**2.Понятие WinAPI и его применение при разработке прикладных программ.**

**Определение с Майкрософт**

с помощью API Windows можно разрабатывать приложения, которые успешно выполняются во всех версиях Windows, при этом используя преимущества функций и возможностей, уникальных для каждой версии. (Обратите внимание, что раньше это называлось API Win32. имя Windows API более точно отражает его корни в 16-разрядной Windows и поддерживает 64-разрядную Windows.)

Win32 API (также называемый Windows API) — это исходная платформа для собственных Windows-приложений на языке C/C++, которым требуется прямой доступ к Windows и оборудованию. Он предоставляет возможности разработки первого класса, не зависящие от управляемой среды выполнения, такой как .NET и WinRT (для приложений UWP для Windows 10). Благодаря этому API Win32 стает оптимальной платформой для приложений, которым требуется самый высокий уровень производительности и прямой доступ к системному оборудованию.

**Определение с Вики**

**Windows API** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *application programming interfaces*) — общее наименование набора базовых функций [интерфейсов программирования приложений](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9) операционных систем семейств [*Microsoft Windows*](https://ru.wikipedia.org/wiki/Microsoft_Windows) корпорации «[Майкрософт](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%B9%D0%BA%D1%80%D0%BE%D1%81%D0%BE%D1%84%D1%82)». Предоставляет прямой способ взаимодействия приложений пользователя с операционной системой *Windows*. Для создания программ, использующих *Windows API*, корпорация «[Майкрософт](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%B9%D0%BA%D1%80%D0%BE%D1%81%D0%BE%D1%84%D1%82)» выпускает [комплект разработчика программного обеспечения](https://ru.wikipedia.org/wiki/SDK), который называется [*Platform SDK*](https://ru.wikipedia.org/wiki/Platform_SDK) и содержит документацию, набор [библиотек](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), утилит и других инструментальных средств для разработки.

**Источник Препода**

Тип возвращаемого значения функции WinMain() определен как WINAPI. Фактически, WINAPI представляет собой один из типов, описанных в заголовочном файле windows.h посредством директивы #define. Его реальное значение зависит от платформы, для которой разрабатывается приложение, используемого компилятора и ряда других факторов. В приложениях, создаваемых для платформы Intel x86, значением WINAPI является \_\_stdcall, что означает функцию, использующую стандартный тип вызова (параметры функции кладутся в стек справа налево, для идентификаторов не генерируются знаки подчеркивания и т.п.). Рекомендуется использовать именно WINAPI как тип возвращаемого значения для WinMain(), в противном случае возможны проблемы при компиляции исходных текстов программ, так как реальное значение WINAPI может измениться при переходе к новой версии Windows.

**Пример программы**

//-------------- Простейшее приложение Windows --------------

#include <windows.h>

int WINAPI WinMain (HINSTANCE, HINSTANCE, LPSTR, int)

{

return MessageBox (NULL, "Hello, Windows!",

"First Windows Program",

MB\_OK | MB\_ICONEXCLAMATION);

}

Как видно из листинга, функция WinMain() просто возвращает результат выполнения функции MessageBox(). MessageBox() является одной из стандартных функций, входящих в состав Win32 API, поэтому сведения о ее параметрах, можно найти в соответствующей документации (в частности, в файлах справки по C++ Builder/Delphi, находящихся в папке Help/MS SDK Help Files).

Вообще говоря, функция WinMain() может содержать любой легальный с точки зрения операционной системы код, например, вызовы каких–либо функций Win32 API. Иначе говоря, структура приложения Win32 определяется программистом, и может быть вполне произвольной. Так, к примеру, приложение может содержать функции, выполняющие некоторые действия, а вызовы этих функций могут производиться в теле функции WinMain(), либо WinMain() может включать в себя полностью весь код приложения. ***Каких-либо специальных ограничений на код, находящийся в теле функции WinMain(), при программировании в Win32 не существует!***. Единственным требованием к приложению Windows является обязательное наличие WinMain(). Примером программ подобного рода являются консольные приложения Win32, т.е. полностью 32-разрядные приложения, которым доступны все возможности Win32 API, но которые для операций ввода/вывода используют не окна Windows, а текстовое окно (консоль).

