Процессы и потоки

Что это

Один из основных объектов в любой ОС - **процесс**.

Процесс - это программа, запущенная на исполнение. У процесса есть свое адресное пространство, а также набор некоторых ресурсов - например, дескрипторы открытых файлов и информация о потоках управления.

Зачем вообще нужна концепция многопроцессности? Вряд ли такой вопрос может возникнуть в наше время, но все-таки в качестве ответа можно указать простой пример того, что задача может требовать, скажем, считывания большого объема данных с диска или из сети. В то время, пока это происходит, процессор простаивает, хотя мог бы выполнять полезную работу. Концепция системы с множеством процессов позволяет решить эту задачу. Второй ответ - возможность параллельного или квазипараллельного выполнения различных программ: например, возможно одновременно набирать текст в текстовом редакторе и слушать музыку на однопроцессорной машине.

В первых операционных системах процесс имел единственный поток управления. Для некоторых современных ОС, в основном, RTOS, это по-прежнему остается правдой; однако большинство популярных ОС реализуют концепцию многопоточности: процесс может иметь более одного потока управления. Таким образом, процесс превращается в своего рода "контейнер" для потоков. Поток - наименьшая независимая единица, исполнение которой может контролироваться планировщиком (в составе операционной системы либо пользовательского уровня). Все потоки одного процесса имеют общее адресное пространство, разделяют между собой ресурсы процесса, но каждый имеет свой персональный **контекст** - как правило, туда входит набор регистров и стек.

Вопрос, аналогичный первому: зачем нужна многопоточность? Ответ также аналогичен: для параллельного или квазипараллельного решения относительно независимых задач в рамках одного процесса. Например, компьютерная игра может одновременно общаться с видеокартой, обмениваться данными по сети, обсчитывать игровую логику, реагировать на действия пользователя и подгружать данные с жесткого диска. Кроме того, возможность исполнять сразу несколько потоков позволяет разбить логику программы на относительно независимые взаимодействующие модули.

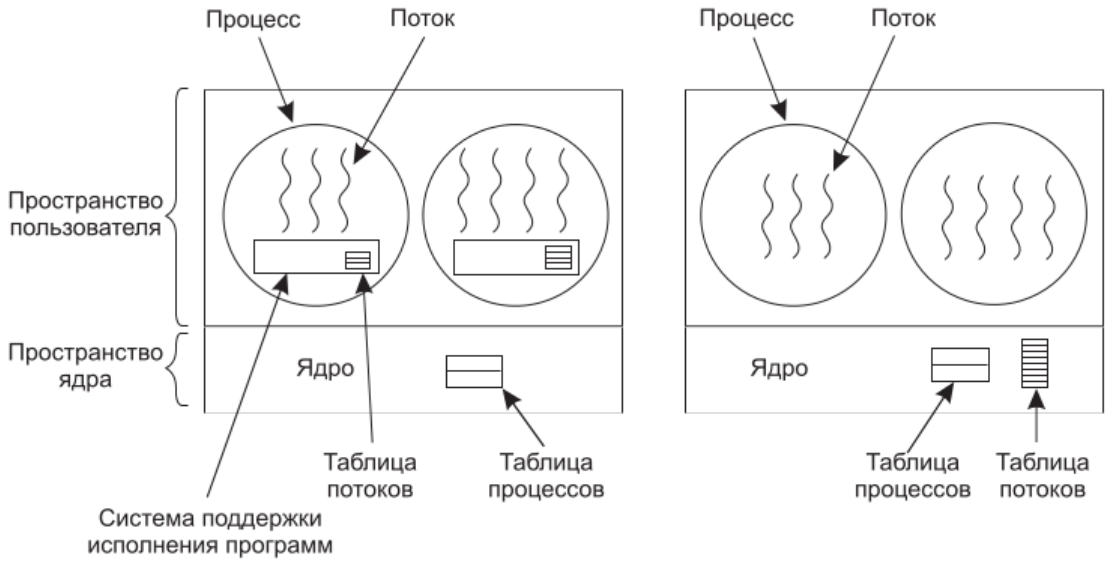
Реализация потоков

В общем и целом существует два варианта реализации потоков:

- в пользовательском пространстве;
- в ядре ОС.

Первый случай проще с точки зрения самой ОС - она по-прежнему оперирует простыми однопоточными процессами. Однако дополнительная нагрузка ложится на плечи прикладного программиста: управление потоками приходится реализовывать ему самостоятельно, да еще с учетом того, что в рамках процесса, как правило, нет механизма прерывания самого себя по таймеру. Плюсы такого подхода очевидны - поскольку потоки, по сути, представляют собой обычные процедуры, то не тратится время и ресурсы на переключение в режим ядра, запуск тяжеловесного планировщика ОС, переключение обратно в режим пользователя. Также каждый процесс может иметь свой собственный планировщик потоков, подходящий под свои нужды. Но возникает проблема с реализацией блокирующих системных вызовов: при обращении одного из потоков к такому вызову "замрет" весь процесс, так как ядро ничего не знает о потоках. При этом потоки наиболее востребованы там, где подобные ситуации блокировки возникают часто. Одним из вариантов решения этой проблемы является концепция "активации диспетчера", где ядро ОС хоть и не занимается диспетчеризацией потоков непосредственно, но при вызове приложением блокирующего системного вызова может дать ему об этом сигнал - таким образом, планировщик пользовательского уровня имеет возможность переключиться на другой поток и программа продолжит свое выполнение.

Второй подход проще с точки зрения прикладного программиста: он просто создает потоки, а их управлением занимается ОС. Однако это приводит к более существенным затратам - переключение контекста между потоками требует перехода из пространства пользователя в пространство ядра и обратно, несмотря на то, что потоки одного процесса отличаются друг от друга только контекстом исполнения, который, как правило, весьма мал.



alt text

Стоит отметить, что существует гибридный вариант - когда имеются потоки режима ядра, а в каждом из них могут быть еще несколько потоков режима пользователя. Это наиболее интересный, но и наиболее сложный вариант.

