5 Управление выполнением программ. Процессы и потоки

5.1 Основные определения, концепции и сущности

Вычислительная система предоставляет пользователю свои ресурсы: время своего вычислительного устройства, т.е. процессора/процессоров, и прочие (файлы и т.д.).
Время процессора – тоже один из ресурсов, но он рассматривается отдельно от других в силу его «врожденных» особенностей.

Процессорное время утилизируется в виде ***вычислительных процессов*** и ***потоков***.

Процесс – «совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие» (ISO 9000:2000)[1]

Вычислительный процесс, ***Process*** (в упрощенном представлении) – выполняющаяся программа и все ее элементы (выделенные для нее ресурсы).

Для процесса выделяют собственно процесс как ход выполнения программы и ***образ процесса*** – код, подлежащий выполнению, причем разделяются образ ***на диске*** (исполняемый файл) и образ ***в памяти*** (уже загруженный в оперативную память и настроенный для исполнения). Код по себе (без средств его исполнения) процессом не является, поэтому вычислительный процесс часто определяют как ***динамический объект***, соответствующий выполняющейся программе.

[1] Серия стандартов ISO 9000 относится к управлению, менеджменту качества, процессам достижения заданных результатов. Таким образом, это определение и очень универсальное, и очень абстрактное.

Образ в памяти – сегментация. Основные сегменты: ***код*** (***Code***, ***Text***), данные (***Data***), стек (***Stack***). Могут дополнительно присутствовать также сегменты ***неинициализированных*** данных (***BSS***), ***кучи*** (***Heap***) и др.

(краткая характеристика сегментов)

Многозадачность (в ОС) – способность ОС выполнять параллельно (псевдопараллельно) более одной программы («задачи»), более или менее равноправных между собой, с возможностью полноценно и эффективно управлять ими.
В частности, даже при наличии ***резидентных программ*** и ***обработчиков прерываний*** ОС уровня MS DOS остается однозадачной.

Подходы к обеспечению многозадачности:

Пакетная многозадачность – выполнение потока задач в пределах отведенных лимитов времени, по истечении которого задача удаляется как необслуженная. Задачи в очереди могут переупорядочиваться для оптимальной загрузки ресурсов системы, для достижения максимальной общей пропускной способности.

Разделение времени – организация выполнения нескольких задач на одном процессоре в виде чередующихся достаточно коротких шагов. Такая многозадачность может быть двух видов: вытесняющая или невытесняющая.

Вытесняющая многозадачность («истинная», ***preemptive***) с ***квантованием*** времени – предоставление задачам процессорного времени в виде отрезков фиксированной длительно-

сти – ***квантов*** (обычно от единиц до десятков миллисекунд). Переключение задач – по инициативе диспетчера.

Кооперативная многозадачность (***non-preemptive***) – задача получает управление и сохраняет его до тех пор, пока сама не заявит о возможности себя прервать. Этим потенциально повышает эффективность, но также и зависимость от качества реализации отдельных задач.

В обоих случаях выбор следующей задачи из числа готовых к исполнению – с использованием или без использования ***приоритетов***. Более подробно дисциплины планирования будут обсуждаться позже.

Многозадачность ***параллельных*** систем (включая вычислительные кластеры и вычислительные сети) – достижение истинного параллелизма выполнения за счет использования параллельных архитектур с физически отдельными исполнительными блоками.

Многозадачность ***реального времени*** (***real-time***) – обеспечение в первую очередь гарантированной реакции системы на события с задержкой не более заданной. Программы обычно строятся в виде совокупности ***обработчиков*** событий.

Для современных «универсальных» ОС типична многозадачность на основе квантования времени, вытесняющая, с приоритетами.

Концепция процессов оказалась недостаточной, особенно для параллельных вычислительных архитектур. Была введена новая сущность – вычислительный поток.

Вычислительный поток, ***Thread*** (ранее встречались варианты «нить» или «шаг») – минимальная «единица обработки», исполнение которой обеспечивается операционной системой (очень абстрактное формальное определение); последовательность инструкций, исполняемых независимо и асинхронной по отношению к другим таким же последовательностям.

Реализация процессов и потоков сильно зависит от конкретной операционной системы, но так или иначе в ***многопоточной*** системе именно поток становится основным объектом планирования времени выполнения и участником взаимодействия параллельных программ (получателем процессорного времени). Процесс при этом выступает как «контейнер» потоков и обладатель прочих «материальных» вычислительных ресурсов (объектов), которые доступны всем его потокам.

Таким образом, можно выделить еще две разновидности реализации многозадачности:
– «обычная», основанная на поддержке только процессов;
– многопоточная – поддерживаются также и потоки.

Процесс (и поток) характеризуются ***контекстом***, обладают ***состоянием*** и рядом других ***атрибутов***.

Контекст процесса (в упрощенном представлении) – вся информация, необходимая для однозначного и исчерпывающего описания процесса. К контексту относятся состояние регистров процессора, принадлежащие процессу пространства адресов (определяемое селекторными регистрами) и дескрипторов объектов, относящиеся к процессу управляющие структуры системы (включая дескрипторные таблицы), окружение и т.д. Переключение между процессами технически представляет собой переключение текущего контекста. Ввиду значительного объема входящей в контекст информации эта операция является длительной и ресурсоемкой («тяжелой»).