**3.Процессы и потоки.**

Каждый процесс предоставляет ресурсы, необходимые для выполнения программы. Процесс содержит виртуальное адресное пространство, исполняемый код, открытые дескрипторы для системных объектов, контекст безопасности, уникальный идентификатор процесса, переменные среды, класс приоритета, минимальный и максимальный размеры рабочего множества и, по крайней мере, один поток выполнения. Каждый процесс запускается с одним потоком, который часто называется основным потоком, но может создавать дополнительные потоки из любого потока.

Поток — это сущность внутри процесса, для которой можно запланировать выполнение. Все потоки процесса совместно используют свои виртуальные адресные пространства и системные ресурсы. Кроме того, каждый поток поддерживает обработчики исключений, приоритет планирования, локальное хранилище потока, уникальный идентификатор потока и набор структур, которые система будет использовать для сохранения контекста потока, пока он не будет запланирован. Контекст потока включает набор регистров компьютера, стек ядра, блок среды потока и пользовательский стек в адресном пространстве процесса потока. Потоки также могут иметь собственный контекст безопасности, который можно использовать для олицетворения клиентов.

Microsoft Windows поддерживает вытеснение по нескольким задачам, что создает воздействие одновременного выполнения нескольких потоков из нескольких процессов. На многопроцессорном компьютере система может одновременно выполнять столько потоков, сколько процессоров на компьютере.

Объект задания позволяет управлять группами процессов как единым целым. Объекты заданий являются намаблеными, защищаемыми объектами, совместно используемые объекты, которые управляют атрибутами связанных с ними процессов. Операции, выполняемые над объектом задания, влияют на все процессы, связанные с объектом Job.

Приложение может использовать пул потоков , чтобы уменьшить число потоков приложения и обеспечить управление рабочими потоками. Приложения могут ставить в очередь рабочие элементы, связывать работу с ожидающими дескрипторами, автоматически ставить в очередь на основе таймера и выполнять привязку с помощью операций ввода-вывода.

Планирование пользовательского режима (UMS) — это упрощенный механизм, с помощью которого приложения могут планировать собственные потоки. Приложение может переключаться между UMS-потоками в пользовательском режиме без использования [планировщика системы](https://docs.microsoft.com/ru-ru/windows/win32/procthread/scheduling) и повторного получения управления процессором, если поток UMS блокируется в ядре. Каждый поток UMS имеет собственный контекст потока вместо того, чтобы совместно использовать контекст потока одного потока. Возможность переключения между потоками в пользовательском режиме делает UMS более эффективным, чем пулы потоков для кратковременных рабочих элементов, требующих небольшого количества системных вызовов.

Волокно — это единица выполнения, которая должна быть запланирована приложением вручную. Волокна выполняются в контексте потоков, планирующих их. Каждый поток может запланировать несколько волокон. Как правило, волокна не предоставляют преимуществ по сравнению с хорошо спроектированным многопоточным приложением. Однако использование волокон может упростить перенос приложений, разработанных для планирования собственных потоков

**4.Планирование в системах с одним процессором.**

Эта ограниченная ситуация позволяет определить и прояснить многие вопросы, связанные с планированием. Глава начинается с рассмотрения трех типов планирования: долгосрочного, среднесрочного и краткосрочного. В основном материал главы посвящен вопросам краткосрочного планирования. Будут представлены в сравнении различные алгоритмы планирования.

Планирование оказывает большое влияние на производительность системы, поскольку именно оно определяет, какой процесс будет выполняться, а какой — ожидать выполнения

**Долгосрочный планировщик**

Это также называется **планировщиком заданий** . Долгосрочный планировщик определяет, какие программы допущены в систему для обработки. Он выбирает процессы из очереди и загружает их в память для выполнения. Процесс загружается в память для планирования ЦП.