Управление и диспетчеризация

Выше неявно предполагалось, что раз потоки могут выполняться параллельно либо квазипараллельно, приостанавливаться и возобновляться, то существует некоторый механизм, управляющий этим процессом. Такой механизм называется **диспетчером**. Именно диспетчер определяет, какой поток будет выполняться, а какой - нет. Существуют различные схемы, или **дисциплины**, диспетчеризации. Основные моменты будут рассмотрены на лекции, сейчас лишь важно подчеркнуть, что большинство алгоритмов тем или иным способом определяет для каждого потока некоторую характеристику - **приоритет** - и выбирает на исполнение потоки в соответствии с ней. Как правило, одна часть этого приоритета (**статическая**) может задаваться программистом, а другая (**динамическая**) вычисляется самим диспетчером по определенному алгоритму, который учитывает использование потоком процессорного времени, ожидание в ходе ввода/вывода, группировку потоков по процессам, наличие в системе логических процессоров (Hyper-Threading) и тому подобное.

Стоит отметить, что если потоки реализованы в пространстве пользователя, то, фактически, имеется два диспетчера: диспетчер ядра ОС выполняет планирование однопоточных процессов, тогда как в каждом процессе свой собственный планировщик потоков уже распределяет время, выделенное процессу, между его потоками.

Предпочтение процессору (Processor affinity) и многопроцессорное планирование

Ситуация с процессами, потоками, приоритетами и диспетчеризацией становится заметно более сложной, если в составе вычислительной системы присутствует более одного логического процессора (а в современных реалиях это почти всегда так). Здесь планировщик каждый квант времени должен принимать множество решений: какой поток на каком процессоре выполнить; стоит ли перебрасывать выполнение конкретного потока на другой, свободный процессор, или лучше из соображений сохранения данных и кода в кэше подождать, пока освободится процессор, на котором поток выполнялся в последний раз; справедливо ли исполнять в один и тот же момент времени несколько потоков одного процесса на разных процессорах, или лучше дать возможность поработать потокам различных приложений. Это только самая вершина айсберга, поскольку приложения и даже само ядро ОС теперь могут выполняться не *квази*, а вполне себе реально параллельно. Так, один из экземпляров диспетчера, исполняющийся в виде потока ядра на одном из логических процессоров, может принять решение о необходимости переноса исполнения потока пользовательского приложения на другой логический процессор. Но в этот момент времени на этом логическом процессоре тоже исполняется какой-то код; вполне может быть, что это другой экземпляр диспетчера, который также принимает некоторое решение в данный момент. Возникают задачи синхронизации и взаимного исключения компонентов самой ОС, которая гораздо сложнее, чем аналогичная задача для пользовательских программ. Впрочем, о проблемах взаимного исключения и синхронизации - позднее.

Еще сложнее дело обстоит, если у нас не простая SMP-система с абсолютно равными между собой процессорами (или ядрами), а гетерогенная. Еще в начале века на компьютеры домашних пользователей (в серверном сегменте ее обкатали немного раньше) пришла технология Hyper-Threading, позволяющая одному физическому ядру

выглядеть как два логических. Первые версии Windows XP и Windows 2000 рассматривали подобные системы как обычные многоядерные, не учитывая того, что логические ядра имеют большое количество общих функциональных блоков и конкурируют за них, что приводило к *снижению* производительности вместо ее повышения. В начале 2010-х компания ARM предложила концепцию big.LITTLE, в рамках которой на одном кристалле объединяются два типа ядер: маломощные, но экономные, и высокопроизводительные, но прожорливые; процессоры, произведенные по этому принципу, устанавливаются в большинство современных флагманских смартфонов, включая iPhone 8 и iPhone X. Разница между "маленькими" и "большими" ядрами не только в производительности; в первых реализациях технологии между наборами ядер не соблюдался даже принцип кэш-когерентности, а в последующих отличались размерности линий кэша, что приводило к абсолютно мистическим ошибкам в программах, перемещаемых с одного ядра на другое.

Таким образом, перед операционной системой, управляющей не каким-нибудь крупным вычислительным кластером, а обычным домашним компьютером, ноутбуком или вовсе телефоном, возникает задача грамотно и эффективно распоряжаться всем этим зоопарком вычислителей, учитывая его особенности. Однако для того, чтобы бороться с описанными выше трудностями (и мириадами других) ОС должна каким-то образом "предсказывать" поведение программы, что на самом деле тоже весьма нетривиальная задача. По этой причине современные ОС предлагают прикладному программисту механизм, при помощи которого он сам в ручном режиме может рекомендовать ОС какие потоки своего процесса на каких процессорах он хотел бы исполнять - задать им *предпочтение*, или *сродство процессору* (processor affinity). Как правило, ОС будет придерживаться указанных рекомендаций.

Процессы, потоки и опять потоки и еще потоки в Windows NT

Windows NT реализует гибридную схему многопоточности: процессы (Process) являются контейнерами для потоков режима ядра (Thread), каждый из которых, в свою очередь, может содержать один или более потоков режима пользователя (Fiber, обычно переводят как "волокно") либо потоков, основанных на концепции "активации планировщика" (UMS-потоки). Кроме этого, несколько процессов могут объединяться в так называемые "задания" (Job), хотя на практике это применяется редко. Задания разделяют между собой некоторые ресурсы - например, дисковые и процессорные квоты. Одним из примеров задания может служить набор программ, выполняющихся при входе пользователя в систему (находящихся в автозагрузке) - начиная с Windows Vista им принудительно выставляется пониженный приоритет, чтобы позволить графическому интерфейсу системы побыстрее прогрузиться.

