Маппинг с заданным именем.

```
hMapFile = CreateFileMapping(
               INVALID_HANDLE_VALUE,    // использование файла подкачки
               NULL,                    // защита по умолчанию
               PAGE_READWRITE,          // доступ к чтению/записи
               0,                       // макс. размер объекта
               BUF_SIZE,                // размер буфера
               szName);                 // имя отраженного в памяти объекта
```

Имя Маппинга

```
if (hMapFile == NULL || hMapFile == INVALID_HANDLE_VALUE)
{
   printf("Не может создать  отраженный в памяти объект (%d).\n",
   GetLastError());
   return;
}
pBuf = (LPTSTR) MapViewOfFile(hMapFile,   // дескриптор  отраженного
                                          // в памяти объекта
                     FILE_MAP_ALL_ACCESS, // разрешение чтения/записи
                     0,
                     0,
                     BUF_SIZE);
```

А это второй процесс, который открывает существующий Маппинг с заданным именем.

```
hMapFile = OpenFileMapping(
              FILE_MAP_ALL_ACCESS,   // доступ к чтению/записи
              FALSE,                 // имя не наследуется
              szName);               // имя "проецируемого " объекта
```

Такое же имя

```
if (hMapFile == NULL)
{
   printf("Невозможно открыть объект "проекция файла" (%d).\n",
   GetLastError());
   return;
}

pBuf = MapViewOfFile(hMapFile,    // дескриптор "проецируемого" объекта
          FILE_MAP_ALL_ACCESS,  // разрешение чтения/записи
          0,
          0,
          BUF_SIZE);
```

10. Динамически вызываемые библиотеки: структура DLL-библиотеки, экспорт функций, загрузка динамической библиотеки, динамический вызов функций динамической библиотеки, создание и применение библиотеки импорта.

Динамическая библиотека (DLL) — это библиотека динамической компоновки, которая выступает в роли общего хранилища ресурсов и функций для других приложений. При использовании динамической сборки в приложение встраивается только ссылка на DLL файл в системе, а когда она понадобится приложение благодаря библиотеке импорта сопоставит ее с областью памяти приложения.

Библиотека должна содержать:

- необязательную часть кода, которая отвечает за инициализацию и очистку библиотеки;
- набор подпрограмм библиотеки;
- явное указание, какие подпрограммы должны экспортироваться из библиотеки.

Инициализирующая часть кода оформляется в виде процедуры, которая получает ряд параметров, один из которых указывает на причину обращения. Этот параметр может принимать одно из следующих значений. DLL_PROCESS_ATTACH/DETACH (библиотека подключена/отключена), DLL_THREAD_ATTACH/DETTACH (процесс создал/завершил поток)

В языке C++ основной функцией dll-библиотеки является функция, которую обычно называют DllMain, и которая выглядит следующим образом.

```
BOOL APIENTRY DllMain(HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpReserved)
{ switch (ul_reason_for_call)
   { case DLL_PROCESS_ATTACH:
     case DLL_THREAD_ATTACH:
     case DLL_THREAD_DETACH:
     case DLL_PROCESS_DETACH:
      break;
    }
  return TRUE;
 }
```

Функция DllMain получает три параметра. Первый является идентификатором библиотеки. Второй определяет причину вызова функции DllMain. Третий параметр зарезервирован для внутреннего использования в Windows.

экспорт функций

Функции, которые будут экспортироваться из dll-библиотеки, должны быть объявлены со спецификацией extern "C" __declspec(dllexport), которая указывает, что функция должна быть доступна приложению, использующему библиотеку, и что имена должны преобразовываться по правилам языка C.

Компоновщик использует преобразованные (decorated) имена функций. Для языка C++ в такое имя добавляется информации о классе или пространстве имён, которому принадлежит функция, о типах параметров и результата, о соглашении о вызовах.

Для языка C способ преобразования имени зависит от соглашения о вызовах. директива extern "C", которая указывает, что необходимо применять правила компоновки языка C.

загрузка динамической библиотеки

Прежде чем вызывать подпрограммы, включённые в dll-библиотеку, их надо импортировать в приложение. Это можно сделать двумя способами:

- путём объявления внешней процедуры или функции
- динамически с помощью Win32 API функций.

Подпрограммы из dll-библиотеки будут импортированы только во время работы приложения.

Чтобы загрузить dll-библиотеку, операционная система должна найти её. Поиск осуществляется в следующих местах:

- директория, из которой загружено приложение, требующее dll-библиотеку;
- текущая директория;
- системная директория (обычно C:\Windows\System32);
- системная директория для 16-битных приложений (обычно C:\Windows\System);
- Windows-директория;
- директории, указанные в переменной окружения PATH.

Последовательность просмотра директорий может быть разной, это зависит от настроек, но, в общем, просматриваются перечисленные директории до нахождения требуемой библиотеки. Если ни в одной из перечисленных директорий библиотека не найдена, приложение получает уведомление об ошибке.

Динамическое подключение

Доступ к подпрограммам dll-библиотеки можно получить с помощью Win32 API функций LoadLibrary, FreeLibrary и GetProcAddress. Это позволяет уменьшать требуемое количество памяти и запускать приложение, даже если какие-то библиотеки отсутствуют на компьютере.

При динамическом подключении dll-библиотеки компилятор не может проверить типы и количество передаваемых параметров. Поэтому программист сам должен следить за тем, чтобы процедура получила корректный набор параметров.

Динамическое подключение dll-библиотек более трудоёмко, чем статическое, т.к. нужно самостоятельно вызывать все необходимые функции и проверять возвращаемые ими значения. Кроме того, при динамическом подключении генерируется чуть более объёмный код. Но динамическое подключение является более гибким механизмом. При статическом подключении в случае отсутствия необходимой dll-библиотеки приложение просто не запустится. При динамическом подключении в случае отсутствия dll-библиотеки программист может предпринять какие-либо действия, например, попытаться загрузить другую библиотеку или позволить пользователю работать с теми функциями приложения, которые определены не в отсутствующей библиотеке.