Основной целью планировщика заданий является обеспечение сбалансированного сочетания заданий, таких как привязка ввода-вывода и привязка процессора. Он также контролирует степень мультипрограммирования. Если степень мультипрограммирования стабильна, то средняя скорость создания процесса должна быть равна средней скорости вылета процессов, покидающих систему.

В некоторых системах долгосрочный планировщик может быть недоступен или минимален. Операционные системы с разделением времени не имеют долгосрочного планировщика. Когда процесс меняет состояние с нового на готовое, тогда используется долгосрочный планировщик.

**Краткосрочный планировщик**

Он также называется **планировщиком ЦП** . Его основная цель — повысить производительность системы в соответствии с выбранным набором критериев. Это изменение состояния готовности в рабочее состояние процесса. Планировщик ЦП выбирает процесс среди процессов, готовых к выполнению, и выделяет ЦП одному из них.

Краткосрочные планировщики, также известные как диспетчеры, принимают решение о том, какой процесс выполнять дальше. Краткосрочные планировщики быстрее, чем долгосрочные.

**Среднесрочный планировщик**

Среднесрочное планирование является частью **обмена** . Удаляет процессы из памяти. Это уменьшает степень мультипрограммирования. Среднесрочный планировщик отвечает за обработку замененных процессов.

Работающий процесс может быть приостановлен, если он сделает запрос ввода-вывода. Приостановленные процессы не могут достичь прогресса. В этом случае, чтобы удалить процесс из памяти и освободить место для других процессов, приостановленный процесс перемещается во вторичное хранилище. Этот процесс называется **обменом** , а процесс называется обмен или выкатывание. Обмен может быть необходимым для улучшения технологического процесса.

**Сравнение среди Планировщика**

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Долгосрочный планировщик** | **Краткосрочный планировщик** | **Среднесрочный планировщик** |
| 1 | Это планировщик работы | Это планировщик процессора | Это планировщик обмена процессами. |
| 2 | Скорость меньше, чем краткосрочный планировщик | Скорость самая быстрая среди двух других | Скорость находится между краткосрочным и долгосрочным планировщиком. |
| 3 | Контролирует степень мультипрограммирования | Это обеспечивает меньший контроль над степенью мультипрограммирования | Это уменьшает степень мультипрограммирования. |
| 4 | Она практически отсутствует или минимальна по времени | Это также минимальная система разделения времени | Это часть систем разделения времени. |
| 5 | Он выбирает процессы из пула и загружает их в память для выполнения | Он выбирает те процессы, которые готовы выполнить | Это может повторно ввести процесс в память, и выполнение может быть продолжено. |

**Типы многозадачности**

* Невытесняющие. В этом случае находящийся в состоянии выполнения процесс продолжает выполнение до тех пор, пока он не завершится или пока не окажется в заблокированном состоянии ожидания завершения операции ввода-вывода или запроса некоторого системного сервиса.
* Вытесняющие. Выполняющийся в настоящий момент процесс может быть прерван и переведен операционной системой в состояние готовности к выполнению. Решение о вытеснении может приниматься при запуске нового процесса по прерыванию, которое переводит заблокированный процесс в состояние готовности к выполнению, или периодически — на основе прерываний таймера.

**Алгоритмы планирования**

**Критерии краткосрочного планирования**

Основная цель краткосрочного планирования состоит в распределении процессорного времени таким образом, чтобы оптимизировать один или несколько аспектов поведения системы.

**Использование приоритетов**

Во многих системах каждому процессу присвоен некоторый приоритет, и планировщик всегда должен среди процессов выбирать тот, у которого приоритет наибольший.

При выборе процесса планировщик начинает с очереди процессов с наивысшим приоритетом (RQ0). Если в очереди имеются один или несколько процессов, процесс для работы выбирается с использованием некоторой стратегии планирования. Если очередь RQ0 пуста, рассматривается очередь RQ1 и т.д.

