Лекция 6:

**Процессы и потоки**

**Основные понятия**

*Программа* (program) – это последовательность команд, реализующая *алгоритм* решения задачи. *Программа* может быть записана на языке программирования (например, на *Pascal*, С++, BASIC); в этом случае она хранится на диске в виде текстового файла с расширением, соответствующим языку программирования (например, .PAS, .*CPP*, .VB). Также *программа* может быть представлена при помощи машинных команд; тогда она хранится на диске в виде двоичного исполняемого файла (*executable file*), чаще всего с расширением .EXE. Исполняемый *файл* генерируется из текста программы при компиляции.

Процесс (process) – это *программа* (пользовательская или системная) в ходе выполнения.

В современных операционных системах процесс представляет собой *объект* – структуру данных, содержащую информацию, необходимую для выполнения программы. *Объект* "Процесс" создается в момент запуска программы (например, *пользователь* дважды щелкает мышью на исполняемом файле) и уничтожается при завершении программы.

Если *операционная система* умеет запускать в одно и то же время несколько процессов, она называется многозадачной (*multitasking*) (пример – *Windows*), иначе – однозадачной (пример – MS *DOS*).

Процесс может содержать один или несколько потоков (*thread*) – объектов, которым *операционная система* предоставляет *процессорное время*. Сам по себе процесс не выполняется – выполняются его потоки. Таким образом, машинные команды, записанные в исполняемом файле, выполняются на процессоре в составе потока. Если потоков несколько, они могут выполняться одновременно.

Замечание. "Одновременное" (или "параллельное") выполнение потоков подразумевает одну из двух принципиально разных ситуаций, зависящих от количества процессоров (ядер) на компьютере. В том случае, если имеется всего один *процессор* с одним ядром, в один момент времени может выполняться только один *поток*. Однако *операционная система* может быстро переключать *процессор* с выполнения одного потока на другой и вследствие высокой частоты процессоров у пользователя возникает иллюзия одновременной работы нескольких программ. Такая ситуация называется псевдопараллельное выполнение потоков. Если в компьютере установлен многоядерный *процессор* или количество процессоров больше одного, то возможно истинно параллельное или просто параллельное выполнение потоков.

Операционные системы, поддерживающие несколько процессоров, называются многопроцессорными.

Если система допускает наличие нескольких потоков в одном процессе, она называется многопоточной (*multithreading*).

Многопоточность – это средство распараллеливания действий внутри процесса.

Примеры.

1. В современных браузерах каждой вкладке соответствует свой поток.
2. В текстовом редакторе один поток может управлять вводом текста, второй – проверять орфографию, третий – выводить документ на принтер.
3. В компьютерной игре-стратегии за обработку действий каждого игрока отвечает отдельный поток.

Каждый процесс имеет свое собственное *виртуальное* *адресное пространство* (см. лекцию 11 "*Управление памятью*"), в котором независимо друг от друга работают все потоки процесса. Например, *поток* 1 может записывать данные в ячейку с адресом 100, а *поток* 2 читать данные из ячейки с адресом 101.

Замечание. Конечно, если два (или более) потоков захотят записать что-то свое в одну и ту же ячейку, возникнет неопределенность – кто раньше? Это одна из подзадач обширной проблемы синхронизации.

Каждый *поток* имеет свою область памяти – *стек* (stack), в которой хранятся, например, локальные переменные и параметры, передаваемые в функции. Кроме того, во *время выполнения* потока изменяются значения регистров процессора, которые при переключении на другой *поток* должны быть сохранены. Эта *информация* является частью контекста потока, поэтому при переключении потоков происходит переключение их контекстов (сохранение в *память* одного и *загрузка* другого).

**Структуры данных для процессов и потоков**

В WRK за *управление процессами* отвечает *диспетчер* процессов (base\ntos\ps), а многие важные структуры данных описаны в заголовочных файлах base\ntos\inc\ps.h и base\ntos\inc\ke.h.

