Операционные системы, процессы и потоки

Операционные системы

ОС — это комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как **интерфейс** между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для **управления** устройствами и вычислительными процессами, эффективного распределения **ресурсов** между вычислительными процессами и организации надёжных вычислений.

Место ОС в программном обеспечении компьютера показано на следующем рисунке


Прикладные программы
Дополнительное системное ПО
Пользовательское окружение
Операционная система
Ядро операционной системы (супервизор)
Встроенное программное обеспечение (firmware)
Системное программное обеспечение
Программное обеспечение

Ядро системы

Ядро ОС — это центральная часть операционной системы, обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешнее устройство ввода и вывода информации. Также обычно ядро предоставляет сервисы файловой системы и сетевых протоколов.

Варианты реализации ядра:

- монолитное: одна монолитная программа в памяти
- модульное: монолитная программа, предоставляющая интерфейс загрузки и выгрузки доп.модулей
- микроядро: несколько программ, которые взаимодействуют через передачу сообщений
- наноядро: ядро только управляет ресурсами (обработка прерываний)
- экзоядро: наноядро с координацией работы процессов
- Гибридное

Пример ядра современной ОС Windows показан на рисунке

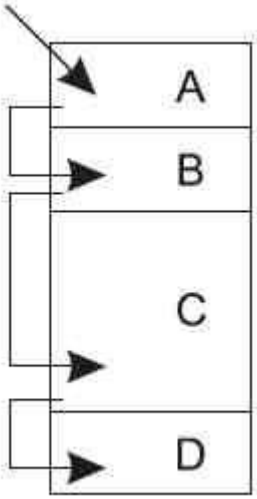
Процессы, планирование процессов

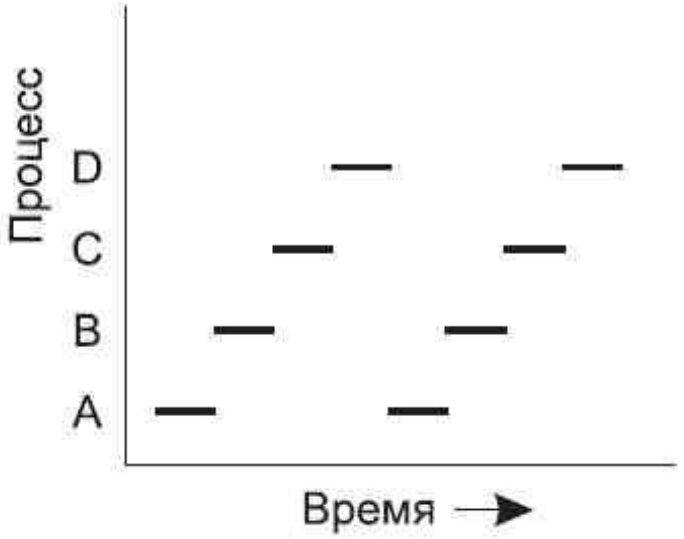
Понятие процесса включает в себя:

- Программу которая исполняется
- Ее динамическое состояние (регистровый контекст, состояние памяти и т.д.)
- Доступные ресурсы (как индивидуальные для процесса, такие как дескрипторы файлов, так и разделяемые с другими)

В любой многозадачной системе центральный процессор быстро переключается между процессами, предоставляя каждому из них десятки или сотни миллисекунд. При этом, хотя в каждый конкретный момент времени центральный процессор работает только с одним процессом, в течение 1 секунды он может успеть поработать с несколькими из них, создавая иллюзию параллельной работы. Это постоянное переключение между процессами называется **многозадачным режимом работы**.

На рисунке слева показан компьютер, работающий в многозадачном режиме и имеющий в памяти четыре программы. На рисунке справа показано, что за довольно длительный период наблюдения продвинулись вперед все процессы, но в каждый отдельно взятый момент времени реально работает только один процесс.





Системы разделения времени реальных ОС, в большинстве своём, работают по схеме круговорота (Round Robin).

Процессорное время в данной схеме квантуется, то есть разделяется на элементарные единицы – кванты. Планировщик задач ОС (модуль, который реализует управление процессами) выделяет задачам процессорное время квантами. По завершении кванта времени планировщик задач принимает решение, какому процессу выделить следующий квант. Выбор слишком маленького кванта времени приводит к увеличению непроизводительного расхода процессорного времени – ведь после каждого кванта планировщик должен определять, кому предоставить следующий, а на это тоже затрачивается процессорное время. При выборе слишком большого размера кванта снижается оптимальность распределения времени.

Конкретный размер кванта зависит от ОС, а в некоторых ОС может настраиваться.

На следующем рисунке показана схема круговорота.