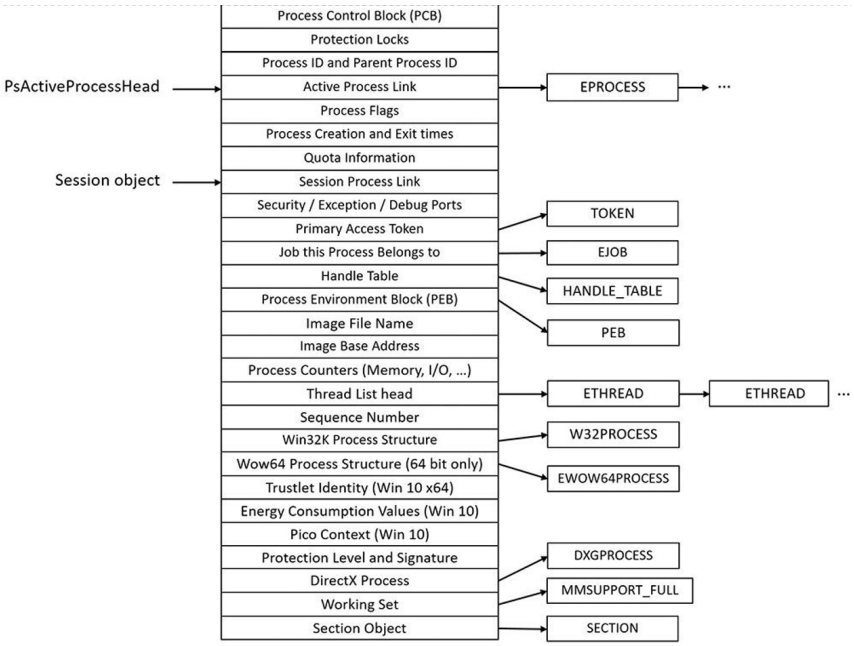
Процессы в Windows NT

Каждый процесс в Windows описывается двумя структурами: EPROCESS и KPROCESS. В них содержится вся базовая информация о процессе, необходимая ядру ОС для управления им, такая, как:

- идентификатор процесса
- идентификатор родителя
- время создания

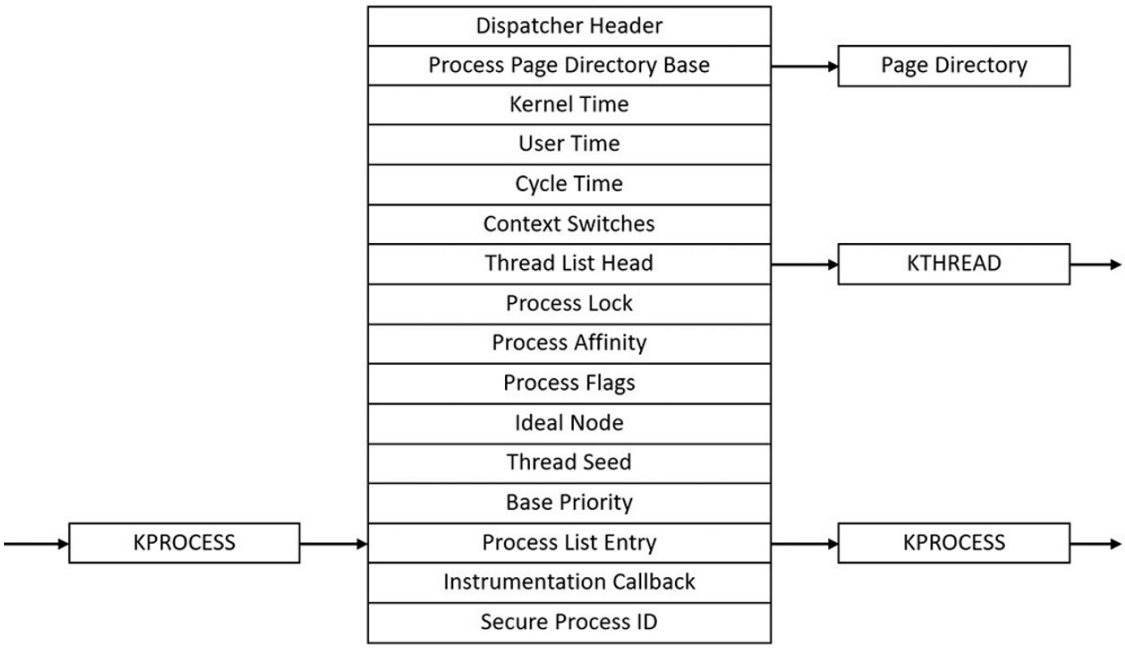
- информация о памяти процесса
- информация о приоритете процесса
- информация о потоках

и многое другое. Эти структуры расположены в пространстве ядра и сам процесс не имеет к ним доступа.



alt text

https://github.com/reactos/reactos/blob/master/sdk/include/ndk/pstypes.h#L1192 - структура EPROCESS



alt text

https://github.com/reactos/reactos/blob/master/sdk/include/ndk/ketypes.h#L1972 - структура KPROCESS

Также в памяти самого процесса есть небольшая структура, называемая Process Environment Block (PEB), в которой содержится некоторая общая информация о нем (данные об исполняемом файле, из которого был загружен процесс, о куче процесса, о контексте DirectX и тому подобное).

https://github.com/reactos/reactos/blob/master/sdk/include/ndk/peb_teb.h#L25 - структура PEB.