Процесс в *Windows* описывается структурой данных EPROCESS [[5](https://www.intuit.ru/studies/courses/10471/1078/literature#literature.5)]. Эта структура в WRK содержится в файле base\ntos\inc\ps.h (строка 238). Рассмотрим некоторые её поля.

* Pcb (Process Control Block – блок управления процессом) – представляет собой структуру KPROCESS, хранящую данные, необходимые для планирования потоков, в том числе указатель на список потоков процесса (файл base\ntos\inc\ke.h, строка 944).
* CreateTime и ExitTime – время создания и завершения процесса.
* UniqueProcessId – уникальный идентификатор процесса.
* ActiveProcessLinks – элемент двунаправленного списка (тип LIST\_ENTRY), содержащего активные процессы.
* QuotaUsage, QuotaPeak, CommitCharge – квоты (ограничения) на используемую память.
* ObjectTable – таблица дескрипторов процесса.
* Token – маркер доступа.
* ImageFileName – имя исполняемого файла.
* ThreadListHead – двунаправленный список потоков процесса.
* Peb (Process Environment Block – блок переменных окружения процесса) – информация об образе исполняемого файла (файл public\sdk\inc\pebteb.h, строка 75).
* PriorityClass – класс приоритета процесса (см. лекцию 9 "Планирование потоков").

Структура для потока в *Windows* называется ETHREAD и описывается в файле base\ntos\inc\ps.h (строка 545). Её основные поля следующие:

* Tcb (Thread Control Block – блок управления потоком) – поле, которое является структурой типа KTHREAD (файл base\ntos\inc\ke.h, строка 1069) и необходимо для планирования потоков.
* CreateTime и ExitTime – время создания и завершения потока.
* Cid – структура типа CLIENT\_ID, включающая два поля – идентификатор процесса-владельца данного потока и идентификатор самого потока.
* ThreadsProcess – указатель на структуру EPROCESS процесса-владельца.
* StartAddress – адрес системной стартовой функции потока. При создании потока сначала вызывается системная стартовая функция, которая запускает пользовательскую стартовую функцию.
* Win32StartAddress – адрес пользовательской стартовой функции.

**Создание процесса**

Процессы создаются либо пользователем, либо другим процессом, либо автоматически при загрузке операционной системы.

Процесс, создавший другой процесс, называется родителем, а созданный процесс – потомком. Таким образом, формируется *иерархия* процессов.

Любой процесс начинает свою работу с основного (main), или первичного, потока, который может запускать (порождать) другие потоки – так образуется *иерархия* потоков.

В *Windows* для создания процессов применяется одна из следующих WinAPI-функций: CreateProcess, CreateProcessAsUser, CreateProcessWithTokenW, CreateProcessWithLogonW [[10](https://www.intuit.ru/studies/courses/10471/1078/literature#literature.10)]. Далее при описании будем использовать функцию CreateProcess.

Создание процессов в *Windows* включает 7 этапов [[5](https://www.intuit.ru/studies/courses/10471/1078/literature#literature.5); [2](https://www.intuit.ru/studies/courses/10471/1078/literature#literature.2)].

**1. Проверка и преобразование параметров.**

Параметры функции CreateProcess проверяются на *корректность* и преобразуются к внутреннему формату системы.

**2. Открытие исполняемого файла.**

Происходит *поиск* файла, который содержит запускаемую программу. Обычно это *файл* с расширением .EXE, но могут быть также расширения .*COM*, .*PIF*, .BAT, .*CMD*. Определяется тип исполняемого файла:

* Windows приложение (.EXE) – продолжается нормальное создание процесса;
* приложение MS-DOS или Win16 (.EXE, .COM, .PIF) – запускается образ поддержки Ntvdm.exe;
* командный файл (.BAT, .CMD) – запускается образ поддержки Cmd.exe;
* приложение POSIX – запускается образ поддержки Posix.exe.