Контекст потока обычно «легче», чем процесса – как минимум, не затрагивается адресное пространство. Соответственно, «легче» и операция переключения контекстов потоков, но лишь до тех пор, пока это потоки одного и того же процесса, иначе потребуется смена также и контекста процесса.

Среди ***атрибутов*** наиболее активно используются:

Идентификатор процесса, ***Process ID*** (***PID***) – сопоставленное процессу числовое значение, уникальное в системе и не изменяющееся в течение «жизни» процесса. Разные экземпляры (копии) одного и того же процесса имеют разные PID.

Идентификатор процесса-«родителя», ***Parent PID*** – PID процесса, считающегося «родителем» данного. Этот атрибут важен, если на «родителя» возложены какие-либо обязанности по отношению к «потомку» (например, недопущение накопления процессов-«зомби»).

Пользователь процесса, ***User ID*** (***UID***) – идентификатор пользователя, от имени которого выполняется процесс. Часто дополняется также ***группой процесса***, ***Group ID*** (***GID***). Используется в первую очередь для контроля прав доступа. В зависимости от ОС, представление информации о пользователе и группе может различаться (например, в Unix-системах выделяются ***реальные*** и ***эффективные*** UID и GID).

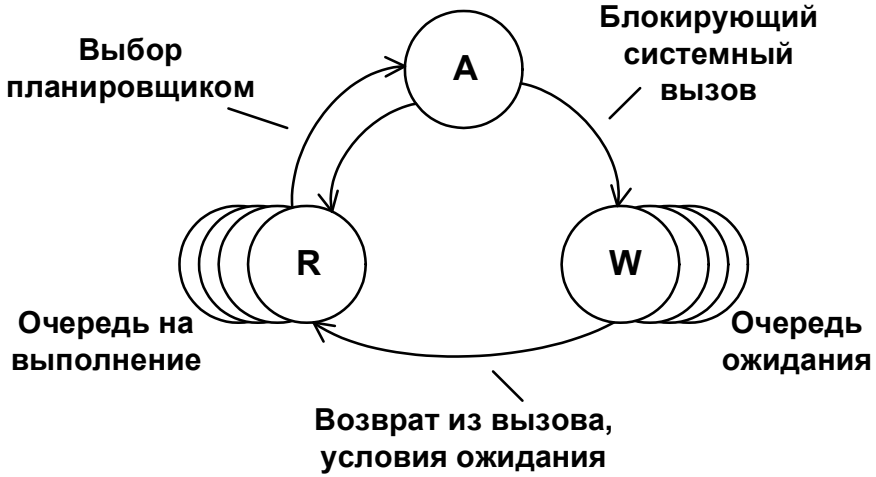
Приоритет, ***Priority*** – параметр, влияющий на логику планировщика задач по отношению к данному процессу. Представление также может сильно различаться в зависимости от ОС (например, в Unix-системах – «***nice number***»). Подробно рассматривается ниже.

Кроме того, ведется ***статистика*** использованного процессом времени процессора.

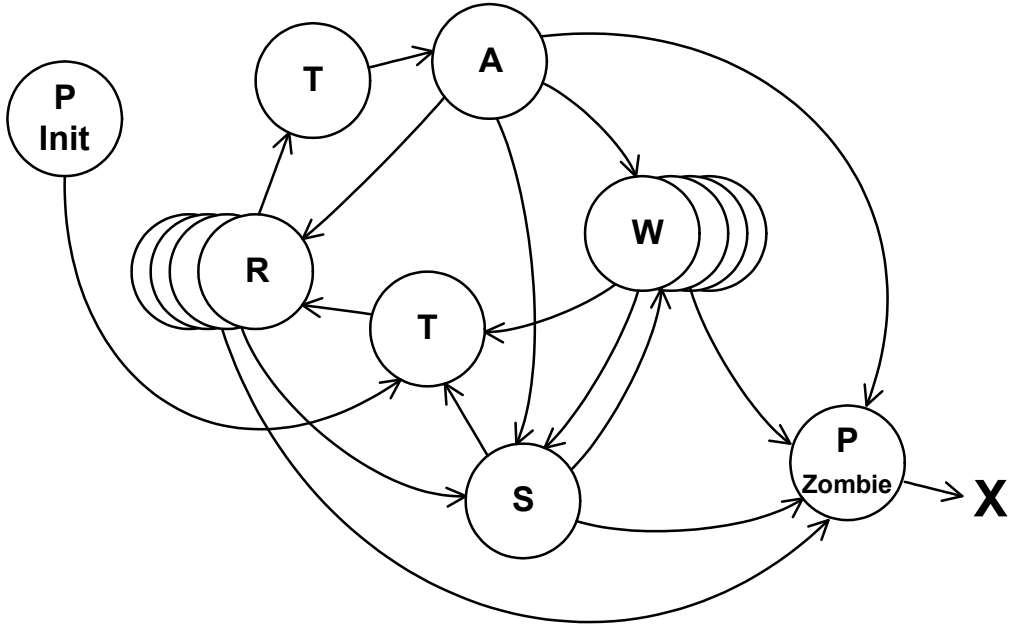
5.2 Жизненный цикл и состояния процессов (потоков)

Поскольку дальнейшее обсуждение относится в основном к многопоточным ОС, будем следовать свойственному им «разделению обязанностей» между сущностями: в планировании выполнения участвуют именно потоки. Соответственно, о жизненном цикле, состояниях и т.п. в контексте планирования выполнения (управления выполнением) будем говорить применительно к потокам.