динамический вызов функций динамической библиотеки

Функция GetProcAddress осуществляет поиск нужной процедуры в dll-библиотеке. Функция получает идентификатор библиотеки и имя или номер процедуры и возвращает указатель на процедуру, если она была найдена, и 0 в противном случае.

```
// Использование dll-библиотеки. Пример библиотеки см. в раздел 2.2.2
#include <cstdio>
#include <windows.h>

typedef void (*LibraryFunction)(char *, char *);       // Объявляем тип для указателя на библиотечную функ

void main()
{ char str[256], res[256];
  HINSTANCE hLib;                        // Объявляем идентификатор библиотеки
  LibraryFunction  f;                       // Объявляем указатель на библиотечную функцию

  hLib = LoadLibrary(TEXT("ExampleDll.dll"));           // Загружаем библиотеку
  if (hLib == NULL)                      // Проверяем результат загрузки библиотеки
   { printf("Unable to load the library 'ExampleDll.dll'!\n");
     return;
    }

  // Получаем указатель на функцию ExampleFunction и преобразуем его к нужному типу
  f = (LibraryFunction)GetProcAddress(hLib, "ExampleFunction");
  if (!f)                            // Проверяем полученный указатель
   printf("Unable to find the function 'ExampleFunction'!\n\n");
  else
   { printf("Input a string\n"); gets(str);
     f(str, res);
     printf("'%s'\n\n", res);
    }
```

создание и применение библиотеки импорта

Для того чтобы редактор связей мог создать ссылку, в файл проекта приложения вы должны включить так называемую библиотеку импорта (import library). В них содержится не сам код библиотеки, а только ссылки на все функции, экспортируемые из файла DLL, в котором все и хранится. В результате библиотеки импортирования, как правило, имеют меньший размер, чем DLL-файлы. Библиотека импорта создается либо на основе dll-файла библиотеки, либо на основе файла определения модуля (.def), используемого для создания DLL-библиотеки. После компиляции dll Visual Studio создаст библиотеку импорта в виде lib-файла, расположенного в том же каталоге, что и исходный dll или def-файл. Этот файл необходимо включить в проект создаваемого вами приложения, пользующегося функциями DLL-библиотеки.

Статическое подключение

Самый простой способ импортировать процедуру или функцию – тем или иным способом объявить её как внешнюю. Необходимо также указать имя подключаемой dll-библиотеки.

Для статического подключения dll-библиотеки в программе на языке C++ необходимо скопировать файл .lib в директорию с исходным кодом приложения и указать имя этого файла в настройках компоновщика. Кроме того, поскольку функции должны иметь прототипы, потребуется заголовочный файл с прототипами функций, который

необходимо вставить с помощью директивы препроцессора #include в файлы с исходным кодом, в которых предполагается использование функций dll-библиотеки. Заголовочный файл поставляется разработчиком библиотеки. Функции должны быть объявлены со спецификацией extern "C" __declspec(dllimport), которая указывает, что функция будет импортирована из внешнего модуля.

Плюсы использования DLL:

Экономия: память и сокращает подкачку. Много процессов одновременно могут обращаться к ОДНОМУ инстансу библиотеки в памяти.

Обслуживание: не нужно заново компилировать приложение если вы внесли изменение в библиотеку. Т.к. библиотека находится ВНЕ приложения

Послепродажная поддержка: после выпуска приложения достаточно будет менять dll файлы для увеличения функционала. Легкая реализация обновлений

Межъязыковое использование: библиотека DLL может быть использована программой, написанной на любом языке программирования лишь при условии правильного соглашения о вызове функций из данной библиотеки (порядок передачи параметров в стек, очистка стека, передача аргументов в регистры)

Динамическая подгрузка: прямо во время исполнения приложения или при его загрузке можно, например, по геолокации загрузить библиотеку с нужной локализацией слов. Без создания отдельных приложений для каждого региона.

Минусы:

Автономность: Приложение, использующее dll, не является автономным, оно требует для работы dll библиотеку, которая находится вне .exe файла.

Структура .h файлов:

```
#pragma once

#ifdef MATHLIBRARY_EXPORTS
#define MATHLIBRARY_API __declspec(dllexport)
#else
#define MATHLIBRARY_API __declspec(dllimport)
#endif

extern "C" MATHLIBRARY_API bool fibonacci_next();
extern "C" MATHLIBRARY_API unsigned long long fibonacci_current();
extern "C" MATHLIBRARY_API unsigned fibonacci_index();
```

Зачем extern "C" в dll: extern "C" делает имя функции в C++ связанным с C (компилятор не искажает имя), чтобы клиентский код C мог ссылаться на вашу функцию с помощью совместимого заголовочного файла C, содержащего только объявление вашей функции.

Экспорт функций:

Экспортируемые функции доступны для вызова приложениям Windows. Неэкспортируемые являются локальными для DLL-библиотеки, они доступны только для функций библиотеки.

Самый простой способ сделать функцию экспортируемой в системе - перечислить все экспортируемые функции в файле определения модуля (.def) при помощи оператора EXPORTS:

Динамическая загрузка\выгрузка библиотеки:

В качестве примера приведем фрагмент исходного текста приложения, загружающего DLL-библиотеку из файла srcdll.dll

```
HINSTANCE hLib;
hLib = LoadLibrary("srcdll.dll");
if(hLib >= HINSTANCE_ERROR)
{
  // Работа с DLL-библиотекой
}
FreeLibrary(hLib);
```

Функция LoadLibrary может быть вызвана разными приложениями для одной и той же DLL-библиотеки несколько раз. В этом случае загрузка DLL-библиотеки выполняется только один раз. Последующие вызовы функции LoadLibrary приводят только к увеличению счетчика использования DLL-библиотеки.