**3. Создание объекта "Процесс".**

Формируются структуры данных EPROCESS, KPROCESS, PEB, инициализируется *адресное пространство* процесса. Для этого вызывается системная *функция* NtCreateProcess (*файл* base\ntos\ps\create.c, строка 826), которая затем вызывает функцию NtCreateProcessEx (тот же *файл*, строка 884), а та, в свою *очередь*, функцию PspCreateProcess (тот же *файл*, строка 1016).

Замечание. Начиная с *Windows Vista* при создании процесса вызов нескольких функций (NtCreateProcess, NtWriteVirtualMemory, NtCreateThread) заменен вызовом одной функции NtCreateUserProcess.

Рассмотрим некоторые важные действия, выполняемые функцией PspCreateProcess.

* Если в параметрах функции PspCreateProcess указан процесс-родитель:
  + по его дескриптору определяется указатель на объект EPROCESS (функция ObReferenceObjectByHandle, строка 1076);
  + наследуется от процесса родителя маска привязки к процессорам (Affinity, строка 1092).
* Устанавливается минимальный и максимальный размеры рабочего набора (WorkingSetMinimum = 20 МБ и WorkingSetMaximum = 45 МБ, строки 1093 и 1094, см. лекцию 11 "Управление памятью").
* Создается объект "Процесс" (структура EPROCESS) при помощи функции ObCreateObject (строка 1108).
* Инициализируется двунаправленный список потоков при помощи функции InitializeListHead (строка 1134).
* Копируется таблица дескрипторов родительского процесса (строка 1270).
* Создается структура KPROCESS при помощи функции KeInitializeProcess (строка 1289).
* Маркер доступа и другие данные, связанные с безопасностью (см. лекцию 13 "Безопасность", копируются из родительского процесса (функция PspInitializeProcessSecurity, строка 1302).
* Устанавливается приоритет процесса, равный Normal; однако, если приоритет родительского процесса был Idle или Below Normal, то данный приоритет наследуется (строки 1307–1312, см. лекцию 9 "Планирование потоков").
* Инициализируется адресное пространство процесса (строки 1337–1476).
* Генерируется уникальный идентификатор процесса (функция ExCreateHandle) и сохраняется в поле UniqueProcessId структуры EPROCESS (строки 1482–1488).
* Создается блок PEB и записывается в соответствующее поле структуры EPROCESS (строки 1550–1607).
* Созданный объект вставляется в хвост двунаправленного списка всех процессов (строки 1613–1615) и в таблицу дескрипторов (строки 1639–1644). Первая вставка обеспечивает доступ к процессу по имени, вторая – по ID.
* Определяется время создания процесса (функция KeQuerySystemTime) и записывается в поле CreateTime структуры EPROCESS (строка 1733).

**4. Создание основного потока.**

Формируется *структура данных* ETHREAD, *стек* и *контекст* потока, генерируется *идентификатор* потока. *Поток* создается при помощи функции NtCreateThread, определенной в файле base\ntos\ps\create.c, (строка 117), которая вызывает функцию PspCreateThread (тот же *файл*, строка 295). При этом выполняются следующие действия:

* создается объект ETHREAD (строка 370).
* Заполняются поля структуры ETHREAD, связанные с процессом-владельцем, – указатель на структуру EPROCESS (ThreadsProcess) и идентификатор процесса (Cid.UniqueProcess) (строки 396 и 398).
* Генерируется уникальный идентификатор потока (функция ExCreateHandle) и сохраняется в поле Cid.UniqueThread структуры EPROCESS (строки 400–402).
* Заполняются стартовые адреса потока, системный (StartAddress) и пользовательский (Win32StartAddress) (строки 468-476).
* Инициализируются поля структуры KTHREAD при помощи вызова функции KeInitThread (строки 490–498 для потока пользовательского режима и 514–522 для потока режима ядра).
* Функция KeStartThread заполняет остальные поля структуры ETHREAD и вставляет поток в список потоков процесса (строка 564).
* Если при вызове функции PspCreateThread установлен флаг CreateSuspended ("Приостановлен") поток переводится в состояние ожидания (функция KeSuspendThread, строка 660); иначе вызывается функция KeReadyThread (строка 809), которая ставит поток в очередь готовых к выполнению потоков (см. лекцию 9 "Планирование потоков").