В течение своего существования (жизненного цикла) поток проходит ряд состояний, часть из которых «проходные», другие повторяются многократно:



Основные состояния потока: выполнение (A), готовность (R), ожидание (W)



Более сложный граф состояний потока
(T – транзитные (переходные) состояния

Состояния:

P – ***Passive*** – «пассивное» состояние (например, начальное – до старта, или финальное – «зомби»). Выполнение потока невозможно, необходимо закончить его построение, инициализацию, разрушение и т.д.

R – ***Ready*** – «готовность». Поток обладает всеми необходимыми ресурсами, кроме времени процессора, и находится в ***очереди*** на выполнение.

A – ***Active***, в других источниках также ***Running*** – «активное», выполнение потока. Поток обладает всеми необходимыми ресурсами, включая предоставленный ему процессор. В системе может быть может быть несколько активных потоков одновременно (по количеству доступных процессоров или ядер процессоров), но в большинстве случаев достаточно упрощенной модели с единственным активным потоком.

W – ***Wait***, иногда также ***Sleep*** – «ожидание» («сон»). Поток запросил некоторый недостающий ресурс или длительную (блокирующую) операцию (обычно связанную с вводом-выводом), и ожидает завершения запроса в очереди, не конкурируя за право выполнения.

Основной «треугольник» состояний составляют ***R*** – ***A*** – ***W***. Обычно большая часть времени существования потока проходит в этих состояниях, сменяющих друг друга.

Переходы между состояниями:

– Переходы между Active и Ready – переключение потоков планировщиком.

– Переход из Active в Wait – обращение потока к блокирующему системному вызову.

– Возврат из Wait (в Ready) – возврат из вызова: завершение операции ввода-вывода, наступление события, истечение интервала ожидания и т.д.

В зависимости от реализации в конкретных ОС могут выделяться другие состояния – дополнительные, промежуточные, переходные. Они в целом не меняют схему смены состояний, но иногда бывают важны с точки зрения корректного управления. Например:

S – ***Suspended*** – «приостановленное» состояние. Поток способен выполняться, но остановлен искусственно. Обратный переход также искусственный, по инициативе других потоков.

Отдельное внимание уделяется этапу завершения потока и финальному состоянию, называемому «зомби»:

Z – ***Zombie*** – состояние «зомби», конечное в жизненном цикле. Поток прекратил выполнение, занимаемые им ресурсы освобождены, но соответствующий системный объект сохраняется как структура данных: существует валидный идентификатор, доступен код завершения, статистика времени выполнения и т.п. При выполнении определенных условий (например, закрытие последнего дескриптора на этот объект) он удаляется из системы окончательно.

Расход ресурсов системы на каждый поток (и процесс) в состоянии «зомби» относительно невелики, но при большом их

количестве суммарные становятся значительными. Качественно реализованное приложение должно заботиться о своевременном удалении завершившихся потоков. Это наиболее актуально для «долгоживущих» процессов, например серверов.

При наличии многопоточности состояниями процесса можно считать:
– начальное – до старта главного потока
– активное – при наличии хотя бы одного выполняющегося потока
– «зомби» – все потоки завершены, но объект «процесс» еще сохраняется
– заблокированное – все потоки остановлены или находятся в ожидании (в частности, ***тупик***)

5.3 Процессы и потоки в Windows

Основные объекты в реализации многозадачности Windows: вычислительный ***процесс*** (***process***), вычислительный ***поток*** (***thread***). Дополнительно: ***нить*** (***fiber***), ***задание*** (***job***).

Модель многозадачности и дисциплина планирования – много-поточная, с квантованием времени, вытесняющая с абсолют-ными приоритетами.

5.3.1 Процесс

Объект Process – объект ядра.

Идентификация:
– уникальный в системе ***идентификатор*** процесса – Process ID, ***PID***
– ***дескрипторы*** в пространствах других процессов – ***Handle***.

В отличие от большинства других объектов, Process сам тоже владеет собственным дескриптором – ***локальным*** Handle. Это позволяет ему существовать независимо от сохранения его дескрипторов другими процессами.

Порождение – из исполняемого файла. Основная системная функция:

`CreateProcess();`

Функции-надстройки:

`exec**()` – *семейство функций стандартной библиотеки* C/C++

`WinExec()` – *более простая функция, унаследованная из Win16 (не рекомендуется)*

`ShellExecute()` – *«выполнение» и иные операции с файлами, в т.ч. файлами данных (через ассоциации их с приложениями)*

Самостоятельная функция, обеспечивающая загрузку двоичного образа из файла для выполнения различных действий, в т.ч. для исполнения программ:

`LoadModule()`

Завершение текущего процесса – основная системная функция. Оставляет код возврата:

`ExitProcess( unsigned uExitCode);`

К тому же результату приводят:

`exit()` – стандартная библиотека C/C++
`return` – оператор выхода из подпрограммы в головном модуле.

Характерно, что в обоих случаях завершение реально обрабатывает именно **`ExitProcess()`**: функция **`exit()`** передает ей управление, то же происходит автоматически при выходе из головного модуля программы.

Завершение другого процесса «извне» по его дескриптору:

```
TerminateProcess( hProcess, uExitCode);
```

Объект Process может быть открыт для доступа по его ID:

```
OpenProcess(
  dwDesiredAccess, bInheritHandle, dwProcessID);
```

Результат выполнения – дескриптор «чужого» процесса для доступа заданного вида к нему (необходимы соответствующие права).