Динамический вызов функций из dll:

Для того чтобы вызвать функцию из библиотеки, зная ее идентификатор, необходимо получить значение дальнего указателя на эту функцию, вызвав функцию GetProcAddress :

```
FARPROC WINAPI GetProcAddress(HINSTANCE hLibrary, LPCSTR lpszProcName);
```

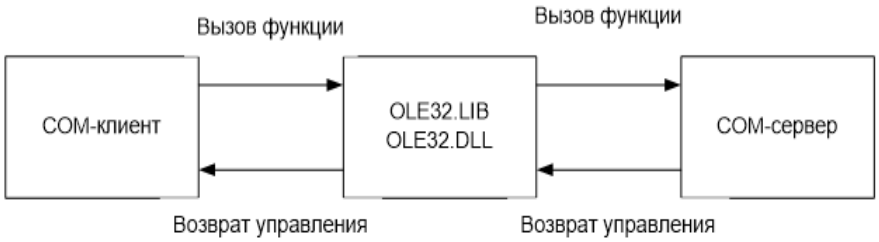
hLibrary - идентификатор DLL-библиотеки, полученный ранее от функции LoadLibrary.

lpszProcName - указатель на строку, содержащую имя функции или ее порядковый номер, преобразованный макрокомандой MAKEINTRESOURCE.

11. Спецификация COM: понятие позднего связывания программных модулей, COM-интерфейс, стандартные COM-интерфейсы, структура COM-клиента, структура COM/DLL-сервера, экспортируемые стандартные функции, регистрация COM/DLL-сервера.

Common Object Model (COM) – спецификация, позволяющая разрабатывать программное обеспечения. Модель программного обеспечения. Разработана Microsoft.
COM-программирование – разработка COM-компонентов (объектов), программного обеспечения, имеющего модель COM.
COM-объект – специализированный объект времени исполнения (экземпляр). Для идентификации компонента используется идентификатор CLSID.

Понятие позднего связывания программных модулей и Структура COM-клиента: подключение программного модуля во время исполнения программы. При создании объекта посредником между COM-клиентом (программный модуль, создающий COM-объект и использующий его методы) и COM-сервером (программный модуль, реализующий COM-объект.) выступает библиотека OLE32.DLL (библиотека импорта OLE32.LIB). Все функции OLE32.DLL возвращают значение типа HRESULT.



Пример связывания:

На стороне COM-сервера

```
#pragma once
#include <objbase.h>

static const GUID IID_IX =
{ 0xe04d1098, 0x2f6f, 0x4413, { 0x88, 0xa9, 0x17, 0xaa, 0x9b, 0x97, 0x9d, 0x95 } };

interface IX:IUnknown
{
    virtual HRESULT  STDMETHODCALLTYPE  Fx1() = 0;
    virtual HRESULT  STDMETHODCALLTYPE  Fx2() = 0;
};
```

На стороне COM-клиента

```
#include "stdafx.h"
#include <iostream>
#include <objbase.h>
#include "IX.h"
#include "COM.h"

int _tmain(int argc, _TCHAR* argv[])
{

  HRESULT hr0, hr;

  IX* pIX = NULL;

  CoInitialize(NULL);           // инициализация библиотеки OLE32

  hr0 = CoCreateInstance(CLSID_CA, NULL, CLSCTX_INPROC_SERVER, IID_IX, (void**)&pIX );

  if (SUCCEEDED(hr0))
  {

      if (!SUCCEEDED(pIX->Fx1())) std::cout << "error IX::Fx1"<<std::endl ;

      if (!SUCCEEDED(pIX->Fx2())) std::cout << "error IX::Fx2"<<std::endl ;

  }
  else std::cout << "error IX"<<std::endl ;

  pIX->Release();

  CoFreeUnusedLibraries();    // завершение работы с библиотекой

  return 0;
}
```

Пояснения к скрину выше:

все константы и прототипы функций необходимые для поддержки COM-программирования становятся доступными с помощью #include <objbase.h>

CoCreateInstance - функция для создания COM объекта на клиенте

- CLSID_CA - идентификатор объекта (GUID)
- COM-сервер может иметь тип:
  - CLSCTX_INPROC_SERVER (DLL внутрипроцессный сервер);
  - CLSCTX_LOCAL_SERVER (EXE-сервер на том же компьютере),
  - CLSCTX_REMOTE_SERVER (EXE-сервер на удаленном компьютере);
- IID_IX - идентификатор интерфейса к которому будет приведен COM-объект;
- pIX - выходящий параметр, который будет инициализирован внутри функции, куда будет помещен COM-объект.

COM-интерфейс

Интерфейсы используются для доступа к методам COM-объектов. Описывает один или несколько методов. Каждый интерфейс имеет свой идентификатор. В спецификации COM есть несколько стандартных интерфейсов (идентификатор и методы которого являются общедоступными), которые заранее прописаны в документации (например, IUnknown (базовый интерфейс для всех интерфейсов), IClassFactory(создаёт экземпляр COM-объекта)).

методы интерфейса IUnknown

- QueryInterface (получает id в ответ отправляет ссылку, можно получить указатель на другой интерфейс)

- AddRef (внутри компонента поддерживается счётчик, этот метод увеличивает на 1 счётчик ссылок на интерфейс)
- Release (уменьшает счётчик ссылок на интерфейс на 1)

Фабрика классов - Специальная компонента, задача которой создавать экземпляры объекта. Для каждого компонента своя фабрика. Фабрика классов реализует интерфейс IClassFactory

методы интерфейса IClassFactory

- CreateInstance создаёт экземпляры рабочих компонентов (создаёт инстанс СА). Для каждого сервера фабрика классов (ClassFactory) создаёт компонент. После создания инстанса, возвращает клиенту указатель на интерфейс IUnknown.
- LockServer(params). LockServer мб необходим, если необходимо эксклюзивное использование сервера (1 – сервер заблокирован, 0 – разблокирован).

Счетчик ссылок на интерфейсы - Для каждого компонента необходимо подсчитывать сколько ссылок сделано на его интерфейс. Это нужно для того, чтобы знать, сколько клиентов подключены к DLL. Если счётчик компонента == 0, он сам себя убивает. Увеличивается при использовании какого-либо метода, уменьшается после Release.

Поддерживают соглашение о вызове stdcall (аргументы передаются через стек, справа налево, очистку стека производит вызываемая подпрограмма); используется макрос STDMETHODCALLTYPE. Возвращают значение типа HRESULT (как функции OLE32). Исключение составляют методы AddRef и Release.