**5. Уведомление подсистемы Windows.**

Подсистеме *Windows* отправляется сообщение о вновь созданных процессе и его основном потоке, в которое входят их дескрипторы, идентификаторы и другая *информация*. Подсистема *Windows* добавляет новый процесс в общий *список* всех процессов и готовится к запуску основного потока.

**6. Запуск основного потока.**

Основной *поток* стартует, но начинают выполняться системные функции, завершающие *создание процесса* – осуществляется его *инициализация*.

**7. Инициализация процесса.**

* Проверяется, не запущен ли процесс в отладочном режиме;
* проверяется, следует ли производить предвыборку блоков памяти (тех участков памяти, которые при прошлом запуске использовались в течение первых 10 секунд работы процесса);
* инициализируются необходимые компоненты и структуры данных процесса, например, диспетчер кучи;
* загружаются динамически подключаемые библиотеки (DLL – Dynamic Link Library);
* начинается выполнение стартовой функции потока.

**Резюме**

В этой лекции введены понятия "процесса" и "потока". Рассмотрены структуры данных, представляющие в операционной системе процесс (EPROCESS) и *поток* (ETHREAD). Описан ход создания процесса с использованием структур данных и функций *Windows* Research *Kernel*.

Лекция 7:

**Планирование потоков**

Если *операционная система* поддерживает многопоточность, она может распределять *процессорное время* либо между процессами, либо между потоками. В операционной системе *Windows* *процессор* предоставляется потокам, иначе говоря, осуществляется планирование на уровне потоков.

Таким образом, если один процесс имеет пять потоков, а второй – десять, то первый процесс будет занимать *процессор* в два раза больше времени, чем второй (при условии, конечно, что все потоки имеют равный приоритет и выполняют примерно одинаковую работу).

**Алгоритмы планирования**

Существуют разные алгоритмы планирования. Рассмотрим основные виды.

**1. Вытесняющие/невытесняющие алгоритмы.**

В случае вытесняющего алгоритма операционная система в любой момент времени может прервать выполнение текущего потока и переключить процессор на другой поток. В невытесняющих алгоритмах поток, которому предоставлен процессор, только сам решает, когда передать управление операционной системе.

**2. Алгоритмы с квантованием.**

Каждому потоку предоставляется квант времени, в течение которого поток может выполняться на процессоре. По истечении кванта операционная система переключает процессор на следующий поток в очереди. Квант обычно равен целому числу интервалов системного таймера1.

**3. Алгоритмы с приоритетами.**

Каждому потоку назначается приоритет (priority) – целое число, обозначающее степень привилегированности потока. Операционная система при наличии нескольких готовых к выполнению потоков выбирает из них поток с наибольшим приоритетом.

В Windows реализован смешанный алгоритм планирования – вытесняющий, на основе квантования и приоритетов.

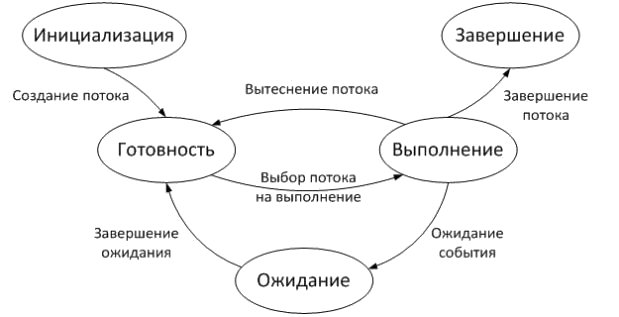
**Состояния потоков**

За время своего существования поток может находиться в нескольких состояниях. Перечислим основные состояния:

1. Готовность (Ready) – поток готов к выполнению и ждет своей очереди занять процессор.
2. Выполнение (Running) – поток выполняется на процессоре.
3. Ожидание (Waiting) – поток не может выполняться, поскольку ждет наступление некоторого события (например, завершения операции ввода-вывода или сообщения от другого потока)

Кроме основных существует ещё несколько состояний – Инициализация (Init), Завершение (Terminate), Состояние простоя (Standby), Переходное состояние (Transition), Состояние отложенной готовности (Deferred ready). Подробнее о них можно узнать в [[5](https://www.intuit.ru/studies/courses/10471/1078/literature#literature.5); [2](https://www.intuit.ru/studies/courses/10471/1078/literature#literature.2)].