Кроме того, процессы, использующие подсистему Win32, также имеют еще две структуры, ассоциированные с ними - это CSR_PROCESS и W32PROCESS. CSR_PROCESS заводится процессом подсистемы Win32 csrss.exe, отвечающим за обработку запросов приложений на некоторые ресурсы в рамках Win32 API например, ввод/вывод в консоли и работу с пользовательскими сеансами; соответствующая информация в этой структуре и хранится. Структура же W32PROCESS нужна ядерной части подсистемы Win32 (win32k.sys). Она создается при первом вызове процессом любой функции, имеющей отношение к графическому интерфейсу пользователя, и хранит все ресурсы, связанные с ГИП.

https://github.com/reactos/reactos/blob/master/sdk/include/reactos/subsys/csr/csrsrv.h#L36 - структура CSR_PROCESS

https://github.com/reactos/reactos/blob/master/win32ss/user/ntuser/win32.h#L218 - структура W32PROCESS

CreateProcess

Функция CreateProcess отвечает за создание нового процесса. Она позволяет указать файл, который будет запущен на исполнение, а также флаги нового процесса, переменные окружения, текущую директорию и много чего ещё.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/proc.c#L4768

CreateProcessInternalW

На самом деле, CreateProcess является всего лишь оберткой над настоящей функцией создания процесса - CreateProcessInternalW. Напрямую вызвать ее нельзя, но ее код очень подробно демонстрирует все шаги, которые предпринимаются системой для создания процесса, поэтому рекомендуется с ним ознакомиться.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/proc.c#L2279

ExitProcess

ExitProcess завершает **текущий** процесс (то есть останавливает ВСЕ его потоки) и заносит в структуру EPROCESS код завершения, указанный параметром. Этот код в дальнейшем могут получить другие процессы системы.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/proc.c#L1532

TerminateProcess

TerminateProcess позволяет попытаться завершить некоторый **произвольный** процесс, хэндл на который передан первым параметром. Естественно, абсолютно любой процесс в системе завершить не получится - у процесса, вызывающего TerminateProcess, должны быть на это соответствующие полномочия (например, полномочия отладчика).

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/proc.c#L1577

GetCurrentProcess

Возвращает псевдо-хэндл текущего процесса, "псевдо" - потому что только в его контексте и валиден. Для того, чтобы получить хэндл, который можно было бы передать другому процессу, необходимо выполнить вызов DuplicateHandle над этим объектом. Процесс по умолчанию имеет максимальные привилегии по отношению к данному псевдо-хэндлу.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/proc.c#L1272

OpenProcess

Позволяет попытаться получить хэндл произвольного процесса по его идентификатору. Для того, чтобы попытка была успешной, как и в случае с TerminateProcess, необходимы соответствующие привилегии.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/proc.c#L1272

Get/SetPriorityClass

Функции GetPriorityClass и SetPriorityClass позволяют запросить или установить **класс приоритета** для процесса, хэндл которого указан первым параметром. Класс приоритета используется планировщиком Windows, как - будет сказано далее.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/proc.c#L1702 - GetPriorityClass

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/proc.c#L1737 - SetPriorityClass

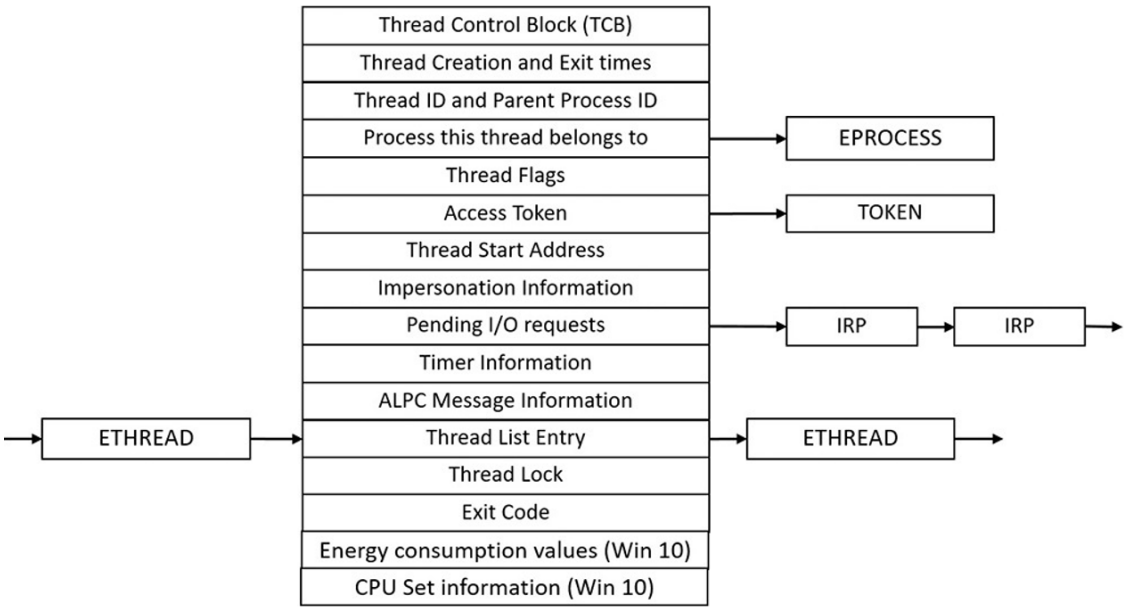
Потоки в Windows NT

Потоки в Windows описываются абсолютно аналогично процессам. Точно так же существуют структуры ETHREAD и KTHREAD, определяющие параметры потока, необходимые ядру для управления им - идентификатор потока, его приоритет, время создания и тому подобное. Однако дополнительно к информации, аналогичной информации процесса, в этих структурах хранится:

- информация о процессе, которому принадлежит этот поток;
- адрес начала потока (точка входа в него);
- информация о таймерах;
- время, проведенное потоком в режиме пользователя и режиме ядра;
- информация о блокировках;