COM/DLL-сервер:

COM-сервер представляет собой приложение или библиотеку, которая предоставляет услуги приложению-клиенту или библиотеке. COM-сервер содержит один или более COM-объектов, где COM-объекты выступают в качестве наборов свойств, методов и интерфейсов.

Клиенты не знают как COM-объект выполняет свои действия. COM-объект предоставляет свои услуги при помощи интерфейсов. В дополнение, приложению-клиенту не нужно знать, где находится COM-объект. Технология COM обеспечивает прозрачный доступ независимо от местонахождения COM-объекта.

Когда клиент запрашивает услугу от COM-объекта, он передает COM-объекту идентификатор класса (CLSID). После передачи CLSID, COM-сервер должен обеспечить так называемую фабрику класса, которая создает экземпляры COM-объектов.
В общих чертах, COM-сервер должен выполнять следующее:

- регистрировать данные в системном реестре Windows для связывания модуля сервера с идентификатором класса (CLSID);
- предоставлять фабрику COM-класса, создающую экземпляры COM-объектов;
- обеспечивать механизм, который выгружает из памяти серверы COM, которые в текущий момент вревающий один или более интерфейс. COM-объекты могут

предоставлять только те услуги, которые определены в интерфейсах CoClass. Экземпляры CoClass создаютсямени не предоставляют услуг клиентам.

Экспортируемые стандартные функции:

Внутренний COM-сервер должен экспортировать четыре функции:

1. function **DllRegisterServer**: HResult; stdcall;
2. function **DllUnregisterServer**: HResult; stdcall;
3. function **DllGetClassObject** (const CLSID, IID: TGUID; var Obj): HResult;
4. function **DllCanUnloadNow**: HResult; stdcall;

DllRegisterServer - применяется для регистрации DLL COM-сервера в системном реестре Windows. При регистрации COM-класса в системном реестре создается раздел в HKEY_CLASSES_ROOT\CLSID\{XXXXXXXX-XXXX-XXXX-xxxx-xxxxxxxx}, где число, записанное вместо символов x, представляет собой CLSID данного COM-класса.

DllUnregisterServer - применяется для удаления всех разделов, подразделов и параметров, которые были созданы в системном реестре функцией DllRegisterServer при регистрации DLL COM-сервера.

DllGetclassObject - возвращает фабрику класса для конкретного COM-класса.

DllcanUnloadNow - применяется для определения, можно ли в настоящий момент времени выгрузить DLL COM-сервера из памяти.

Регистрация COM/DLL-сервера:
Утилита regsvr32.exe является стандартной программой командной строки для регистрации и отмены регистрации элементов управления OLE, ActiveX и библиотек DLL в реестре Windows.

Реестр Windows, или системный реестр — иерархически построенная база данных параметров и настроек в большинстве операционных систем Microsoft Windows.

12. Управление пользователями и группами пользователей в Windows: понятие дискреционной системы безопасности, типы Windows-пользователей, группы пользователей, возможности API управления пользователями и группами.

Дискреционное разграничение доступа к объектам характеризуется следующим набором свойств:

- все субъекты и объекты компьютерной системы должны быть однозначно идентифицированы;
- для любого объекта компьютерной системы определен пользователь-владелец;
- владелец объекта обладает правом определения прав доступа к объекту со стороны любых субъектов компьютерной системы;
- в компьютерной системе существует привилегированный пользователь, обладающий правом полного доступа к любому объекту (или правом становиться владельцем любого объекта).

Дискреционное разграничение доступа реализуется обычно в виде матрицы доступа, строки которой соответствуют субъектам компьютерной системы, а столбцы — ее объектам. Элементы матрицы доступа определяют права доступа субъектов к объектам. В целях сокращения затрат памяти матрица доступа может задаваться в виде списков прав субъектов (для каждого из них создается список всех объектов, к которым разрешен доступ со стороны данного субъекта) или в виде списков контроля доступа (для каждого объекта информационной системы создается список всех субъектов, которым разрешен доступ к данному объекту).

Достоинства:

- простая реализация (проверка прав доступа субъекта к объекту производится в момент открытия этого объекта в процессе субъекта)
- хорошая изученность (в наиболее распространенных операционных системах универсального назначения типа Microsoft Windows и Unix применяется именно эта модель разграничения доступа).

Недостатки:

- статичность разграничения доступа — права доступа к уже открытому субъектом объекту в дальнейшем не изменяются независимо от изменения состояния компьютерной системы.
- не обеспечивает защиты от утечки конфиденциальной информации.
- автоматическое назначение прав доступа субъектам (из-за большого количества объектов в информационной системе в качестве субъектов доступа остаются только ее пользователи, а значение элемента матрицы доступа вычисляется с помощью функции, определяющей права доступа порожденного пользователем субъекта к данному объекту компьютерной системы).

типы Windows-пользователей

Все учетные записи в зависимости от своих функциональных возможностей разделяются на три типа:

- администратор компьютера
- ограниченная учетная запись
- гостевая учетная запись

Пользователь с ограниченной учетной записью может выполнять операции со своим паролем (создание, изменение, удаление), изменять рисунок своей учетной записи, параметры настройки рабочего стола, а также просматривать файлы. Учетная запись администратора позволяет выполнять следующие операции:
• Создание, удаление и редактирование учетных записей пользователей (в том числе и собственной учетной записи);
• Операции со своим паролем (создание, редактирование, удаление);
• Установка и удаление программ и оборудования, редактирование их параметров и свойств;
• Чтение всех общих файлов;
• Внесение изменений в конфигурацию на уровне системы.

Что касается гостевой записи, то она формируется автоматически в процессе установки системы, и предназначена для сторонних пользователей, не имеющих на данном компьютере собственных учетных записей. Под учетной записью гостя нет доступа к файлам, папкам, параметрам и приложениям, которые защищены паролем.

Группа это просто набор учетных записей пользователей, которые объединены по какому-либо признаку, например, пользователи одной группы могут работать в одном отделе.