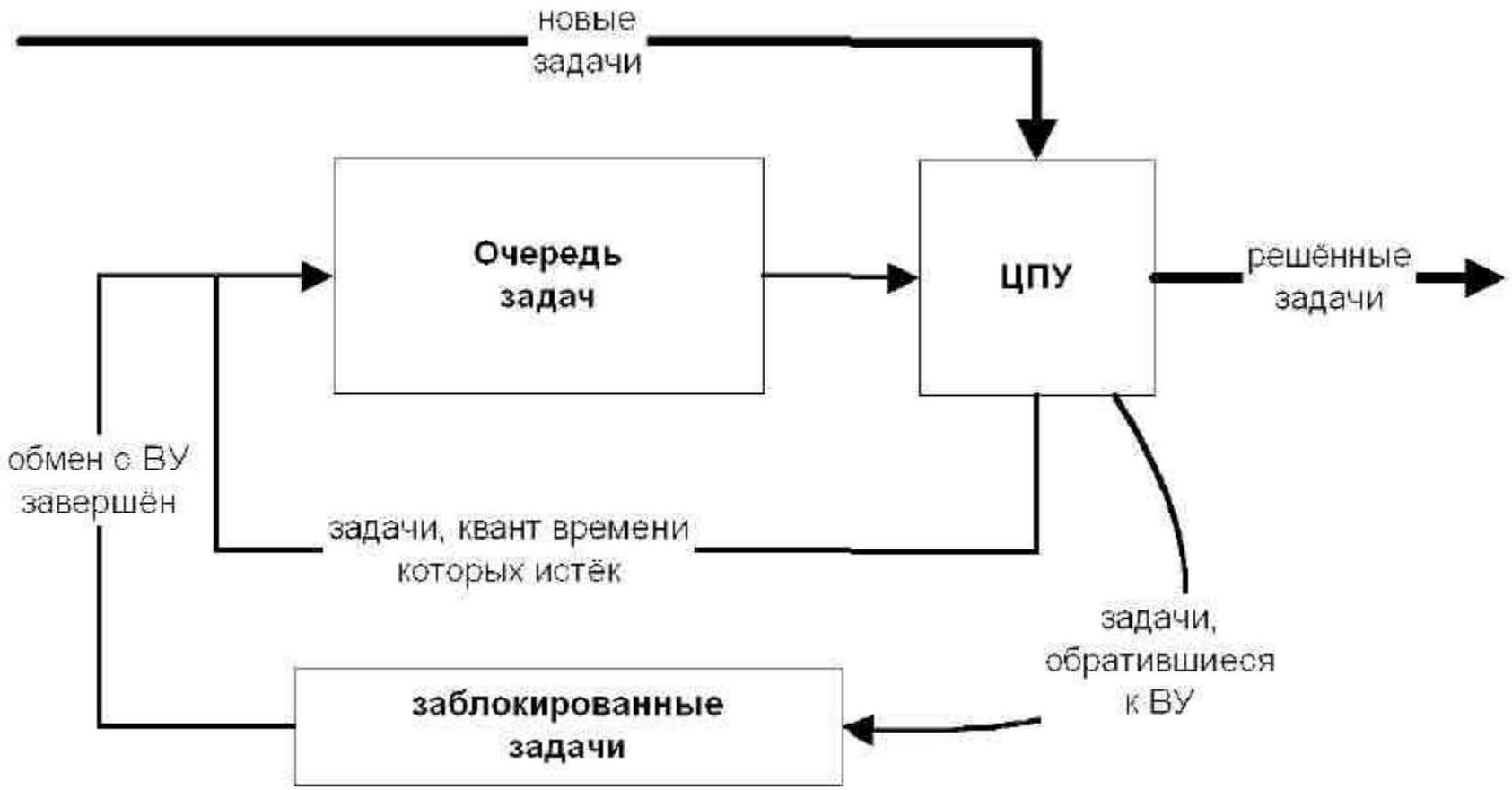

новые
задачи
Очередь
задач
ЦПУ
решённые
задачи
обмен с ВУ
завершён
задачи, квант времени
которых истёк
задачи,
обратившиеся
к ВУ
заблокированные
задачи

Схема функционирует следующим образом. Поступающая в систему задача сразу ставится на процессор. Если на момент поступления новой задачи процессор уже занят какой-то другой задачей, то эта задача снимается с процессора и ставится в очередь. Поступившая задача выполняется в течение одного кванта времени, либо до момента, когда ей потребуется обратиться к ВУ, либо до момента поступления в систему новой задачи, в зависимости от того, что случится раньше. Если истёк квант времени, но задача ещё не решена, то задача ставится в конец очереди и ждёт, а на процессор поступает первая задача из очереди. Если пришла новая задача, то она ставится на процессор, не дожидаясь конца кванта, а решаемая до этого задача ставится в очередь. Если задача завершилась, то она покидает систему, а планировщик завершает текущий квант времени и начинает следующий. Если, наконец, задача обратилась к ВУ, то она снимается с процессора и блокируется на время обращения, а после работы с ВУ ставится в конец очереди к процессору.