На [рис.9.1](https://www.intuit.ru/studies/courses/10471/1078/lecture/16576?page=1#image.9.1) показаны основные состояния потока, возможные переходы между состояниями и условия переходов.



**Рис. 9.1.**Состояния потока

**Кванты**

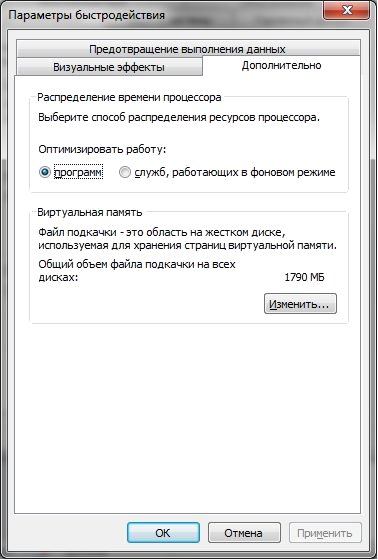
В Windows имеется два базовых размера кванта – 2 интервала системного таймера и 12 интервалов. Если квант времени короткий, то потоки будут переключаться быстрее и "отзывчивость" (responsiveness) системы улучшится – это важное свойство для пользователя, поэтому в клиентских системах Windows по умолчанию используются короткие кванты. При этом производительность системы в целом снижается, поскольку потоки не будут успевать выполнять свои задачи в течение выделенного кванта, а частые переключения создадут высокие накладные расходы (служебные операции системы при смене потоков). Вследствие этого в серверных версиях Windows по умолчанию применяются длинные кванты.

Длительность интервала системного таймера (в сотнях наносекунд) хранится в переменной KeMaximumIncrement (для x86 – файл base\ntos\ex\i386\splocks.asm, строка 140; для x64 – файл base\ntos\ex\amd64\hifreqlk.asm, строка 147) и устанавливается функцией KeSetTimeIncrement (файл base\ntos\ke\miscc.c, строка 711 на основе значения, предоставляемого HAL.

Каждый процесс хранит величину кванта в поле QuantumReset структуры KPROCESS (файл base\ntos\inc\ke.h, строка 1029). Значение в этом поле равно количеству интервалов таймера, умноженному на 3. Например, для длинных квантов (12 интервалов) значение QuantumReset будет равно 36. Таким образом, при каждом срабатывании таймера (возникает прерывание) система уменьшает квант выполняющегося потока на 3 единицы.

Умножение на три введено для того чтобы можно было в разной степени уменьшать квант в двух различных ситуациях – срабатывании таймера (квант уменьшается на 3 единицы) и выходе из состояния ожидания (квант уменьшается на единицу). Уменьшение кванта при выходе потока из состояния ожидания применяется чтобы избежать ситуации бесконечно выполняющегося потока: если при каждом срабатывании таймера поток находится в состоянии ожидания, а при выходе из ожидания значение кванта не изменяется, то теоретически поток может выполняться бесконечно. Поэтому при выходе из состояния ожидания текущее значение его кванта уменьшается на единицу.

Значение кванта может быть изменено пользователем. Например, на Windows 7 нужно проделать следующее: Компьютер – Свойства – Дополнительные параметры системы – вкладка "Дополнительно" – раздел "Быстродействие" – Параметры – вкладка "Дополнительно" – раздел "Распределение времени процессора". Можно выбрать короткие кванты ("Оптимизировать работу программ") или длинные ("Оптимизировать работу служб, работающих в фоновом режиме") ([рис.9.2](https://www.intuit.ru/studies/courses/10471/1078/lecture/16576?page=1#image.9.2)).