Группы пользователей
- Администраторы. Неограниченный доступ.
- Операторы архива. Права создания резервной копии даже тех объектов, к которым не имеют доступа.
- Опытные пользователи.
- Пользователи системного монитора. Есть чудесная вещь под названием Системный монитор(perfmon.msc), с помощью которого можно отследить использование различных ресурсов компьютером. А группа дает доступ к данному инструменту.
- Операторы настройки сети. Члены группы могут изменять параметры TCP/IP.
- Пользователи удаленного рабочего стола. Пользователи этой группы смогут входить в систему через удаленный рабочий стол.
- Пользователи журналов производительности. 4-ая группа дает только поверхностный доступ к Системному монитору. Данная группа дает более полные права.
- Пользователи DCOM. Пользователи группы могут манипулировать объектами распределенной модели DCOM.
- Криптографические операторы.
- Читатели журнала событий.

возможности API управления пользователями и группами

1. Имя текущего пользователя. GetUserName (буфер имени, длина буфера)

2. Информация о пользователей NetUserGetInfo (servername, username, уровень инфы, буфер)

3. Освободить память NetApiBufferFree

4. Получить коллекцию пользователей NetUserEnum

5. Группы пользователя NetUserGetLocalGroups

6. Добавить пользователя NetUserAdd

7. Удалить пользователя NetUserDel

8. Изменить информацию о пользователе NetUserSetInfo

9. Изменить пароль NetUserChangePassword

10. Добавление группы NetLocalGroupAdd

11. Получить информацию о группе GetLocalGroupInfo

12. Получить коллекцию групп NetLocalGroupEnum

13. Изменить информацию о группе NetLocalGroupSetInfo

14. Добавление членов локальной группы (исп. INFO_3) NetLocalGroupAddMembers

15. Удаление членов группы NetLocalGroupDelMember

16. Получить коллекцию членов группы NetLocalGroupGetMembers

17. Установка членов локальной группы NetLocalGroupSetMembers

18. Удалить группу NetLocalGroupDel

19. Подключение пользователя LogUserW

```
//BOOL LogonUserW (
//      LPCWSTR lpszUsername,   // имя пользователя
//      LPCWSTR lpszDomain,     //NULL или имя домена
//      LPCWSTR lpszPassword,   // пароль
//      DWORD dwLogonType,      // LOGON32_LOGON_BATCH/LOGON32_LOGON_INTERACTIVE ...LOGON32_LOGON_NETWORK
//      DWORD dwLogonProvider,  //LOGON32_PROVIDER_DEFAULT,  LOGON32_PROVIDER_WINNT50,LOGON32_PROVIDER_WINNT40
//      PHANDLE phToken         //CreateProcessAsUser
//            );
```

13. Windows-сервисы: понятие и назначения сервиса, структура сервиса, порядок разработки и принцип работы сервиса, команды управления сервисом.

Сервис - процесс, выполняющий служебные функции (программа, которая запускается при загрузке операционной системы). Обычно сервис обеспечивает фоновый процесс(сервер), работу с внешним устройством (драйвер), следит за работой приложений (монитор). Сервис может быть запущен при загрузке ОС или из приложения.

```
#pragma once
#include <windows.h>
#include <iostream>
#include <fstream>

#define SERVICENAME  L"smwsvc"
#define TRACEPATH "D:\\Service.trace"

VOID WINAPI ServiceMain(DWORD dwArgc, LPTSTR *lpszArgv);      //main service function
VOID WINAPI ServiceHandler(DWORD fdwControl);                 //service handler function

void trace(const char* msg, int r = std::ofstream::app);
```

```
// service
#include <windows.h>
#include <iostream>
#include <fstream>
#include "Service.h"    // SERVICENAME, ServiceMain-прототип, trace-прототип


int main()              // процесс-Windows-сервис
{
    WCHAR  ServiceName[] = SERVICENAME;                                 // имя сервиса

    SERVICE_TABLE_ENTRY ServiceStartTable[] = {                         // таблица сервисов
                                            {ServiceName, ServiceMain},  // имя сервиса, точка входа сервиса
                                            {NULL, NULL}                 // конец таюлицы
                                           };

    if (!StartServiceCtrlDispatcher(ServiceStartTable))                 // запуск сервисов
          trace("StartServiceCtrlDispatcher", std::ofstream::out);

    return 0;
}
```

Создание и регистрация сервиса

1. OpenSCManager(NULL, NULL, SC_MANAGER_CREATE_SERVICE)
2. CreateService (manager, name, nameToDisplay, SERVICE_ALL_ACCESS, type, startType(autorun), servicepath)

Удаление сервиса.

1. OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS)
2. OpenService(manager, name, SERVICE_ ALL_ACCESS)
3. DeleteService(openService)

Остановка сервиса.

1. OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS)
2. OpenService(manager, name, SERVICE_ ALL_ACCESS)
3. ControlService(openService, SERVICE_CONTROL_STOP, &status)

Старт сервиса.

1. OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS)
2. OpenService(manager, name, SERVICE_ ALL_ACCESS)
3. StartService(openService, 0, null)

14. Структурная обработка ошибок в Windows: программное исключение, программные конструкции для обработки ошибок в Windows, фильтры, возможности API для структурной обработки ошибок, генерация ошибок, финальная обработка исключений.

Исключение – событие в программе, произошедшее во время ее выполнения, в результате которого нормальное выполнение программы становится невозможным. (Ошибка в программе). Для дальнейшей работы приложения требуется либо восстановить ее рабочее состояние, либо аварийно ее завершить с очищением всех ресурсов этой программы. Для этого применяется механизм SHE, он может отлавливать не только программные, но и аппаратные ошибка имеет определенный тип

SEH (structured exception handling) – механизм структурной обработки событий в Windows.

Фрейм – блок кода, в котором может произойти исключение. Код называется охраняемым кодом.

Обработчик исключения – блок программного кода, который обрабатывает исключение.

Исключения могут быть основаны на оборудовании или программном обеспечении. Структурированная обработка исключений полезна даже в том случае, когда приложения не могут полностью восстанавливаться после исключений оборудования или программного обеспечения. SEH позволяет отображать сведения об ошибках и захватывать внутреннее состояние приложения, чтобы помочь в диагностике проблемы. Это особенно полезно для периодических проблем, которые не очень просты в воспроизведении.