Схема круговорота обычно используется с одним важным дополнением – с учётом **приоритетов** задач. Каждой задаче перед первой постановкой на процессор пользователем или системой приписывается целое число, называемое приоритетом. Одни значения приоритета считаются более высокими, другие — более низкими. Считается, что высокоприоритетные задачи более важны, чем низкоприоритетные, и поэтому они должны решаться быстрее. Схема разделения времени функционирует при этом точно так же, как было описано, но при постановке задачи в очередь к процессору она ставится не в конец очереди, а после самой последней в очереди задачи с тем же приоритетом или, если таких задач нет, после последней задачи с более высоким приоритетом. Если в системе появляется задача с приоритетом более высоким, чем приоритет задачи, находящейся на процессоре, то процессор немедленно освобождается для более высокоприоритетной задачи.

В некоторых случаях оказывается желательным предоставление простым задачам дополнительных преимуществ перед сложными.

Это бывает нужно, когда, наряду с постоянной загрузкой системы относительно сложными задачами, время от времени возникают задачи простые, но требующие быстрого решения. В таких случаях может применяться система **динамических приоритетов**.

Эта система аналогична круговороту с приоритетами, но приоритет задачи не задаётся изначально, а изменяется по ходу её выполнения. Первоначально, при входе задачи в систему, ей присваивается наивысший приоритет. После каждого снятия задачи с процессора её приоритет становится более низким. В очереди к процессору задачи располагаются в порядке уменьшения приоритета. Чем дольше задача выполняется, тем реже, соответственно, ей предоставляется очередной квант времени. При такой схеме короткая задача, требующая всего несколько квантов времени, попадает на процессор часто. Если же задача в ходе выполнения набирает наименьшее значение динамического приоритета среди всех задач в системе, то она сможет попадать на процессор только тогда, когда тот не будет занят никакой другой задачей, то есть превратится в фоновую задачу.

Вытесняющая (preemptive) многозадачность

Ядро операционной системы является, как и все задачи пользователя, процессом, запущенным на том же компьютере. Однако, в силу своего особого положения, ядро выполняется несколько необычным образом.

В состав ядра входит планировщик задач ОС. Планировщик задач должен периодически получать управление, чтобы обеспечивать переключение процессора с одной задачи на другую. Естественно, модуль ядра, содержащий планировщик, является привилегированным. Он может в любой момент вытеснить с процессора любую задачу, его же не может вытеснить никто. Планировщик запускается периодически, независимо ни от каких событий, происходящих в системе.

Порядок работы остальных компонентов ядра во многом зависит от его архитектуры. В ОС с монолитным ядром всё ядро (в силу своей нераздельности) имеет тот же приоритет, что и планировщик задач. Это означает, что выполнение любой операции в ядре не может быть приостановлено планировщиком задач в интересах какого-то другого процесса. Поэтому когда в ядре системы происходит какая-нибудь задержка, вся система на время этой задержки «подвисает» и никакие программы выполняться не могут. Естественно, в правильно спроектированных ядрах такого рода задержки редки и не особенно велики, но, тем не менее, они имеют место.

В ОС с микроядром ситуация достаточно сильно отличается. Само микроядро содержит, можно считать, только планировщик задач. Все остальные модули ядра выполняются на общих основаниях, хотя и могут иметь больший, чем программы пользователя, приоритет. Следовательно, прикладная программа, выполняющаяся в системе, вполне может «подвинуть» выполняющуюся на процессоре операцию ядра (например, связанную с вводом-выводом). Именно по этой причины микроядерные системы (или системы, в которых реализованы сходные механизмы), иногда называют «системами с вытесняемым ядром», хотя этот термин и не вполне точен.

Таким образом, большая средняя производительность систем с монолитными ядрами достигается за счёт привилегированного режима исполнения *всех* операций ядра, но эта же привилегированность является причиной увеличения времени отклика. Низкое время отклика микроядерных систем достигается за счёт того, что меньший объём кода исполняется в привилегированном режиме, но, по той же причине, в некоторых случаях может снизиться средняя производительность системы.

Кооперативная (cooperative) многозадачность

Вытесняющая многозадачность удобна и надёжна, но для её эффективной реализации аппаратная часть системы должна содержать специальные поддержки, обеспечивающие выполнение необходимых действий (таких как переключение задач). Одним из способов реализации многозадачности в системах, не имеющих для этого соответствующих аппаратных средств, является кооперативная многозадачность. В системе с кооперативной многозадачностью ОС не в состоянии в произвольный момент времени вытеснить выполняющуюся задачу с процессора. Переключение процессов происходит в моменты, когда процессы обращаются к внешним устройствам или ожидают от пользователя ввода данных или команды. Для выполнения ввода-вывода процесс обычно вызывает функции API, а эти функции написаны таким образом, что при их вызове ОС производит переключение процессора с одной задачи на другую.

Для того чтобы программы, не обращающиеся к функциям ввода-вывода, могли выполняться параллельно в такой системе, в ней к написанию программ предъявляются специальные требования, заключающиеся в следующем. Программа, написанная для ОС с кооперативной многозадачностью, если она содержит достаточно длительные операции, не связанные с вводом-выводом, должна во время выполнения этих операций достаточно часто вызывать некоторую стандартную функцию API. При вызове этой функции задача приостанавливается и активизируется ОС. ОС принимает решение, какую задачу поставить на процессор далее, и запускает её. Таким образом, задача в тех случаях, когда её не может снять с процессора ОС, фактически снимает себя с процессора самостоятельно. Естественно, чтобы задачи выглядели выполняющимися параллельно, вызовы ОС из них должны производиться достаточно часто.