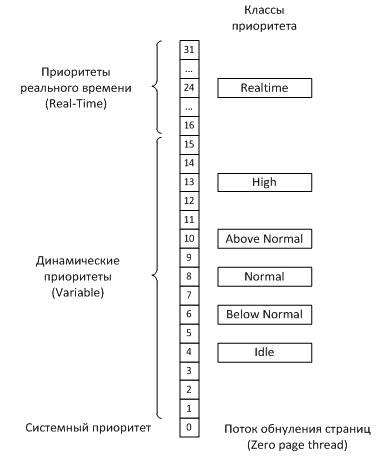
**Рис. 9.2.**Изменение величины кванта в Windows 7number

За изменение величины кванта отвечает функция KeSetQuantumProcess (файл base\ntos\ke\procobj.c, строка 1393).

Кроме длинных и коротких квантов в Windows реализовано динамическое увеличение размера кванта для потоков активного процесса (т.е. того процесса, окно которого в настоящий момент активно). За повышение кванта (и приоритета) отвечает функция PspComputeQuantumAndPriority (файл base\ntos\ps\psquery.c, строка 4415). Более подробную информацию о динамическом увеличении кванта см. [[5](https://www.intuit.ru/studies/courses/10471/1078/literature#literature.5), стр. 361].

**Приоритеты**

В операционной системе Windows имеется 32 уровня приоритета – от 0 до 31 ([рис.9.3](https://www.intuit.ru/studies/courses/10471/1078/lecture/16576?page=1#image.9.3)).



**Рис. 9.3.**Приоритеты в Windows

Приоритеты назначаются процессам и потокам. У процесса имеется единственный приоритет, который называется базовым. Значение этого приоритета хранится в поле BasePriority структуры KPROCESS (файл base\ntos\inc\ke.h, строка 1028). В WinAPI для работы с базовым приоритетом процесса используются классы приоритета (например, REALTIME, NORMAL и т. д.); соответствие классов приоритета числовым значениям показано на [рис.9.3](https://www.intuit.ru/studies/courses/10471/1078/lecture/16576?page=1#image.9.3). Например, при создании процесса можно указать класс приоритета в качестве параметра WinAPI-функции CreateProcess, иначе будет установлен приоритет по умолчанию (см. лекцию 6 "Процессы и потоки", раздел "Создание процесса"). В дальнейшем класс приоритета процесса можно изменить при помощи WinAPI-функции SetPriorityClass.

В WRK структура PROCESS\_PRIORITY\_CLASS и значения соответствующих констант (заметьте, что эти значения не совпадают с числовыми значениями приоритетов) определены в файле public\sdk\inc\ntpsapi.h (строка 399). Класс приоритета процесса хранится в поле PriorityClass структуры EPROCESS (см. лекцию 7 "Процессы и потоки", раздел "Структуры данных для процессов и потоков"). Таким образом, если, например, процессу назначен класс приоритета High, то в поле PriorityClass запишется число 3 (значение константы PROCESS\_PRIORITY\_CLASS\_HIGH), в поле BasePriority – значение 13 (соответствующее числовое значение приоритета).

Поток имеет два значения приоритета – базовый и текущий. При создании потока базовый приоритет потока принимается равным базовому приоритету процесса-владельца. Можно изменить базовый приоритет потока при помощи WinAPI-функции SetThreadPriority. Параметрами этой функции являются дескриптор потока и относительный приоритет, который определяет смещение базового приоритета ([таблица 7.1](https://www.intuit.ru/studies/courses/10471/1078/lecture/16576?page=1#table.9.1)).

|  |  |  |
| --- | --- | --- |
| Таблица 9.1. Влияние относительных приоритетов | | |
| **Относительный приоритет** | **Смещение для динамических приоритетов** | **Смещение для приоритетов реального времени** |
| Time Critical | Базовый приоритет = 15 | Базовый приоритет = 31 |
| Highest | +2 | +2 |
| Above Normal | +1 | +1 |
| Normal | 0 | 0 |
| Below Normal | –1 | –1 |
| Lowest | –2 | –2 |
| Idle | Базовый приоритет = 1 | Базовый приоритет = 16 |

Пример. Имеется процесс с базовым приоритетом Below Normal (6). Поток, принадлежащий этому процессу, имеет такой же базовый приоритет. Вызов функции SetThreadPriority с параметром Highest сделает базовый приоритет потока равным 8, а с параметром Time Critical – равным 15.