Составной оператор после __try предложения — тело или защищенный раздел. __except Выражение также называется критерием фильтра. Его значение определяет, как обрабатываются исключения. Составной оператор после предложения __except является обработчиком исключения. Обработчик задает действия, выполняемые при возникновении исключения во время выполнения раздела body. Выполнение происходит следующим образом:
- Сначала выполняется защищенный раздел.
- Если исключение при этом не возникает, выполнение переходит в инструкцию, стоящую после предложения __except .
- Если во время выполнения защищенного раздела возникает исключение или в любой подпрограмме вызывается защищенный раздел, __except выражение вычисляется. Возможны три значения.

- EXCEPTION_CONTINUE_EXECUTION (-1) Исключение закрыто. Выполнение продолжается в точке, в которой возникло исключение.
- EXCEPTION_CONTINUE_SEARCH (0) исключение не распознано. Продолжайте выполнять поиск обработчика в стеке, сначала для содержащихся try-except инструкций, а затем для обработчиков со следующим высшим приоритетом.

- EXCEPTION_EXECUTE_HANDLER (1) распознано исключение.

нельзя goto охраняемый код и в обработчик. в выражении фильтра можно использовать две функции: GetExeptionCode, GetExceptionInformation. Переменные объявленные внутри {} – локальные

```
#include "stdafx.h"
#include "windows.h"
#include <iostream>
int _tmain(int argc, _TCHAR* argv[])
{

int  x = 33;
int *px = NULL;
 __try
 {
  // охраняемый код
    std::cout << "Hello from farame code 1" << std::endl;
    std::cout << "x = " << *px << std::endl;
    std::cout << "Hello from farame code 2" << std::endl;
 }
 __except(EXCEPTION_EXECUTE_HANDLER)
 {
  // код обработки исключения
    std::cout << "Hello from handlercode 1" <<  std::endl;
    px = &x;
 }

 std::cout << "x = " << *px << std::endl;
 system( "pause");
 return 0;
}
```

DWORD GetExceptionCode() – может использоваться только в выражении-фильтре (для определения дальнейших действий: обрабатывать, искать обработчик, вернуть управление в точку прерывания) или в блоке обработки исключения (для получения кода исключения).

В **функции фильтра** нельзя вызывать GetExeptionCode или GetExceptionInformation, но можно использовать для инициализации параметров этой функции в выражении-фильтре. В общем случае это не удается: один оператор C++, как правило, состоит из нескольких инструкций процессора, а возврат осуществляется к инструкции, что может привести к зацикливанию (постоянно вызывается __exept)

LPEXCEPTION_POINTERS GetExceptionInformation() – может быть использована только в выражении фильтра

генерация программных исключений: RaiseException(код исключения, флаг возобновляемого исключения, количество аргументом, массив аргументов)

__leave – для выхода из блока

финальная обработка исключений __try {code} __finaly { if(AbnormaleTermination) {аварийное завершение} else {успешное}}

15. Windows-консоль: определение, применение стандартных потоков для ввода/вывода в консоль, возможности API для управления консолью.

Входной буфер консоли содержит очередь записей, которые описывают события ввода. События ввода подразделяются на следующие категории:

- ввод с клавиатуры;
- ввод с мыши;
- изменение размеров окна;
- изменение фокуса ввода;
- события, связанные с меню.

Буфер экрана является двумерным массивом, элементы которого представляют собой записи типа:

```
typedef struct _CHAR_INFO {
union {
WCHAR UnicodeChar;
CHAR AsciiChar;
} Char;bb
WORD Attributes;
} CHAR_INFO, *PCHAR_INFO;
```

где объединение Char содержит символ, представленный в коде Unicode или ASCII, а поле Attributes определяет цвет фона и цвет текста, которыми выводятся символы на экран дисплея. Это значение может быть равно 0, что обозначает фон — черный, а цвет — белый, или любой комбинации из следующих констант:

- BACKGROUND_BLUE — фон синий;
- BACKGROUND_GREEN — фон зеленый;
- BACKGROUND_RED — фон красный;
- BACKGROUND_INTENSITY — фон яркий;
- FOREGROUND_BLUE — текст синий;
- FOREGROUND_GREEN — текст зеленый;
- FOREGROUND_RED — текст красный;
- FOREGROUND_INTENSITY — текст яркий.

Консоль — это приложение, которое предоставляет службы ввода-вывода для приложений в символьном режиме. Состоит из входного буфера и одного или нескольких буферов экрана. Входной буфер содержит очередь входных записей, каждая из которых содержит сведения о событии ввода. Буфер экрана — это двумерный массив символьных и цветовых данных для вывода в окне консоли. Консоль может совместно использоваться любым количеством процессов.

Стандартные потоки:

```cpp
// OS_0065
 #include <Windows.h>
 #include <iostream>

 int main()
 {
     HANDLE hc = CreateFile(L"CONOUT$", GENERIC_WRITE, NULL, NULL, OPEN_EXISTING, FILE_SHARE_WRITE,NULL);

     DWORD n = 0;
     CHAR  buf[] = "--- console output ---\n";
     BOOL b = WriteFile(hc, buf, sizeof(buf), &n, NULL);

     CloseHandle(hc);

     return 0;
 }
```

```cpp
// OS_0066
#include <Windows.h>
#include <iostream>

int main()
{
    HANDLE ohc = CreateFile(L"CONOUT$", GENERIC_WRITE, NULL, NULL, OPEN_EXISTING, FILE_SHARE_WRITE, NULL);
    HANDLE ihc = CreateFile(L"CONIN$",  GENERIC_READ , NULL, NULL, OPEN_EXISTING, FILE_SHARE_WRITE, NULL);
    HANDLE ehc = CreateFile(L"CONOUT$", GENERIC_WRITE,  NULL, NULL, OPEN_EXISTING, FILE_SHARE_WRITE, NULL);
    DWORD n = 0;
    CHAR  obuf[] = "--- console output --->  ";
    CHAR  ibuf[] = "\n--->                   ";
    try
    {
        if (!WriteFile(ohc, obuf, sizeof(obuf), &n, NULL)) throw "WriteFile";
        if (!ReadFile(ihc, ibuf + 6, 10, &n, NULL))        throw "ReadFile ";
        if (!WriteFile(ohc, ibuf, sizeof(ibuf), &n, NULL)) throw "WriteFile";
    }
    catch (const char* e){WriteFile(ehc, e, 10, &n, NULL);}

    CloseHandle(ohc);
    CloseHandle(ihc);
    CloseHandle(ehc);

    return 0;
}
```

Функция CreateFile позволяет процессу получить дескриптор для входного буфера консоли и активного буфера экрана, даже если STDIN и STDOUT были перенаправлены. Укажите значение CONOUT$ при вызове CreateFile, чтобы открыть дескриптор для активного буфера экрана консоли. CreateFile позволяет указать доступ только для чтения и записи в возвращаемом дескрипторе.