Единственным плюсом кооперативной многозадачности является простота её реализации. Платой за эту простоту является принципиальная неустойчивость системы по отношению к неправильно работающим прикладным программам. Если запущенная задача не активизирует ОС, то эта задача единолично захватывает процессор. При этом остальные задачи и ядро самой ОС будут просто стоять. Если одна задача зависнет, то вместе с нею зависнет вся система.

Диаграмма состояний процесса

В многозадачных операционных системах процесс может находиться в одном из трех основных состояний. Это «выполнение», «ожидание», «готовность».

«**Выполнение**» - активное состояние процесса. В данном состоянии процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором.

«**Ожидание**» - пассивное состояние процесса. Процесс заблокирован и не может выполняться по своим внутренним причинам. Такими причинами могут являться: ожидание завершения операции ввода-вывода; получение сообщения от другого процесса; освобождение необходимого для продолжения вычислений ресурса.

«**Готовность**» - также пассивное состояние процесса. В этом состоянии процесс заблокирован в связи с внешними причинами, по инициативе операционной системы. Процесс имеет все требуемые для выполнения ресурсы, однако процессор занят выполнением другого процесса.

В ходе своего выполнения каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе

В состоянии «выполнение» в однопроцессорной системе может находиться только один процесс. В каждом из состояний «ожидание» и «готовность» - несколько процессов. Эти процессы образуют очереди. Исполнение процесса начинается с состояния «готовность». В данном состоянии он находится в очереди планировщика процессов операционной системы. При активизации процесс переходит в состояние «выполнение» и находится в нем до тех пор, пока: ему не потребуется ждать некоторого события; он сам не освободит процессор; он будет принудительно вытеснен планировщиком.

Переходя в состояние «ожидания», процесс помещается в очередь, связанную с конкретным событием, которое он ожидает. Например, процесс может попасть в очередь процессов, ожидающих завершения ввода-вывода.

Если процесс вытесняется или добровольно отдает управление планировщику, он попадает в состояние «готовность» и помещается в очередь планировщика. В это же состояние процесс переходит из состояния «ожидание» после того, как произойдет ожидаемое событие.

Информационные структуры процесса

Информационные структуры, которые используются для управления исполнением процессов, называются ***контекст*** и ***дескриптор***. Программный код только тогда начнет выполняться, когда для него операционной системой будет создан процесс. Создание процесса состоит из трех этапов: создания дескриптора и контекста процесса; включения дескриптора нового процесса в очередь готовых процессов; загрузки кодового сегмента процесса в оперативную память.

На протяжении существования процесса его выполнение может быть многократно прервано и продолжено. Для того, чтобы возобновить выполнение процесса, необходимо восстановить состояние его операционной среды. Состояние операционной среды состоит из значений регистров и программного счетчика, режима работы процессора, указателей на открытые файлы, информации о незавершенных операциях ввода-вывода, кодов ошибок, выполняемых данным процессом системных вызовов. Эта информация называется ***контекстом процесса***. Контекст является зависимой от аппаратуры структурой данных.

Операционной системе для реализации планирования процессов требуется дополнительная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, данные о нахождении процесса в очередях, указатель на контекст процесса. Эта информация называется ***дескриптором процесса***. Содержание информационных полей дескриптора определяется разработчиком операционной системы, зависит от особенностей алгоритма планирования и не зависит от аппаратуры.

Очереди процессов представляют собой дескрипторы процессов, объединенные в списки. Поэтому каждый дескриптор, содержит, по крайней мере, один указатель на другой дескриптор, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

Процессы в Win32 API

С точки зрения ОС процесс является объектом ядра. Для того чтобы запустить какую-либо программу, необходимо создать объект ядра «процесс» с помощью функции *CreateProcess*, входящей в API. Функция возвращает true, если процесс создан, и false, если произошла ошибка.

По завершении работы с процессом вызывающая программа должна закрыть его хэндл вызовом функции *CloseHandle*. При этом, если процесс ещё выполняется, он не будет завершён. Процесс продолжит выполнение, просто к нему уже не будет доступа из родительского процесса.

Процесс завершается, когда из него вызывается функция API *ExitProcess*. Данная функция имеет один параметр — код завершения процесса, который передаётся в окружение, в котором был запущен процесс. Код может быть любым, он используется для того, чтобы сигнализировать о том, как завершился процесс (корректно, из-за ошибки и т.п.). При завершении процесса автоматически завершаются все потоки, закрываются объекты ядра и уничтожаются все остальные объекты ОС, созданные процессом, освобождается память, занятая процессом. При написании программы обычно нет необходимости в вызове *ExitProcess*, поскольку этот вызов добавляется компилятором автоматически после завершения выполнения всего кода программы.