(Часть перечисленных функций имеют также и **-Ex**-версии.)

5.3.2 Поток

Объект ***Thread*** – объект ядра, идентифицируется ID (практически используется редко) и Handle. Подобно процессу, имеет «локальный» handle потока.

Процесс всегда имеет хотя бы один «первичный» («главный») поток, в котором осуществляется выполнение процесса. Дополнительно может создаваться явным образом произвольное количество «вторичных» потоков. Завершение главного потока приводит к завершению всего процесса и всех остальных его потоков.

Все потоки процесса делят его адресное пространство, кроме стека, и пространство дескрипторов (объектов), но имеют собственные сегменты ***стека***. Данные потоков могут храниться также в ***локальной памяти потоков*** (***TLS***, см. ниже).

Порождение – из ***функции потока***, процедурный тип **LPTHREAD_START_ROUTINE**:

```
DWORD WINAPI MyThreadRoutine( void* pMyThreadParam)
{
  ... // действия, выполняемые в потоке
  return dwMyThreadResult;
}
```

Создание нового потока:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES pThreadSecurityAttr,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE ThreadRoutine,
  void* pThreadParam,
  DWORD dwCreationFlags,
  DWORD* pThreadId);
```

Дополнительная возможность – создание потока «удаленно», т.е. в адресном пространстве другого процесса:

`CreateRemoteThread();`
`CreateRemoteThreadEx();`

Завершение потока:

`ExitThread( dwExitCode);` – *нормальное завершение потока «изнутри»*
`return dwExitCode;` – *выход из функции потока (неявное обращение к ExitThread)*
`TerminateThread( hThread, dwExitCode);` – *принудительное завершение потока.*

Принудительное завершение процессов и потоков нежелательно, т.к. происходит в произвольные моменты, при неизвестном их состоянии. Например, могут остаться незавершенными последовательности операций записи в файл, из-за чего возмож-

на потеря целостности содержимого. В случае потоков возможна «утечка» ресурсов – объекты, которые были созданы потоком и не удалены при завершении, остаются в памяти до завершения всего процесса. А при использовании средств синхронизации и взаимного исключения могут остаться заблокированными ***критические секции***, что нарушает дальнейшую работу взаимосвязанных процессов (потоков) вплоть до состояния ***тупика***. Также может быть нарушена работа подключенных динамических библиотек DLL (включая системные!), т.к. они не получают извещения о принудительном завершении потоков.

В любом случае будет определен результат выполнения потока в виде возвращаемого из функции потока значения (или аргумента **`ExitThread()`** или **`TerminateThread()`**). Это значение будет доступно посредством функции:

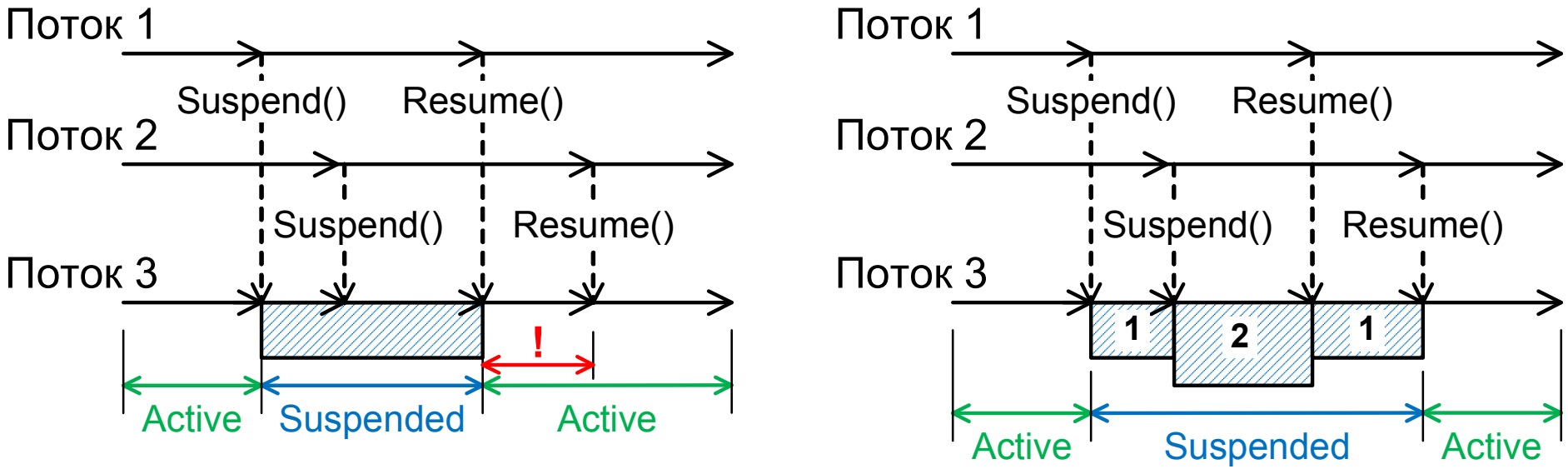
`GetExitCodeThread()`

Приостановка («заморозка», «усыпление») и возобновление выполнения потока:

```
SuspendThread();
ResumeThread();
```

«Приостановленное» (***suspended***) состояние управляется внутренним счетчиком: при нулевом его значении поток выполняется, при положительном – остановлен. Функция **`SuspendThread()`** безусловно наращивает счетчик, **`ResumeThread()`** – уменьшает значение, но не ниже нуля (т.е. поведение счетчика подобно семафору). Таким образом, приостановка может быть многократной (вложенной, ***рекурсивной***), причем остановка потока срабатывает всегда, но для возобновления его работы необходимо «раскрутить» всю глубину вложенности. Это защищает программы от неожиданного «пробуждения» остановленного потока, но создает риск оставить его неактивным.

Пример: несколько потоков (здесь 1 и 2), которые могут конфликтовать с третьим и поэтому временно приостанавливают его и затем снова возобновляют его выполнение.



Простая и рекурсивная (вложенная) приостановки потока

Без рекурсивной логики блокировки возможна ситуация, когда активное состояние потока будет неожиданным и нежелательным для одного или нескольких ранее блокировавших его.[1]

Как и принудительное завершение, безусловная директивная остановка потоков может приводить к «срыву синхронизации», нарушению согласованной работы потоков, блокировкам. Использование этого механизма требует продуманного проектирования и хорошего представления целей и результатов.

[1] Это – типичная критическая секция, и для разрешения коллизий желательно использовать соответствующие механизмы и объекты синхронизации (ISO): мьютексы и им подобные. Решение с использованием приостановки потоков потенциально менее надежно, но иногда приходится применять и его.

5.3.3 Использование памяти многопоточным приложением

Поскольку все потоки процесса делят его адресное пространство, кроме стека, и пространство дескрипторов, то переменные статического (**`static`** в C/C++) класса хранения и созданные любом из потоков объекты, а также блоки памяти, динамически выделенные в «куче» (***Heap***), потенциально доступны всем потокам того же процесса. Однако каждый поток имеет собственный стек, поэтому создаваемые в нем переменные ***автоматического*** (**`auto`** в C/C++) класса хранения – собственные для каждого из потоков, даже если исполняются одни и те же функции.

Следствие общего адресного пространства и общих объектов:
– упрощение и ускорение взаимодействие между потоками в рамках одного процесса;
– риск конфликтов при совместном использовании, необходимость синхронизации;
– риск «утечек» ресурсов, когда поток запрашивает их для монопольного использования и не освобождает перед завершением (созданные объекты не известны другим потокам и, следовательно, не могут быть ими освобождены).

Следствие собственных стеков потоков:
– независимость (между потоками) локальных переменных функций;
– невозможность использования таких переменных во взаимодействии потоков.

Рис. – некорректный доступ к переменной в стеке

Локальная память потоков, ***Thread Local Storage*** (***TLS***) – позволяет иметь у разных потоков одинаковый набор переменных с разными значениями (собственный экземпляр переменной для каждого потока). TLS-переменные идентифицируется индексом, причем каждому индексу соответствует «слот» в TLS-таблице, содержащий независимые экземпляры (значения) этой переменной для каждого из потоков процесса. Единственный поддерживаемый тип переменных – void-указатель; при необходимости хранить иные (но «помещающиеся» в этот формат!) значения потребуется явное ***приведение типа***.

Рис. – организация TLS

Функции для работы с TLS:

DWORD TlsAlloc(void); *//созданние TLS-переменной, возвращает ее индекс*
BOOL TlsFree(DWORD *dwIndex*); *//удаление TLS-переменной по индексу*
void* TlsGetValue(DWORD *dwIndex*); *//чтение TLS-переменной* **по индексу**
BOOL TlsSetValue(DWORD *dwIndex*, void* *pValue*); *//запись TLS-переменной*

TLS-переменные требуют дополнительных затрат по сравнению с обычными переменными auto или static. При необходимости иметь большое количество таких переменных рекомендуется группировать их в структуру в динамически выделенной памяти и хранить в TLS-слоте указатели на экземпляры этих структур каждого из потоков.

TLS позволяет упорядочить использование памяти потоками: переменная (по крайней мере факт ее существования) известна вне использующего ее потока. Это полезно, когда архитектура приложения включает поток-«монитор» и несколько однотипных потоков-«исполнителей»: «монитор» создает переменные (и, возможно, структуры в динамической памяти) и затем потоки-«исполнители», передавая тем соответствующие индексы и ведя учет переданных в пользование ресурсов. (Индексы TLS-переменных могут быть глобальными переменными; для потоки-«исполнители» лишь читают их значения, поэтому конфликтов не возникает.) Таким образом, распределение ресурсов сосредотачивается в «мониторе», что позволяет более эффективно предотвращать или устранять возможные «утечки».

5.4 Нити и задания

Дополнительные возможности управления выполнением программ, вторичные по отношению к процессам и потокам сущности, в большинстве случаев не используются, но иногда могут оказаться полезны.

5.4.1 Нити

Нить, ***Fiber*** (также встречается вариант перевода «волокно») – реализуют кооперативный подход к многозадачности: переключение между «единицами планирования» по их собственной инициативе и, соответственно, в наиболее удобные для них моменты.

Нить не является «полноценным» объектом ядра. Идентификация нитей – указатель **`void*`** (**`LPVOID`**), который не является дескриптором.

Создание первой (первичной) нити потока (преобразование потока в нить), выполнение кода продолжается уже в качестве нити:

```
ConvertThreadToFiber();
```

Создание новой нити в текущем потоке – из подпрограммы (***функции нити***), при этом выполнение на новую нить не переключается:

```
void* CreateFiber( SIZE_T dwStackSize,
  LPFIBER_START_ROUTINE FiberRoutine, void* pParam
);
void* CreateFiberEx( SIZE_T dwStackCommitSize,
SIZE_T dwStackReserveSize,
  DWORD dwFlags, LPFIBER_START_ROUTINE
lpStartAddress, void* lpParam
);
```

Подобно потоку, нить получает собственный стек. Первичная нить потока использует его стек. Также нити получают ***локальную память*** – ***Fiber Local Storage*** (***FLS***), подобную TLS.

Процедурный тип **LPFIBER_START_ROUTINE**, который определен для функции нити, соответствует формату:

```
void CALLBACK MyFiberRoutine( void* pMyFiberParam)
{
   ... // действия, выполняемые нитью, в т.ч.
переключения
  return;
}
```

Флаг **`FIBER_FLAG_FLOAT_SWITCH`** (функция **`CreateFiberEx()`**) обеспечивает корректное использование нитью вещественной арифметики. В противном случае попытки вычислений с вещественным типом могут приводить к повреждению данных (вероятно, из-за особенностей совместного доступа к математическому сопроцессору).

Переключение с текущей нити на другую, при этом выполнение текущей приостанавливается:

```
SwitchToFiber( )
```

Переключение между нитями осуществляется только явно, по инициативе текущей нити, и только в пределах одного потока. Отдав управление, нить не «знает», когда оно будет ей возвращено. Поток – «контейнер» нитей – продолжает диспетчироваться наравне с другими потоками, по общим правилам; для планировщика потоком наличие нитей не существенно.

Получение указателя на текущую нить (макрос):

```
void* GetCurrentFiber();
```

Получение информации о текущей нити в структуре **FIBERDATASTRUCT**:

```
void* GetFiberData();
```

Удаление нити:

```
void DeleteFiber( void* pFiber);
```

Удаление нити сопровождается освобождением памяти и очисткой всех данных этой нити.

Удаление текущей нити активного потока из самой этой нити приводит к завершению потока.
Удаление текущей нити из другой нити (очевидно, относящейся к другому потоку) приводит, скорее всего, к аварийному завер-

шению потока удаляемой нити («is likely to terminate abnormally»).

5.4.2 Задания

Задание, ***Job*** – группировка и совместное управление несколькими процессами. В Windows эта сущность появилась начиная с Win XP и Win 2003 Server.

Задание – системный объект, идентифицируемый глобальным именем (необязательным) и дескриптором (handle) в пространстве дескрипторов процесса.

Основные функции работы с заданиями:

```
HANDLE CreateJobObject()
HANDLE OpenJobObject()
BOOL QueryInformationJobObject()
BOOL SetInformationJobObject()
BOOL AssignProcessToJobObject()
```

Кроме того, задания могут выступать в роли ***объектов ожидания*** для функций **`Wait**()`**.

5.5 Приоритеты

Приоритет, ***Priority*** – значение (обычно целочисленное), используемое системным планировщиком и характеризующее важность и срочность выполнения процесса: процесс с более высоким приоритетом время процессора для своего выполнения чаще и/или в большем объеме по сравнению с менее приоритетными. Количество уровней приоритетов обычно составляет от нескольких единиц до нескольких десятков, и лишь редко сотен (например, 0..255). Большое количество уровней бывает востребовано в системах реального времени, где от точного назначения и соблюдения приоритетов зависит корректность разрешения коллизий между событиями.

Диапазон значений приоритетов, способ их формирования, направление убывания/возрастания приоритетов, логика их применения (***дисциплина планирования***) зависят от конкретной

системы. Соответственно, выделяют ряд разновидностей приоритетов, например:

Статический приоритет – остается неизменным в течение всего жизненного цикла процесса;

Динамический приоритет – меняется в зависимости от дополнительных условий, например от длительности пребывания в ожидании, или от внешних событий;

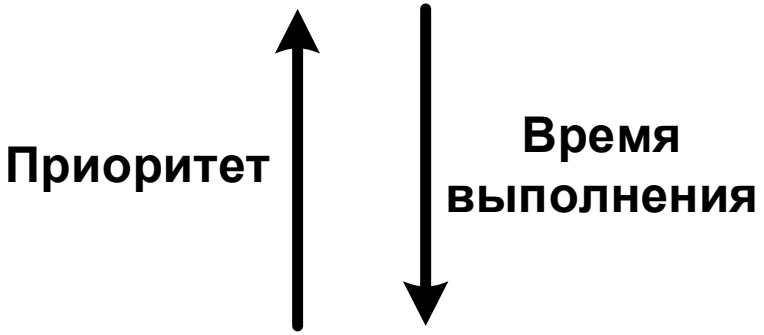
Относительный приоритет – выбор и переключение процессов выполняются после завершения текущего процесса, перехода его в состояние ожидания или окончания кванта времени;

Абсолютный приоритет – при появлении готового к выполнению более приоритетного процесса текущий менее приоритетный прерывается и вытесняется немедленно, не дожидаясь естественной смены состояния (в т.ч. окончания кванта);

В многопоточной системе, где за время процессора конкурируют потоки, корректнее говорить о ***приоритетах потоков***, однако при их формировании обычно учитываются и характеристики процесса, к которому они относятся (составные значения приоритетов).

Приоритеты учитываются только для процессов в состояниях активности (выполнения) и готовности. Процессы в пассивном или ожидающем состояниях не конкурируют за время процессора вне зависимости от их приоритетов.

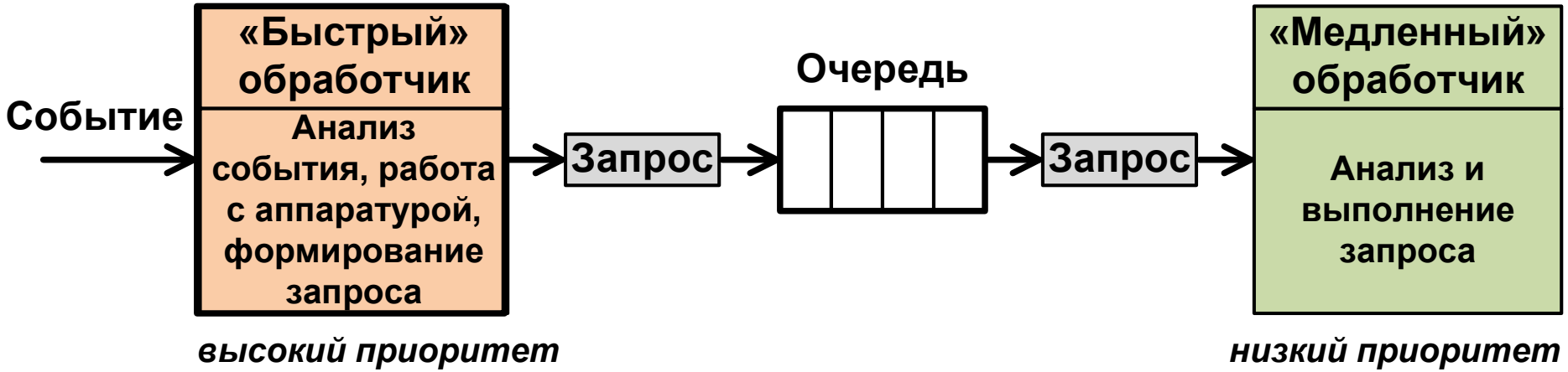
Общее правило: в правильно построенной сбалансированной системе чем выше приоритет, тем меньше общее время исполнения под этим приоритетом (в обмен на оперативность этого исполнения). Иначе говоря, необходимо минимизировать время выполнения кода с высокими приоритетами. Менее приоритетные процессы могут заполнять все оставшееся время, и им не нужно специально заботиться о том, чтобы «уступать место» более приоритетным.



Балансировка приоритета и времени выполнения

Например, обработчик события может делиться на две части: высокоприоритетную «быструю», которая взаимодействует с

аппаратными ресурсами, анализирует событие и формирует запрос, и низкоприоритетную «медленную», выполняющий этот запрос. Передача запросов происходит асинхронно – через очередь.



Сочетание высоко- и низкоприоритетной обработки

Инверсия приоритетов – нарушение установленного порядка приоритетов, блокировка выполнения более приоритетного процесса менее приоритетным. Обычно это происходит из-за конкуренции при доступе к ресурсам: если ресурс уже занят монопольно каким-либо процессом, то любой другой процесс, требующий этого же ресурса, вынужден будет ждать независимо от приоритета, что искажает логику управления процессами вплоть до аварий. Конфликт можно разрешить принудительной приостановкой (удалением) «мешающего» низкоприоритетного процесса и объявлением ресурса незанятым, но такое решение не всегда приемлемо и в системах общего назначения обычно не применяется.

5.6 Приоритеты в Windows

5.6.1 Значения приоритетов

Приоритеты в Windows – целочисленные значения в диапазоне от `0` до `31`: `0` – зарезервированное значение для специального системного потока «zero-page» (очистка освободившихся страниц памяти), и от `1` до `31` – для всех остальных. Большее значение соответствует большему приоритету. Приоритет «обычных» процессов ограничен – от `1` до `15`; приоритет `16` и выше могут иметь (но не обязательно имеют!) только привилегированные (системные) процессы.