параметры консоли

```cpp
//OS_0072
#include <Windows.h>
#include <iostream>
#include <locale.h>
#include <conio.h>

int main()
{
    setlocale(LC_ALL, "");
    try
    {
        wchar_t buf[256];
        HANDLE hout = GetStdHandle(STD_OUTPUT_HANDLE);
        if (hout == INVALID_HANDLE_VALUE)                  throw L"GetStdHandle";
        std::wcout << L"-- информацмя о курсоре  -- \n";

        CONSOLE_CURSOR_INFO info;
        if (!GetConsoleCursorInfo(hout, &info))            throw L"GetConsoleCursorInfo";
        std::wcout << L"-- размер курсора %      : " << info.dwSize << L"\n";
        std::wcout << L"-- видимость курсора     : " << info.bVisible << L"\n";
        std::wcout << L"--> "; std::wcin >> buf;
```

```
        info.dwSize = 100;
        if (!SetConsoleCursorInfo(hout, &info))            throw L"SetConsoleCursorInfo";
        std::wcout << L"-- размер курсора %      : " << info.dwSize << L"\n";
        std::wcout << L"--> "; std::wcin >> buf;

        info.bVisible = FALSE;
        if (!SetConsoleCursorInfo(hout, &info))            throw L"SetConsoleCursorInfo";
        std::wcout << L"-- видимость курсора     : " << info.bVisible << L"\n";
        std::wcout << L"--> "; std::wcin >> buf;

    }
    catch (const wchar_t* e) { std::wcout << L"Error:" << e << "\n"; }
    system("pause");
    return 0;
}
```

позиции курсора `курсора`

```
//OS_0072
#include <Windows.h>
#include <iostream>
#include <locale.h>
#include <conio.h>

int main()
{
    setlocale(LC_ALL, "");
    try
    {
        wchar_t buf[256];
        COORD coord;
        HANDLE hout = GetStdHandle(STD_OUTPUT_HANDLE);
        if (hout == INVALID_HANDLE_VALUE)                throw L"GetStdHandle";
```

```
        coord.X = 0; coord.Y = 0;
        if (!SetConsoleCursorPosition(hout, coord)) throw L"GetConsoleCursorInfo";
        std::wcout << L"-- координаты курсора       :  (" << coord.X << L", "<< coord.Y <<L")\n";
        std::wcout << L"--> "; std::wcin >> buf;

        coord.X = 5; coord.Y = 5;
        if (!SetConsoleCursorPosition(hout, coord)) throw L"GetConsoleCursorInfo";
        std::wcout << L"-- координаты курсора       :  (" << coord.X << L", " << coord.Y << L")";
        std::wcout << L"--> "; std::wcin >> buf;

        coord.X = 10; coord.Y = 10;
        if (!SetConsoleCursorPosition(hout, coord)) throw L"GetConsoleCursorInfo";
        std::wcout << L"-- координаты курсора       :  (" << coord.X << L", " << coord.Y << L")";
        std::wcout << L"--> "; std::wcin >> buf;

        coord.X = 15; coord.Y = 15;
        if (!SetConsoleCursorPosition(hout, coord)) throw L"GetConsoleCursorInfo";
        std::wcout << L"-- координаты курсора       :  (" << coord.X << L", " << coord.Y << L")";
        std::wcout << L"--> "; std::wcin >> buf;
```

```
    }
    catch (const wchar_t* e) { std::wcout << L"Error:" << e << "\n"; }
    system("pause");
    return 0;
}
```