Одна из основных проблем в такой чисто приоритетной схеме планирования состоит в том, что процессы с низким приоритетом могут оказаться в состоянии голодания. Это будет происходить при постоянном поступлении новых готовых к выполнению процессов с высоким приоритетом. Если такое поведение нежела­тельно, приоритет процесса может снижаться при его выполнении (пример такой стратегии планирования будет приведен позже).

**Первым поступил — первым обслужен**

Простейшая стратегия планирования "первым поступил — первым обслужен" (first-come-first-served — FCFS) известна также как схема "первым пришел — первым вышел", или схема строгой очередности. Как только процесс становится готовым к выполнению, он присоединяется к очереди готовых про­цессов. При прекращении выполнения текущего процесса для выполнения выбирается процесс, который находился в очереди дольше других.



**Планирование в многопроцессорных системах.**

В многопроцессорном планировании доступно **несколько процессоров,** и, следовательно, становится возможным **распределение нагрузки** . Однако планирование с несколькими процессорами является более **сложным** по сравнению с планированием с одним процессором.

### Подходы к многопроцессорному планированию —

Один из подходов заключается в том, что все решения по планированию и обработке ввода-вывода обрабатываются одним процессором, который называется **главным сервером,** а другие процессоры выполняют только **пользовательский код** . Это просто и снижает необходимость обмена данными. Весь этот сценарий называется **асимметричной многопроцессорной обработкой** .

Во втором подходе используется **симметричная многопроцессорная обработка, при** которой каждый процессор **самопланируется** . Все процессы могут находиться в общей очереди готовности или у каждого процессора может быть своя собственная очередь для процессов готовности. Планирование продолжается далее, когда планировщик для каждого процессора проверяет готовую очередь и выбирает процесс для выполнения.

### Процессор Affinity —

Сходство процессора означает, что процесс имеет **сходство** с процессором, на котором он в данный момент работает.  
Когда процесс выполняется на определенном процессоре, это влияет на кэш-память. Данные, к которым последний раз обращались к процессу, заполняют кэш-память процессора, и в результате последовательный доступ к памяти процесса часто выполняется в кэш-памяти. Теперь, если процесс мигрирует на другой процессор, содержимое кэш-памяти должно быть признано недействительным для первого процессора, а кэш-память для второго процессора должна быть заполнена заново. Из-за высокой стоимости аннулирования и повторного заполнения кэшей большинство систем SMP (симметричная многопроцессорная обработка) стараются избежать миграции процессов с одного процессора на другой и стараются поддерживать процесс на том же процессоре. Это называется **AFFINITY ПРОЦЕССОРА** .

Существует два типа привязки процессора:

1. **Мягкое сходство.** Если в операционной системе есть политика, направленная на то, чтобы попытаться сохранить процесс на том же процессоре, но не гарантирует, что он это сделает, такая ситуация называется мягким сходством.
2. **Hard Affinity —** некоторые системы, такие как Linux, также предоставляют некоторые системные вызовы, которые поддерживают Hard Affinity, что позволяет процессу мигрировать между процессорами.

### Балансировка нагрузки —

Балансировка нагрузки — это **явление,** обеспечивающее равномерное **распределение** **рабочей нагрузки** по всем процессорам в системе SMP. Балансировка нагрузки необходима только в системах, где каждый процессор имеет собственную частную очередь процессов, которые могут выполняться В SMP (симметричная многопроцессорная обработка) важно поддерживать сбалансированную рабочую нагрузку между всеми процессорами, чтобы в полной мере использовать преимущества наличия более чем одного процессора, в противном случае один или несколько процессоров будут простаивать, в то время как другие процессоры имеют высокие рабочие нагрузки вместе со списками процессоров, ожидающих ПРОЦЕССОР.

Существует два основных подхода к балансировке нагрузки:

1. **Push-миграция — при** принудительной миграции задача обычно проверяет нагрузку на каждый процессор, и если она обнаруживает дисбаланс, то она равномерно распределяет нагрузку на каждый процессор, перемещая процессы из перегруженных в незанятые или менее загруженные процессоры.
2. **Pull Migration —** Pull Migration происходит, когда неработающий процессор извлекает ожидающую задачу из занятого процессора для ее выполнения.