Процесс может также быть остановлен с помощью вызова предназначенной для этого функции API *TerminateProcess*. Этой функцией можно остановить процесс не из него самого, а из любого процесса в системе, который имеет хэндл останавливаемого процесса. Например, из процесса, запустившего данный. Процесс останавливается также, как и после вызова ExitProcess. Необходимо отметить, что завершение процесса функцией TerminateProcess не является безопасным. Хотя ОС и удаляет объекты, созданные процессом и освобождает память, выделенную ему, но внезапная остановка процесса почти всегда приводит к нарушению логики его функционирования, что нежелательно.

При разработке приложений бывает полезно группировать процессы. Например, когда вы прерываете работу с проектом в Visual Studio, среда должна каким-то образом завершить все свои дочерние процессы.

Допустим, клиентская программа просит сервер выполнить приложение (которое создает ряд дочерних процессов) и сообщить результаты Поскольку к серверу может обратиться сразу несколько клиентов, было бы неплохо, если бы он умел как-то ограничивать ресурсы, выделяемые каждому клиенту, и тем самым не давал бы одному клиенту монопольно использовать все серверные ресурсы. Под ограничения могло бы подпадать процессорное время, выделяемое на обработку клиентского запроса.

В Wmdows 2000 введен новый объект ядра — задание (job). Он позволяет группировать процессы и помещать их в контейнер процессов. Бывает также полезно создавать задание и с одним процессом — это позволяет налагать на процесс ограничения, которые иначе указать нельзя.

Для задания могут быть установлены базовые и расширенные ограничения. Соответствующие структуры *JOBOBJECT_BASIC_LIMIT_INFORMATION*, *JOBOBJECT_BASIC_UI_RESTRICTIONS* описаны в *WinNT.h.* Задание создается функцией *CreateJobObject.* Ограничения устанавливаются функцией *SetInformationJobObject*, привязка процесса к заданию – функцией *AssignProcessToJobObject*.

Потоки

Обычный процесса имеет адресное пространство и единственный поток выполнения. Тем не менее нередко возникают ситуации, когда надо было бы иметь в одном и том же адресном пространстве несколько потоков выполнения, выполняемых параллельно. Понятие «поток выполнения» обычно сокращается до слова **поток**.

Основная причина использования потоков заключается в том, что во многих приложениях одновременно происходит несколько действий, часть которых может периодически быть заблокированной.

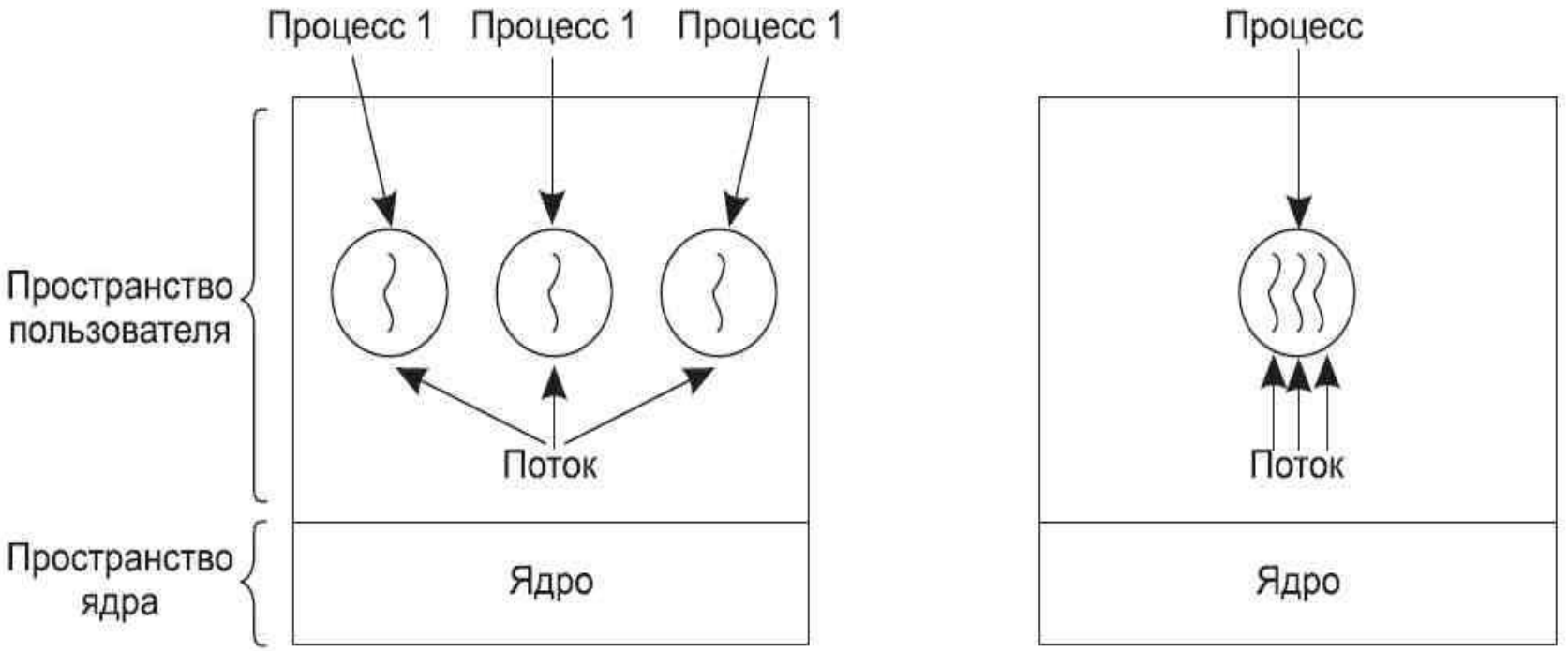
Вторым аргументом в пользу потоков является легкость (то есть быстрота) их создания и ликвидации по сравнению с более «тяжеловесными» процессами.