Установка атрибутов консоли

```cpp
//OS_0073
#include <Windows.h>
#include <iostream>
#include <locale.h>
#include <conio.h>

int main()
{
    setlocale(LC_ALL, "");
    try
    {
        wchar_t buf[256];
        HANDLE hout = GetStdHandle(STD_OUTPUT_HANDLE);
        HANDLE hinp = GetStdHandle(STD_INPUT_HANDLE);
        if (hout == INVALID_HANDLE_VALUE)                throw L"GetStdHandle";

        DWORD oattr = BACKGROUND_INTENSITY | FOREGROUND_GREEN|FOREGROUND_INTENSITY;
        COORD coord; coord.X = 0; coord.Y = 0;
        DWORD n = 0;
        std::wcout << L"BACKGROUND_INTENSITY\n";
        std::wcout << L"FOREGROUND_GREEN|FOREGROUND_INTENSITY\n";
        std::wcout << L"--->";
        if (!FillConsoleOutputAttribute(hout, oattr, 2000, coord, &n)) throw L"FillConsoleOutputAttribute";
        std::wcin >> buf;
    }
    catch (const wchar_t* e) { std::wcout << L"Error:" << e << "\n"; }
    return 0;
}
```

```cpp
// OS_0075
#include <Windows.h>
#include <iostream>
#include <locale.h>
#include <conio.h>

int main()
{
    setlocale(LC_ALL, "");
    try
    {
        wchar_t buf[256];
        HANDLE hout = GetStdHandle(STD_OUTPUT_HANDLE);
        if (hout == INVALID_HANDLE_VALUE)                throw L"GetStdHandle";
        DWORD tattr = BACKGROUND_INTENSITY | FOREGROUND_GREEN | FOREGROUND_RED|FOREGROUND_INTENSITY;
        if (!SetConsoleTextAttribute(hout, tattr)) throw L"SetConsoleTextAttribute";
        std::wcout << L"BACKGROUND_INTENSITY\n";
        std::wcout << L"FOREGROUND_GREEN|FOREGROUND_INTENSITY\n";
        std::wcout << L"--->";
        std::wcin >> buf;
    }
    catch (const wchar_t* e) { std::wcout << L"Error:" << e << "\n"; }
    return 0;
}
```

```cpp
// OS_0076.cpp
    #include <Windows.h>
    #include <iostream>
    #include <locale.h>
    #include <conio.h>
    int main()
    {
        setlocale(LC_ALL, "");
        try
        {
            wchar_t buf[256];
            HANDLE hout = GetStdHandle(STD_OUTPUT_HANDLE);
            if (hout == INVALID_HANDLE_VALUE)                throw L"GetStdHandle";
            DWORD tattr = BACKGROUND_INTENSITY | FOREGROUND_GREEN | FOREGROUND_RED | FOREGROUND_INTENSITY;
            if (!SetConsoleTextAttribute(hout, tattr)) throw L"SetConsoleTextAttribute";
            std::wcout << L"BACKGROUND_INTENSITY\n";
            std::wcout << L"FOREGROUND_GREEN|FOREGROUND_INTENSITY\n";
            std::wcout << L"--->";
            WORD mattr[4] = {
                                BACKGROUND_INTENSITY | BACKGROUND_GREEN | FOREGROUND_RED   | FOREGROUND_INTENSITY,
                                BACKGROUND_INTENSITY | BACKGROUND_RED   | FOREGROUND_GREEN | FOREGROUND_INTENSITY,
                                BACKGROUND_INTENSITY | BACKGROUND_BLUE  | FOREGROUND_GREEN | FOREGROUND_INTENSITY,
                                BACKGROUND_INTENSITY | BACKGROUND_GREEN | FOREGROUND_RED   | FOREGROUND_INTENSITY,
                            };
            COORD coord; coord.X = 0; coord.Y = 0;
            DWORD n = 0;
            if(!WriteConsoleOutputAttribute(hout, mattr, 4, coord, &n))throw L"WriteConsoleOutputAttribute";
            std::wcin >> buf;
        }
        catch (const wchar_t* e) { std::wcout << L"Error:" << e << "\n"; }
        return 0;
```

std::wistream std::wcin

16. Асинхронные операции ввода вывода: понятие асинхронной операции ввода/вывода, особенности программирования асинхронного ввода/вывода.

Имеется два типа синхронизации ввода - вывода (I/O) файлов:
- синхронный ввод - вывод (I/O) файла и
- асинхронный (перекрывающий(overlapped)) ввод - вывод (I/O) файла

При синхронном вводе - выводе файла поток запускает операцию ввода/вывода (I/O) и немедленно вводит ждущее состояние до тех пор, пока, запрос ввода-вывода не завершит работу.
Поток, **выполняющий асинхронный ввод - вывод файла**, отправляет запрос на ввод-вывод данных ядру. Если запрос принят ядром, поток продолжает обрабатывать другое задание до тех пор, пока ядро не подаст сигналы потоку, что операция ввода/вывода (I/O) полностью завершилась. Тогда поток прерывает работу со своим текущим заданием и обрабатывает данные от операции ввода/вывода (I/O) по мере необходимости.

В ситуациях, когда ожидается запрос на ввод-вывод, который займет большое количество времени, такое как обновление или резервное копирование большой базы данных, асинхронный ввод - вывод (I/O) как правило - хороший способ оптимизировать эффективность обработки. Однако, для относительно быстрых операций ввода/вывода (I/O), непроизводительные издержки обработки запросов ядра на ввод-вывод и сигналов ядра могут сделать асинхронный ввод - вывод (I/O) менее выгодным, особенно если должны делаться много быстрых операций ввода/вывода (I/O). В этом случае, синхронный ввод - вывод (I/O) будет лучше.

Шаги, которые необходимо выполнить для асинхронного чтения-записи (два способа, первый при единственной операции ввода/вывода, второй при нескольких):

Способ 1. http://pblog.ru/?p=74

1. при открытии файла (т.е при вызове функции CreateFile)нужно передать в параметр dwFlagsAndAttributes значение FILE_FLAG_OVERLAPPED.
2. создать (объявить) структуру _OVERLAPPED. OVERLAPPED (minwinbase.h) - Win32 apps
3. https://gist.github.com/vkocjancic/80c637d8a9cab30383a71f17f6c1bdc8
4. при вызове функций WriteFile/ReadFile: в параметр lpOverlapped нужно передать указатель на структру из пункта 2.
5. GetLastError == ERROR_IO_PENDING: если вызовем до окончания ввода-вывода;
6. WaitForSingleObject(HANLE, time); HANDLE - это HANDLE файла полученного при вызове CreateFile.
7. Продвинуть Offset, OffsetHigh.

Способ 2.

1. создать (объявить) структуру _OVERLAPPED;
2. создать событие Event с автоматическим сбросом;

3. hEvent = Event (записать событие в _OVERLAPPED);
4. WriteFile/Read lpOverlapped;
5. GetLastError == ERROR_IO_PENDING:
6. WaitForSingleObject(hEvent, time);
7. продвинуть Offset, OffsetHigh.:

Асинхронное блокирование файлов UnlockFile и UnlockFileEx

Определить состояние асинхронного ввода/вывода GetOverlappedResult

Отмена операции ввода/вывода. CancelTo

Проверить завершение асинхронной операции. HasOverlappedIoCompleted

Функции завершения ReadFileEx/WriteFileEx в них можно передать callback который выполнится после окончания операции ввода/вывода.

Порты завершения — это специальный механизм, который позволяет обрабатывать результаты асинхронного ввода-вывода. Их преимущество в том что создается пул потоков которые будут обрабатывать результаты ввода-вывода. Целесообразно применять в приложениях в которых есть частые асинхронные операции (не нужно создавать поток на каждую операцию).

Функции для работы с портами:

CreateIoCompletionPort - создать порт завершения

GetQueuedCompletionStatus - получить пакет порта завершения из очереди.

PostQueuedCompletionStatus - послать пакет в очередь порта завершения.