Текущий приоритет потока при создании потока равен базовому, но в дальнейшем может динамически повышаться и понижаться операционной системой (эта процедура будет рассмотрена далее). Заметим, что для потоков с базовым приоритетом Real Time текущий приоритет не изменяется и всегда равен базовому.

Базовый приоритет потока хранится в поле BasePriority, а текущий – в поле Priority структуры KTHREAD (файл base\ntos\inc\ke.h, строки 1123 и 1237).

**Алгоритм планирования в Windows**

В Windows отсутствует единый модуль, отвечающий за планирование потоков. Алгоритм планирования реализуется несколькими процедурами ядра, совокупность которых называется диспетчером ядра (kernel’s dispatcher).

Для хранения данных, необходимых для планирования, предназначена база данных диспетчера ядра, которая является частью структуры KPRCB (Kernel Processor Control Block), описанной в файле base\ntos\inc\i386.h (строка 1073). Эта структура создается для каждого процессора, присутствующего в системе. Структура KPRCB содержит следующие поля, требуемые для планирования:

* CurrentThread – указатель на текущий выполняющийся поток;
* NextThread – указатель на следующий поток для выполнения;
* IdleThread – указатель на поток простоя;
* DispatcherReadyListHead – массив списков, содержащих указатели на потоки, готовые к выполнению. Количество элементов массива совпадает с количеством уровней приоритета в системе (32), т. е. для каждого приоритета поддерживается своя очередь потоков в состоянии готовности;
* ReadySummary – 32 битное число, каждый из разрядов которого отвечает за один уровень приоритета. Единица в N-ом разряде означает, что очередь готовых к выполнению потоков с приоритетом N не пустая. Это поле используется для ускорения поиска при выборе потока для выполнения.

Выбор потока с максимальным приоритетом из массива DispatcherReadyListHead с использованием поля ReadySummary осуществляется функцией KiSelectReadyThread (файл base\ntos\ke\ki.h, строка 3550).

Рассмотрим основные ситуации, возникающие при планировании потоков.

**1. Выбор потока на выполнение.**

Просматривается очередь готовых к выполнению потоков (сначала поле ReadySummary, затем, когда определена непустая очередь с максимальным приоритетом, поле DispatcherReadyListHead) и выбирается первый поток в очереди с наибольшим приоритетом, которому для выполнения предоставляется квант времени ([рис.9.4](https://www.intuit.ru/studies/courses/10471/1078/lecture/16576?page=2#image.9.4)).

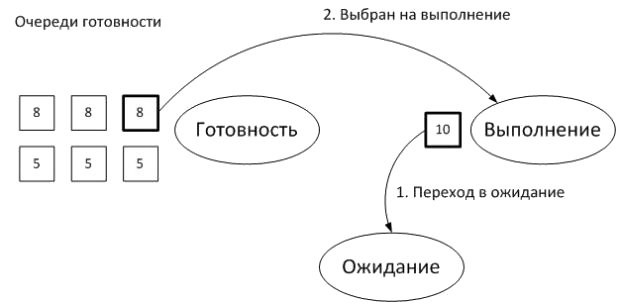


**Рис. 9.4.**Выбор потока для выполнения (квадратами обозначены потоки, числами – их приоритеты)

**2. Переход выполняющегося потока в состояние ожидания.**

Выполняющийся поток вызывает одну из функций ожидания (см. MSDN – Wait Functions [[10](https://www.intuit.ru/studies/courses/10471/1078/literature#literature.10)]) и освобождает процессор. Его квант времени не истек и сохраняется за потоком, но при выходе из состояния ожидания уменьшается на единицу (см. параграф "Кванты" этой лекции).