Третий аргумент в пользу потоков также касается производительности. Когда выполняются значительные вычисления, а также значительная часть времени тратится на ожидание ввода-вывода, наличие потоков позволяет этим действиям перекрываться по времени, ускоряя работу приложения.

И наконец, потоки весьма полезны для систем, имеющих несколько центральных процессоров, где есть реальная возможность параллельных вычислений.

В качестве примера предположим, что поток занимается переформатированием большого объема текста. В этом случае полезно добавить еще один поток, который взаимодействует с пользователем и управляет переформатированием Можно добавить и третий поток, который может заниматься созданием резервных копий на диске, не мешая первым двум.

На рисунке слева показаны три традиционных процесса. У каждого из них имеется собственное адресное пространство и единственный поток выполнения. В отличие от этого, на рисунке справа показан один процесс, имеющий три потока управления. Хотя в обоих случаях у нас имеется три потока, на рисунке слева каждый из них работает в собственном адресном пространстве, а на рисунке справа все три потока используют общее адресное пространство.



Реализация потоков

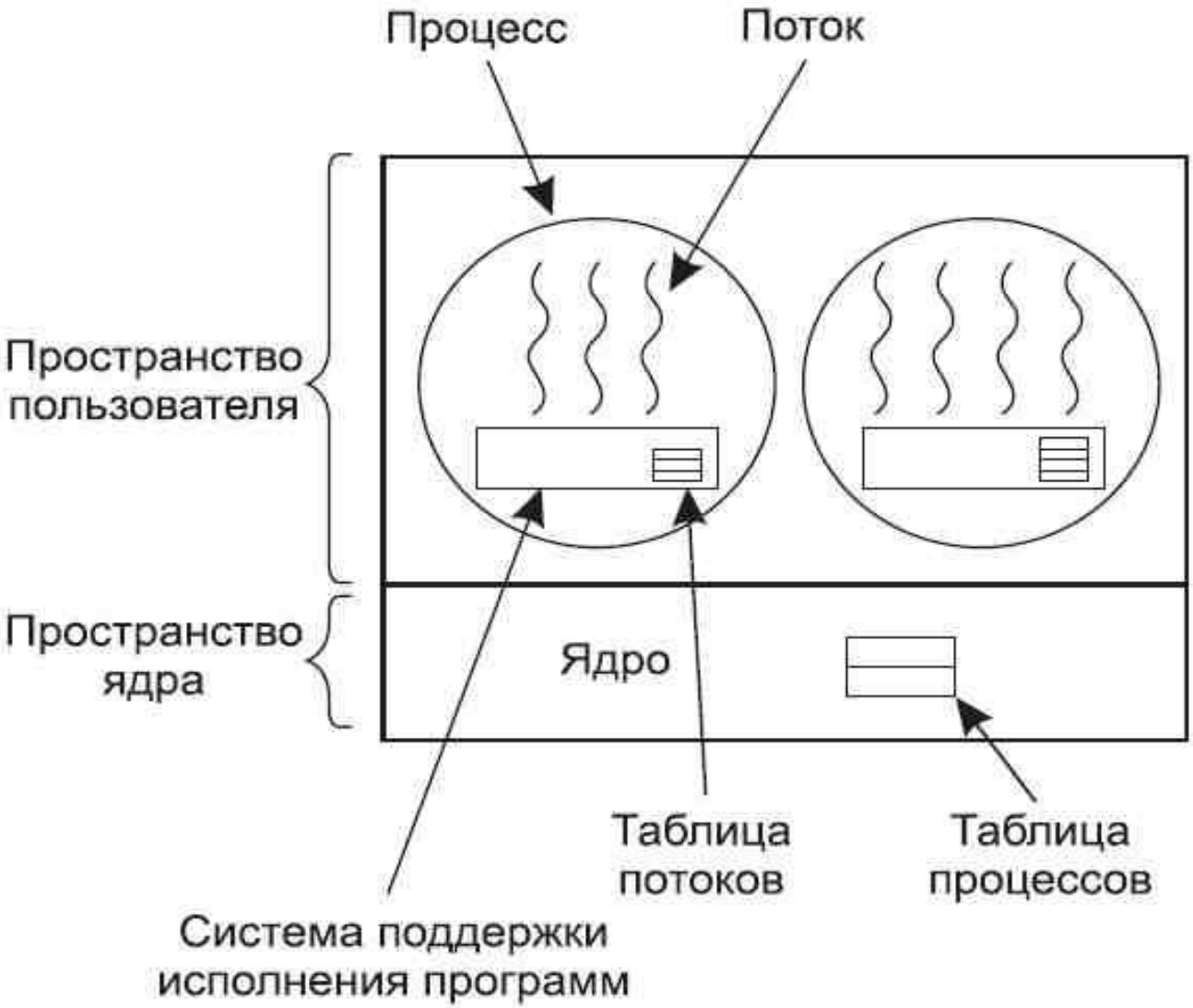
Есть два основных места реализации набора потоков: в пользовательском пространстве и в ядре.