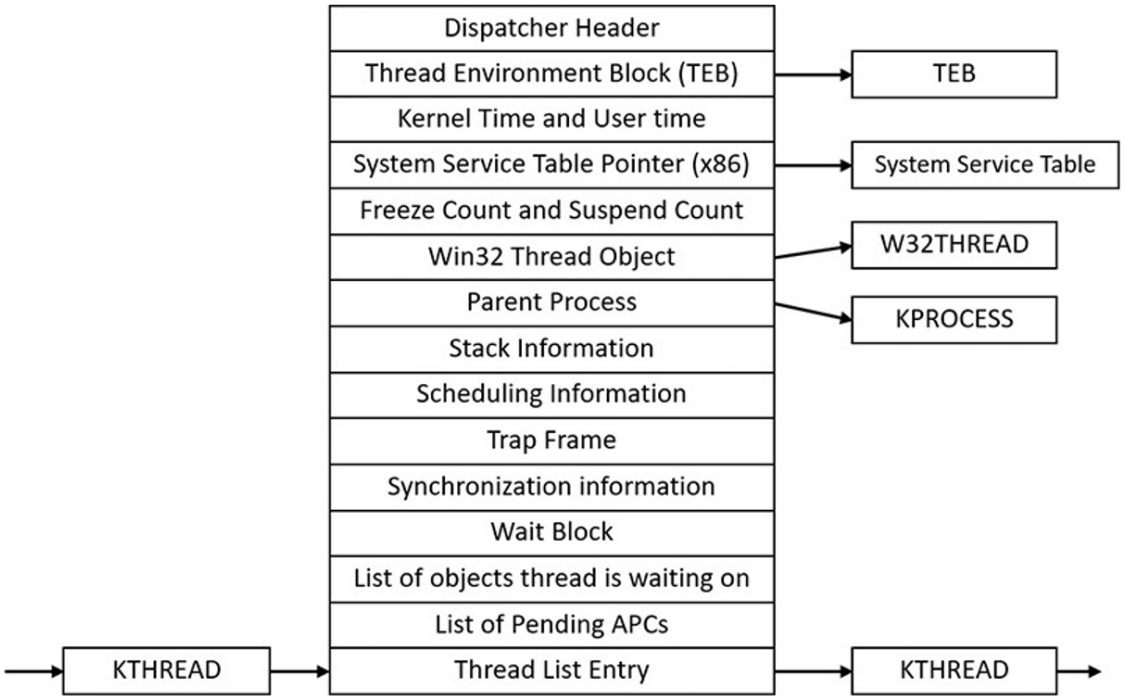
- прочее.

Эти структуры, аналогично EPROCESS и KPROCESS, также хранятся в пространстве ядра.



alt text

https://github.com/reactos/reactos/blob/master/sdk/include/ndk/pstypes.h#L1033 - структура ETHREAD



alt text

https://github.com/reactos/reactos/blob/master/sdk/include/ndk/ketypes.h#L1094 - структура KTHREAD

У потока также имеется Thread Environment Block (TEB), располагающийся в пространстве пользователя и хранящий информацию, связанную с исключениями, стеком потока, критическими секциями, контексте OpenGL и другими объектами. Кроме того, в TEB хранится ссылка на структуру Thread-Local Storage, где находятся данные, используемые библиотечными функциями и специфичные для потока (например, код ошибки последней библиотечной функции или seed для генератора случайных чисел).

https://github.com/reactos/reactos/blob/master/sdk/include/ndk/peb_teb.h#L206 - структура TEB

И вновь по аналогии с процессом на поток Win32 заводятся две дополнительные структуры - CSR_THREAD и W32THREAD. В частности, вторая из них хранит данные о кистях, поверхностях для рисования, спрайтах, антиалиасинге и прочих параметрах графики.

https://github.com/reactos/reactos/blob/master/win32ss/user/ntuser/win32.h#L64 - структура W32THREAD

https://github.com/reactos/reactos/blob/master/sdk/include/reactos/subsys/csr/csrsrv.h#L63 - структура CSR_THREAD

CreateThread

Функция создает новый поток в адресном пространстве **текущего** процесса. В качестве параметров указываются точка входа в поток, размер стека и флаги, определяющие характеристики потока (например, будет ли он приостановлен сразу после создания).

На самом деле, CreateThread реализована как `CreateRemoteThread(NtCurrentThread(), ...)`.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/thread.c#L135

CreateRemoteThread

Данная функция позволяет попытаться создать поток в адресном пространстве **другого** процесса, хэндл на который передается параметром (остальные параметры идентичны CreateThread). Естественно, на это необходимы полномочия. Код данной функции также рекомендуется к ознакомлению.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/thread.c#L157

ExitThread

Завершает **текущий** поток, освобождая все ресурсы, которые были им затребованы. Если это последний поток текущего процесса, то вызывает ExitProcess.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/thread.c#L349

TerminateThread

Пытается завершить **произвольный** поток, указанный параметром, если на то есть необходимые полномочия.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/thread.c#L571

OpenThread

Позволяет попытаться получить хэндл на поток по его идентификатору. Над полученным хэндлом в дальнейшем можно производить операции (например, приостановить или возобновить поток).

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/thread.c#L387

GetCurrentThread

Аналогично CreateProcess, но возвращает псевдо-хэндл текущего потока, а не процесса.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/proc.c#L1193

SuspendThread

Уменьшает счетчик "сонливости" указанного параметром потока на единицу. Если он становится равным нулю, то пытается приостановить выполнение потока.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/thread.c#L626

ResumeThread

Увеличивает счетчик "сонливости" указанного параметром потока на единицу. Если он был равен нулю, то пытается возобновить выполнение потока.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/thread.c#L551

Get/SetThreadPriority

Функции GetThreadPriority и SetThreadPriority позволяют запросить и установить приоритет потока, указанного параметром. Это значение используется планировщиком, как указано далее.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/thread.c#L723
- GetThreadPriority

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/thread.c#L684
- SetThreadPriority

Волокна в Windows NT

Волокна (Fiber) - одна из реализаций концепции потоков в пространстве пользователя в Windows. Это классический вариант, которому присущи все преимущества и недостатки теоретической модели. У каждого волокна имеется небольшой кусочек памяти, Fiber-Local Storage (FLS) по аналогии с поточным TLS, где оно может сохранять какие-нибудь полезные для себя данные.