### Многоядерные процессоры —

В многоядерных процессорах **несколько процессорных** ядер размещаются на одном физическом чипе. Каждое ядро имеет регистр, установленный для поддержания его архитектурного состояния и, таким образом, представляется операционной системе как отдельный физический процессор. **Системы SMP** , использующие многоядерные процессоры, работают быстрее и потребляют **меньше энергии,** чем системы, в которых каждый процессор имеет свой собственный физический чип.

Однако многоядерные процессоры могут **усложнить** задачи планирования. Когда процессор обращается к памяти, он тратит значительное количество времени на ожидание, когда данные станут доступными. Эта ситуация называется **ПАМЯТЬ СТОЛ** . Это происходит по разным причинам, например, из-за отсутствия кэша, который обращается к данным, которых нет в кэш-памяти. В таких случаях процессор может тратить до пятидесяти процентов своего времени в ожидании получения данных из памяти. Чтобы решить эту проблему, в последних разработках аппаратного обеспечения реализованы многопоточные процессорные ядра, в которых каждому ядру назначены два или более аппаратных потока. Поэтому, если один поток останавливается во время ожидания памяти, ядро может переключиться на другой поток.

Есть два способа многопоточности процессора:

1. **Грубая многопоточность — при** крупнозернистой многопоточности поток выполняется на процессоре до тех пор, пока не произойдет событие с большой задержкой, такое как остановка памяти, из-за задержки, вызванной событием с большой задержкой, процессор должен переключиться на другой поток, чтобы начать выполнение. Стоимость переключения между потоками высока, поскольку конвейер команд должен быть прерван до того, как другой поток сможет начать выполнение на ядре процессора. Как только этот новый поток начинает выполнение, он начинает заполнять конвейер своими инструкциями.
2. **Мелкозернистая Многопоточность —** Это многопоточность переключение между потоками на гораздо более точном уровне в основном на границе цикла инструкции. Архитектурный дизайн мелкозернистых систем включает логику для переключения потоков, в результате чего стоимость переключения между потоками невелика.

### Виртуализация и многопоточность —

В этом типе **многопроцессорного** планирования даже система с одним ЦП действует как многопроцессорная система. В системе с виртуализацией виртуализация представляет один или несколько виртуальных ЦП каждой виртуальной машине, работающей в системе, а затем планирует использование физического ЦП среди виртуальных машин. Большинство виртуализированных сред имеют одну хост-операционную систему и множество гостевых операционных систем. Операционная система хоста создает виртуальные машины и управляет ими. На каждой виртуальной машине установлена гостевая операционная система, и в этом госте запускаются приложения. Каждая гостевая операционная система может быть назначена для конкретных случаев использования, приложений или пользователей, включая разделение времени или даже работу в реальном времени. Любой алгоритм планирования гостевой операционной системы, который предполагает определенный прогресс в заданный промежуток времени, будет подвержен негативному влиянию виртуализации. Операционная система с разделением времени пытается выделить 100 миллисекунд каждому интервалу времени, чтобы предоставить пользователям разумное время отклика. Заданный интервал времени в 100 миллисекунд может занимать намного более 100 миллисекунд виртуального процессорного времени. В зависимости от того, насколько занята система, временной интервал может занять секунду или более, что приводит к очень плохому времени отклика для пользователей, вошедших в эту виртуальную машину. Общий эффект такого многоуровневого планирования заключается в том, что отдельные виртуализированные операционные системы получают только часть доступных циклов ЦП, даже если они считают, что они получают все циклы и что они планируют все эти циклы. Как правило, время суток часы в виртуальных машинах неверны, потому что таймеры запускаются не дольше, чем на выделенных процессорах.

Таким образом, **виртуализация** может отменить хорошие усилия алгоритма планирования операционных систем в виртуальных машинах.

**Файлы с презентациями можно тоже посмотреть.**