Диспетчер ядра выбирает на выполнение первый поток из очереди с наибольшим приоритетом ([рис.9.5](https://www.intuit.ru/studies/courses/10471/1078/lecture/16576?page=2#image.9.5)).



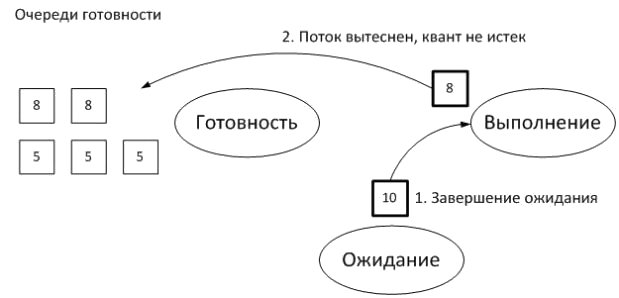
**Рис. 9.5.**Переход потока в состояние ожидания

**3. Вытеснение потоком с большим приоритетом.**

Во время выполнения поток может быть вытеснен при появлении потока с большим приоритетом. Такая ситуация может возникнуть по следующим причинам:

* поток с большим приоритетом завершил ожидание ([рис.9.6](https://www.intuit.ru/studies/courses/10471/1078/lecture/16576?page=2#image.9.6));
* приоритет потока в очереди готовности динамически увеличился (см. далее в этой лекции);
* в системе создан поток с большим приоритетом.

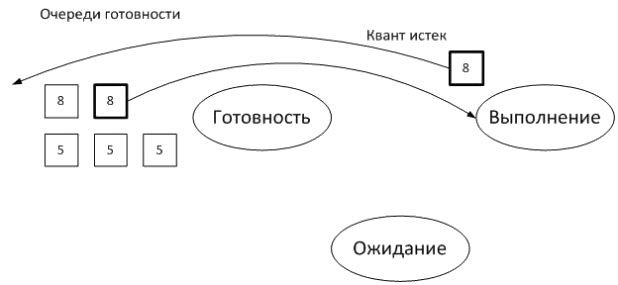
В любом случае выполняющийся поток вытесняется, помещается в начало очереди готовности с соответствующим приоритетом; при этом неистраченная часть кванта остается за потоком.



**Рис. 9.6.**Вытеснение потока

**4. Завершение кванта времени**

Когда квант времени, предоставленный потоку, истекает, операционная система проверяет, есть ли в очереди готовности поток с таким же приоритетом или выше. Если есть, то поток помещается в конец соответствующей очереди готовности и новый поток выбирается на выполнение ([рис.9.7](https://www.intuit.ru/studies/courses/10471/1078/lecture/16576?page=2#image.9.7)). Если такие потоки отсутствуют, выполняющемуся потоку может быть предоставлен новый квант времени.



**Рис. 9.7.**Завершение кванта

**Динамическое повышение приоритета**

Если бы операционная система осуществляла планирование потоков только на основе выше рассмотренных ситуаций, большинство потоков с низким приоритетом вообще никогда не выполнялись бы – диспетчер ядра все время выбирал бы потоки с наивысшим приоритетом.

Чтобы дать всем потокам шанс на выполнение операционная система применяет механизм динамического повышения приоритета (Priority Boosts), который работает в следующих случаях:

* возникает событие диспетчера ядра;
* завершается операции ввода/вывода;
* происходит событие пользовательского интерфейса;
* поток слишком долго ожидает ресурс;
* поток слишком долго ожидает своей очереди на выполнение.

Замечание. Никогда не повышаются приоритеты потоков реального времени (16–31).

**Резюме**

В этой лекции рассмотрены основные алгоритмы планирования потоков, в том числе, вытесняющие и невытесняющие, с квантованием и с приоритетами. Описаны состояния, в которых могут находиться потоки. Приведены особенности реализации квантования и приоритетов в Windows. Рассмотрен алгоритм планирования потоков, используемый в Windows.