CreateFiberEx

Создает новое волокно.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/fiber.c#L169

SwitchToFiber

Переключается с текущего волокна на другое, реализуя кооперативную многозадачность.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/amd64/fiber.S#L25 (не реализовано в ReactOS)

ConvertThreadToFiberEx

Создает внутри текущего потока волокно и устанавливает его в качестве текущего исполняемого. Начиная с этого момента внутри потока становится возможным переключение между волокнами.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/fiber.c#L77

ConvertFiberToThread

Уничтожает текущее волокно текущего потока, превращая его в обычный поток, внутри которого более невозможны переключения между волокнами.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/fiber.c#L42

DeleteFiber

Уничтожает произвольное волокно.

https://github.com/reactos/reactos/blob/master/dll/win32/kernel32/client/fiber.c#L268

UMS-потоки

UMS-потоки (User-Mode Scheduling) - потоки режима пользователя, доступные в 64-разрядных версиях Windows NT, начиная с Windows 7. Планирование выполнения потоков по-прежнему ложится на плечи прикладного программиста и осуществляется в кооперативном режиме, но в отличие от ранее существовавших волокон (fibers), ядро ОС знает о существовании внутри каждого потока UMS-потоков, и при выполнении блокирующего системного вызова может сообщить планировщику пользовательского уровня о необходимости передиспетчеризации.

(В данном разделе нет ссылок на исходный код, поскольку ReactOS не реализует такую возможность).

EnterUmsSchedulingMode

Указывает, что в данном потоке будут использоваться UMS-потоки. Принимает в качестве параметра адрес процедуры-планировщика UMS-потоков.

CreateRemoteThreadEx

Уже знакомая процедура, вызванная с флагом PROC_THREAD_ATTRIBUTE_UMS_THREAD, позволяет создать вместо обычного потока UMS-поток.

CreateUmsCompletionList

Выполняется планировщиком. Создает структуру, куда ядро ОС будет помещать освободившиеся от блокировки и готовые к исполнению UMS-потоки.

UmsThreadYield

Выполняется UMS-потоком в момент, когда он готов передать управление другому UMS-потоку.

ExecuteUmsThread

Функция, при помощи которой планировщик UMS-потоков запускает один из готовых потоков на исполнение.

Диспетчеризация

Планировщик ОС Windows NT реализует вытесняющую схему планирования по приоритетам. Как было сказано ранее, каждый процесс и каждый поток имеют связанную с ними информацию для планировщика. Для процесса это **класс приоритета**, имеющий следующие значения:

- Idle (фоновый);
- Below normal (ниже нормального);
- Normal (нормальный);
- Above normal (выше нормального);
- High priority (высокоприоритетный);
- Realtime (реального времени).

Не любой процесс может запросить любой приоритет. Первый класс обычно выставляется процессам, которые не должны занимать много процессорного времени - например, хранителю экрана. Последние два класса нужно использовать с осторожностью: процесс с классом High priority перетянет на себя почти все процессорное время, а с Realtime - будет более приоритетным, чем большинство потоков ядра, в том числе, отвечающих за работу с устройствами. Со всеми вытекающими последствиями в виде замершей картинки, не отвечающих устройств ввода и так далее. Класс приоритета по умолчанию наследуется процессом от родителя, и может быть изменен вызовом SetPriorityClass, описанным выше.

В свою очередь, потоки также имеют показатель, учитываемый планировщиком - **уровень приоритета**, который может быть одним из:

- Idle (фоновый);
- Lowest (низший);
- Below normal (ниже нормального);
- Normal (нормальный);
- Above normal (выше нормального);
- Highest (наивысший);
- Time critical (критичный к времени).

Поток наследует уровень приоритета от создавшего его потока. Уровень приоритета может быть изменен вызовом SetThreadPriority, однако не любой поток может запросить любой приоритет.

На основе этих двух параметров - уровня приоритета потока и класса приоритета процесса, к которому принадлежит данный поток, по довольно сложной схеме вычисляется используемая планировщиком характеристика - **приоритет диспетчеризации** или **динамический приоритет**. Это целое число в диапазоне от 0 до 31. 0 соответствует самому низкому приоритету - по факту, пользовательские процессы никогда не могут его получить. С этим приоритетом работает системный процесс, который, когда процессор не занят абсолютно ничем, осуществляет всякую вспомогательную работу - подчищает память нулями, например. 31 соответствует самому высокому приоритету - такой поток монопольно займет все доступное ему процессорное время.

Priority Class Relative Priority	Real-Time	High	Above-Normal	Normal	Below-Normal	Idle
Time Critical (+Saturation)	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 (–Saturation)	16	1	1	1	1	1

alt text

Для каждого значения приоритета диспетчеризации ядро ОС поддерживает связанный список потоков (группировка потоков по процессам здесь не учитывается), которые выполняются на данном уровне приоритета. Каждый квант времени планировщик выбирает поток на выполнению по простому алгоритму:

1. Выбрать N = 31.
2. Просмотреть список потоков с приоритетом диспетчеризации уровня N.
    1. Если есть готовый к выполнению поток - отправить его на исполнение.
    2. Если нет готового к выполнению потока, то N = N - 1 и перейти на шаг 2.

Отметим, что алгоритм корректен, так как служебный поток с приоритетом 0 готов к исполнению всегда.