Реализация потоков в пользовательском пространстве. Весь набор потоков находится в пользовательском пространстве. И об этом наборе ядру ничего не известно. Что касается ядра, оно управляет обычными, однопотоковыми процессами.

Данная реализация часто обозначается как N:1 (user-level threading).

Первое и самое очевидное преимущество состоит в том, что набор потоков на пользовательском уровне может быть реализован в операционной системе, которая не поддерживает потоки. При этом подходе потоки реализованы с помощью библиотеки.

У всех этих реализаций одна и та же общая структура.

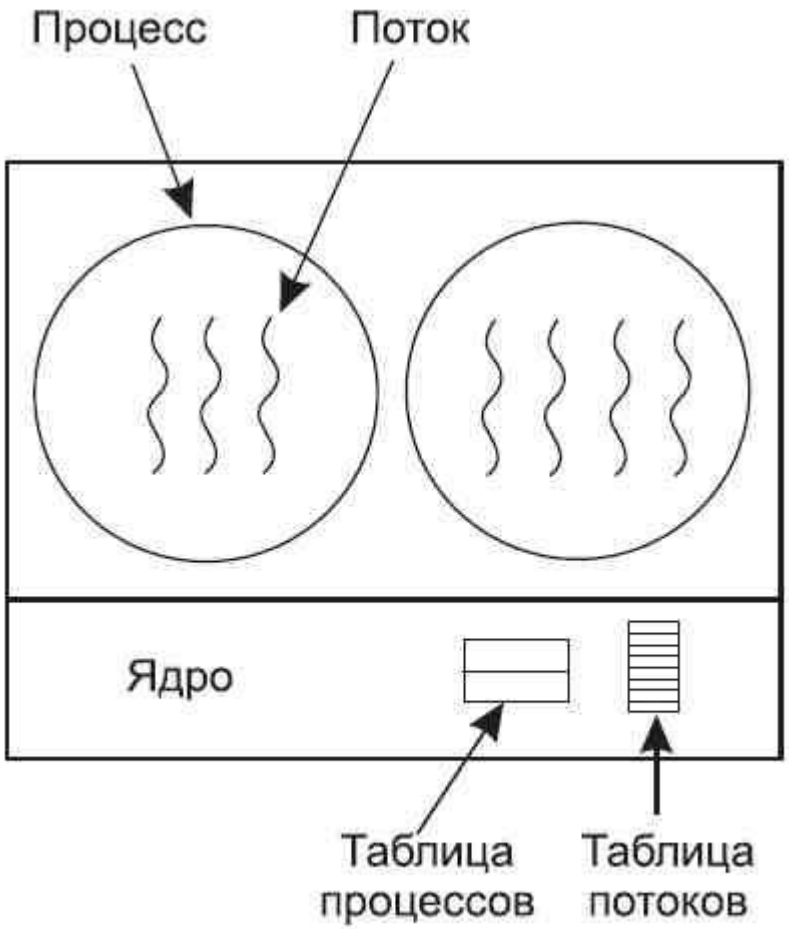

Процесс
Поток
Пространство пользователя
Пространство ядра
Ядро
Таблица потоков
Таблица процессов
Система поддержки исполнения программ

У потоков, реализованных на пользовательском уровне, есть и другие преимущества. Они позволяют каждому процессу иметь собственные настройки алгоритма планирования.. Эти потоки также лучше масштабируются, поскольку потоки в памяти ядра безусловно требуют в ядре пространства для таблицы и стека, что при очень большом количестве потоков может вызвать затруднения.

Но несмотря на лучшую производительность, у потоков, реализованных на пользовательском уровне, есть ряд существенных проблем. Первая из них — как реализовать блокирующие системные вызовы. Представим, что поток считывает информацию с клавиатуры. Мы не можем разрешить потоку осуществить настоящий системный вызов, поскольку это остановит выполнение всех потоков. Одна из главных целей организации потоков в первую очередь состоит в том, чтобы позволить каждому потоку использовать блокирующие вызовы, но при этом предотвратить влияние одного заблокированного потока на выполнение других потоков.