Значение приоритета – составное, оно образуется из трех составляющих: класс приоритета, приоритет потока в классе, динамическое изменение приоритета («boost»).

Класс приоритета, ***Priority Class*** – присваивается процессу. Все потоки одного процесса относятся к одному классу.

Приоритет потока или приоритет ***в классе***, ***Thread Priority*** – присваивается конкретному потоку. Интерпретация значения зависит от принадлежности к классу (см. таблицу).

Класс приоритета и приоритет потока в классе образуют ***базовый*** приоритет.

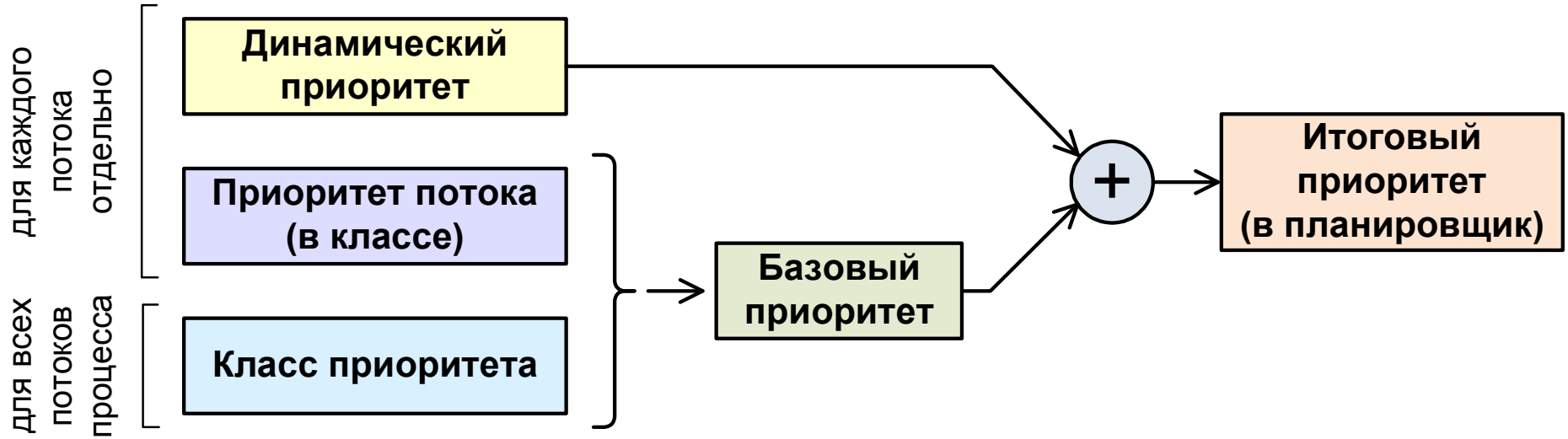


Схема формирования приоритета

Для представления классов приоритетов и приоритетов потоков в классе определены символические константы. В текущих версиях значения констант не обязательно совпадают с числовым значением уровня приоритета!

Класс приоритета		
Константа	**Значение**	**Процессы**
IDLE_PRIORITY_CLASS	**4 (0x00000040)**	«Простаивающие» (например, экранные заставки)
BELOW_NORMAL_PRIORITY_CLASS	**(0x00004000)**	
NORMAL_PRIORITY_CLASS	**7 или 9 (0x00000020)**	Обычные приложения
ABOVE_NORMAL_PRIORITY_CLASS	**(0x00008000)**	
HIGH_PRIORITY_CLASS	**13 (0x00000080)**	Системные процессы и приложения (например, Task List)
REALTIME_PRIORITY_CLASS	**24 0x00000100**	Некоторые системные процессы (обработчики аппаратных событий и т.п.)

Класс приоритета		
Константа	**Значение**	**Процессы**
PROCESS_MODE_BACKGROUND_BEGIN	**(0x00100000)**	Переход в «фоновый» режим (рекомендуется для снижения затрат ресурсов)
PROCESS_MODE_BACKGROUND_END	**(0x00200000)**	Выход из «фонового» режима

Приоритеты потоков	
Константа	**Значение**
THREAD_PRIORITY_IDLE	**=1** (все классы, кроме Real-Time и Background)
	=16 (Real-Time class)
THREAD_PRIORITY_LOWEST	**-2**
THREAD_PRIORITY_BELOW_NORMAL	**-1**
THREAD_PRIORITY_NORMAL	нет изменения относительно класса
THREAD_PRIORITY_ABOVE_NORMAL	**+1**
THREAD_PRIORITY_HIGHEST	**+2**
THREAD_PRIORITY_TIME_CRITICAL	**=15** (все классы, кроме Real-Time и Background)
	=31 (Real-Time класс)
THREAD_PRIORITY_BACKGROUND_BEGIN	**(0x00010000)**
THREAD_PRIORITY_BACKGROUND_END	**(0x00020000)**
	-7..+6 (Real-Time класс)

Результирующие значения базового приоритета (примеры):

Класс приоритета	Приоритет потока	
IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
BELOW_NORMAL_PRIORITY_CLASS		2
		3
		4

Динамический приоритет, ***Dymanic Priority*** – действующее значение приоритета с учетом временного его повышения или понижения. Планировщик повышает приоритет в случаях:
– процесс с приоритетом Normal владеет активным («сфокусированном», foreground) окном;
– в очередь потока поступают сообщения, связанные с вводом-выводом, таймерами и т.п.;
– поток выходит из состояния ожидания.

Такое временное повышение приоритета (***priority boost***) обеспечивает лучшую «отзывчивость» программ на события. Повышение приоритета не применяется к процессам и отдельным потокам с базовыми приоритетами класса Realtime.

Важно! Ни манипулирование приоритетами потоков, ни динамическое повышение приоритета не могут перевести поток из «обычного» (непривилегированного) класса в класс Realtime.

Независимо от того, как было получено итоговое значение приоритета, далее планировщик использует именно его как формальную характеристику, влияющую на выбор потоков при их переключениях.

5.6.2 Функции управления приоритетами

Получение и установка класса приоритета:

```
GetPriorityClasss( HANDLE hProcess);
SetPriorityClass( HANDLE hProcess,
 DWORD dwPriorityClass);
```

Получение и установка приоритета потока (в классе):

```
GetThreadPriority( HANDLE hThread);
SetThreadPriority( HANDLE hThread, int nPriority);
```

Дополнительно – временное повышение приоритета для процесса и потока (priority boost):

```
GetProcessPriorityBoost( hProcess, pbDisableBoost);
SetProcessPriorityBoost( hProcess, bDisableBoost);
GetThreadPriorityBoost( hThread, pbDisableBoost);
SetThreadPriorityBoost( hThread, bDisableBoost);
```

Изменение значения приоритетов не обязательно производит эффект немедленно (например, из-за отсутствия в очереди готовых к исполнению потоков подходящего «конкурента»). Иногда бывает полезно сопроводить изменение приоритета принудительным переключением активного потока:

```
SetPriorityClass( hProcess, nNewPriorityClass);
SetThreadPriority( hThread, nNewPriority);
Sleep (1);
```