Форсирование приоритета

Стоит отметить, что кроме ручного указания уровня приоритета программистом, он также может быть изменен (в большую сторону) самой системой. Такое изменение носит временный характер и называется **форсированием приоритета** (priority boost). Ему подвергаются только потоки процессов, не принадлежащих к классу приоритета Realtime. Происходит это в следующих случаях:

- поток вышел из ожидания ресурса (например, завершена операция ввода/вывода);

- поток обслуживает события окна, находящегося в текущий момент на переднем плане;
- поток долгое время не выполнялся (например, спал).

Поскольку форсирование носит временный характер, приоритет потока после нескольких выделенных ему квантов времени возвращается к своему изначальному значению.

Processor Affinity в Windows NT

В Windows NT существует возможность задать предпочтения по процессорам не только для конкретного потока, но и для всего процесса в целом (т.е. те потоки, для которых affinity не указано явно, будут наследовать его от affinity процесса), при этом affinity потоков не должно конфликтовать с affinity процесса (affinity потока не может содержать процессоры, которых нет в affinity процесса). Windows NT сродство к процессору задается при помощи битовой маски, в которой каждый бит отвечает за то, может ли процесс или поток выполняться на соответствующем процессоре, или нет. Кроме того, можно указать прямо "самый-самый" желаемый процессор из всего перечня.

GetProcessAffinityMask

Позволяет получить текущую битовую маску предпочитаемых процессоров для процесса.

Параметры:

- Хэндл процесса.
- Указатель на переменную, куда будет записана маска affinity процесса.
- Указатель на переменную, куда будет записана системная маска affinity.

SetProcessAffinityMask

Позволяет установить текущую битовую маску предпочитаемых процессоров для процесса.

Параметры:

- Хэндл процесса.
- Переменная, задающая битовую маску affinity процесса.

GetThreadAffinityMask

Позволяет получить текущую битовую маску предпочитаемых процессоров для потока.

Параметры:

- Хэндл потока.
- Указатель на переменную, куда будет записана маска affinity потока.

SetThreadAffinityMask

Позволяет установить текущую битовую маску предпочитаемых процессоров для потока.

Параметры:

- Хэндл потока.
- Переменная, задающая битовую маску affinity потока.

SetThreadIdealProcessor

Позволяет задать самый предпочитаемый процессор для конкретного потока. По возможности ОС будет запускать поток именно на нем.

Параметры:

- Хэндл потока.
- Переменная, задающая номер желаемого процессора.

Потоки в C#

В этом разделе речь пойдёт о наиболее интересных для данной работы *низкоуровневых* компонентах языка и платформы .NET.

class Thread

https://msdn.microsoft.com/ru-ru/library/system.threading.thread(v=vs.110).aspx

Класс-представление потока ОС. Очень тонкая обёртка над функциями WinAPI. Предоставляет средства создания, запуска, приостановки, завершения работы потока, установки приоритета и прочее.

Конструктор

Принимает делегат - функцию, которая будет непосредственно выполняться в отдельном потоке. Эта функция в качестве параметра принимает один единственный `object`.

Start()

Именно данный метод отвечает за запуск потока на выполнение. Может принимать параметр, который передаст на вход функции потока (см. выше).

Join()

Блокирующая операция: выполняет *ожидание завершения* потока, при этом никаким образом не влияя на него.

Thread.Sleep(milliseconds)

Статический метод, приостанавливающий выполнение потока, из которого был вызван, на заданное количество миллисекунд.

Небольшое замечание: представим, что у вас есть некий фоновый поток, который выполняет задачу в бесконечном цикле. Указанный метод *желательно*, если не обязательно, вызывать в конце итерации даже если пауза не нужна (тогда передавать 0 в качестве аргумента) - это позволяет планировщику более оптимально распределять процессорное время между всеми потоками, не допуская его монополизации кем-то из них.

Свойство Priority

Устанавливает/получает приоритет потока. Очень внимательно нужно отнестись к примечанию к этому свойству на MSDN: ОС **не обязана** следовать указанному приоритету. Как это интерпретировать? За поддержку установки приоритетов в первую очередь будет отвечать ОС, а не среда .NET, и в семействе Windows NT этот механизм работает как ожидается, но уже в GNU/Linux в среде Mono (открытая реализация .NET Framework) установка приоритетов имеет свои весьма серьезные особенности - необходимо указать ядру ОС, что процесс должен управляться одним из алгоритмов планирования реального времени (по умолчанию это не так), причем сделать это можно только имея права суперпользователя (root) и только из нативного кода (технически возможен вариант с P/Invoke, но тут и так костылей хватает).

Свойство IsAlive

True, если поток был запущен и не был завершён.

Suspend(), Abort() и их поведение

Методы, отвечающие за приостановку и завершение работы потока соответственно.

Как ни странно, они **не** являются безопасными и имеют не совсем очевидное поведение: они осуществляют прерывание работы потока в непредсказуемом месте - например, внутри критической секции, что может привести к dead lock'у. Отличия Abort() от Suspend() в том, что первый выбрасывает исключение внутри данного потока, тем самым завершая его работу, а значит в некоторых ситуациях (очистка неуправляемых ресурсов в конце работы потока) требуется дополнительные действия со стороны программиста (например, "оборачивание" функции потока в try..finally).

Как же тогда правильно (при)останавливать поток? Предлагается обратить внимание на класс - событие синхронизации потоков:

class ManualResetEvent

https://msdn.microsoft.com/ru-ru/library/system.threading.manualresetevent(v=vs.110).aspx

Событие синхронизации потоков.

Обладает двумя состояниями: сигнальным и несигнальным соответственно. При установке сигнального состояния (метод `Set()`) все ожидающие потоки (см. `WaitOne()`) продолжат выполнение. При установке несигнального состояния (метод `Reset()`) произойдёт блокировка потоков.

Конструктор

Позволяет указать исходное состояние события (сигнальное/несигнальное).

WaitOne()

Версия без параметров блокирует выполнение текущего потока до тех пор, пока событие не будет переключено в сигнальное состояние.

Версия с единственным параметром - временем ожидания в миллисекундах - вернёт истину, если за указанное время произошло переключение состояния в сигнальное, иначе вернёт ложь.

Пример использования

Рассмотрим реализацию завершения потока, выполняющего "бесконечный" цикл. Для этого в глобальной по отношению к потоку области объявим переменную `exitEvent` этого класса. Тогда поток может быть приостановлен следующим образом:

```
void threadTask(object arg) {
    // startup stuff
    do {
        // something very important
    } while (!exitEvent.WaitOne(0));
    // cleanup & stuff
}
```

Здесь в конце каждой итерации проверяется, не было ли извне установлено событие `exitEvent`, причём это делается в "неблокирующем" режиме (нулевое ожидание события). В случае, если событие было установлено, метод WaitOne() вернёт true и цикл будет завершён.

Краткое введение во взаимное исключение в C#

Наиболее простым для понимания способом последовательного доступа к разделяемому ресурсу из нескольких потоков является стандартный оператор языка - `lock()`. Синтаксис:

```
lock (locker) {
    // critical section
}
```

Здесь `locker` - некий объект блокировки, с помощью которого система будет определять, находится ли какой-либо поток внутри критической секции. В принципе может быть любым объектом (абсолютно!), однако *рекомендуется* создавать отдельные приватные объекты для этого. Код внутри критической секции в один момент времени будет выполнятся ровно одним потоком.

Соответственно, может быть несколько критических секций с одним и тем же объектом блокировки, тогда в любой момент времени только один поток может находится внутри какой-либо *одной* из них, что логично: блокируется тот самый объект.

Взаимное исключение и WinForms

https://msdn.microsoft.com/ru-ru/library/zyzhdc6b(v=vs.110).aspx

Как любой приличный UI Framework, WinForms обеспечивает обработку событий интерфейса в отдельном потоке - т.н. UI thread. Его примерная реализация была рассмотрена в лабораторной работе по WinAPI и представляет собой обработку очереди сообщений в цикле.

Как только возникает потребность в доступе к полям и методам компонентов интерфейса из другого потока, возникает и необходимость во взаимном исключении при доступе к ним. Однако само взаимное исключение примет специфический вид: WinForms обязывает программиста осуществлять взаимодействие с интерфейсом *исключительно* из UI thread - в нём создаются все дескрипторы элементов интерфейса - а значит механизм `lock'a` здесь не применим.

В данном случае потребуется использовать метод `Invoke()` класса `Control`. Он позволяет выполнить произвольную функцию (делегат) в потоке, которому принадлежит элемент управления `Control`. Вызов является блокирующим. При вызове этой функции делегат будет добавлен в очередь сообщений, после чего произойдёт ожидание его выполнения потоком UI.

Есть также неблокирующий аналог - `BeginInvoke()`.

Потоки в C++

C++, будучи максимально кроссплатформенным языком, долгое время не поддерживал работу с потоками, поскольку далеко не на всех программно-аппаратных платформах она доступна. Однако начиная со стандарта 2011 года стандартная библиотека C++11 имеет встроенный класс, который реализует самый минимум операций над потоками - `std::thread`.

std::thread::thread (конструктор)

Создает новый поток и немедленно запускает его. В качестве аргумента указывается "объект, который может быть вызван" - в простейшем случае это лямбда-функция либо имя обычной функции.

std::thread::join

Блокирует вызывающий поток до тех пор, пока не завершится поток-объект.

std::thread::detach

Отправляет поток в "свободное плавание", позволяя ему выполняться даже после того, как завершится породивший его поток.

std::thread::get_id

Возвращает некоторый уникальный идентификатор потока.

std::thread::native_handle

Возвращает уникальный идентификатор потока, используемый операционной системой.

std::thread::joinable

Проверяет, возможно ли у потока вызвать метод join (то есть валиден ли поток и не был ли уже вызван join для него).

std::this_thread::sleep_for

Блокирует выполнение текущего потока **не менее чем** на указанный период времени.

std::this_thread::sleep_until

Блокирует выполнение текущего потока до наступления **как минимум** указанного момента времени.

Ограничения

Как можно заметить, стандарт C++ не предоставляет операций по изменению приоритета потока, его принудительному завершению (попытка удалить объект выполняемого в текущий момент потока вызовет исключение) и вообще достаточно ограничен в выразительных средствах. Это сделано из-за все тех же соображений межплатформенной переносимости. В случае острой необходимости можно воспользоваться смесью потоков C++ с нативными функциями операционной системы при помощи `std::thread::native_handle`:

```
std::thread *m_pThread = new std::thread({ while(1); });
SetThreadPriority(th->native_handle(), THREAD_PRIORITY_TIME_CRITICAL);
```

Описание примера

Рядом с данными заметками расположен демонстрационный пример графического приложения на языке C#. Пример занимается отрисовкой окружностей из двух потоков на экранную форму, обеспечивает возможность приостанавливать один из них, а также изменять приоритет потоков с нормального на заданный в исходном коде. В приложении реализовано сразу два класса, отвечающих за поток отрисовки кругов, один из них значительно переусложнён, однако достаточно документирован, другой же значительно упрощён, но и фактически без документации.

С целью повышения самостоятельности студентов, из обеих реализаций были удалены значимые фрагменты (...) и заменены на плейсхолдеры (тег `TODO`, исключение `NotImplementedException`). Студентам, взявшим этот пример в качестве основы для реализации данной работы предлагается самостоятельно (с помощью документации в исходном коде, теоретического материала, а также документации в MSDN и иных источниках) восстановить недостающие фрагменты и тем самым выполнить основные критерии задания.