Реализация потоков в ядре При такой реализации ядро знает о потоках и может управлять ими. Как показано на рисунке, здесь уже не нужна система поддержки исполнения программ. Также здесь нет и таблицы потоков в каждом процессе. Вместо этого у ядра есть таблица потоков, в которой отслеживаются все потоки, имеющиеся в системе. Когда потоку необходимо создать новый или уничтожить существующий поток, он обращается к ядру, которое и создает или разрушает поток путем обновления таблицы потоков в ядре.

Такая реализация обозначается как 1:1 (kernel-level threading) и доступна в частности в Windows.


Процесс
Поток
Ядро
Таблица процессов
Таблица потоков

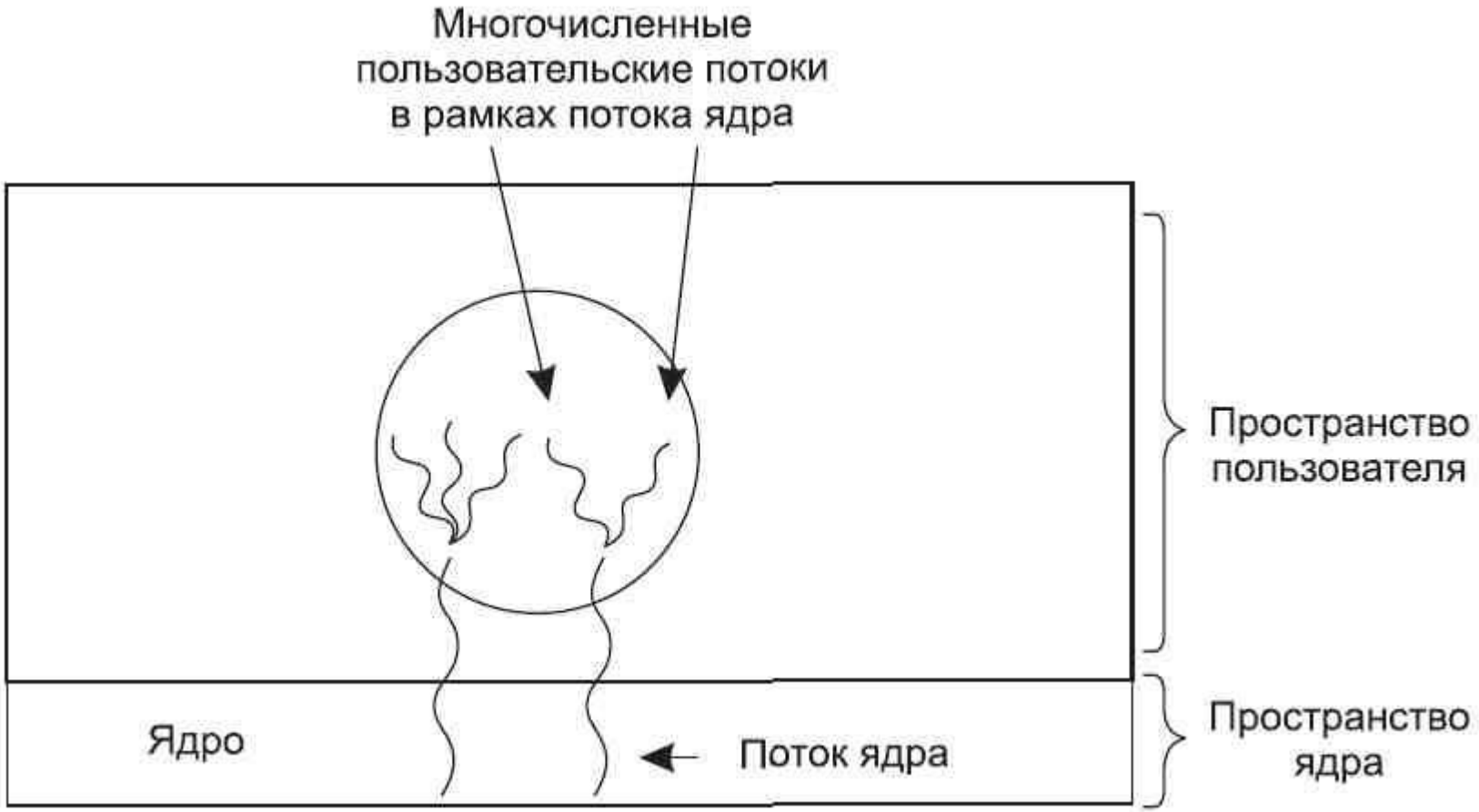
В таблице потоков, находящейся в ядре, содержатся регистры каждого потока, состояние и другая информация. Вся информация аналогична той, которая использовалась для потоков, создаваемых на пользовательском уровне, но теперь она содержится в ядре, а не в пространстве пользователя (внутри исполнительной системы). Эта информация является подмножеством той информации, которую поддерживают традиционные ядра в отношении своих однопоточных процессов. Вдобавок к этому ядро поддерживает также традиционную таблицу процессов с целью их отслеживания.

Все вызовы, способные заблокировать поток, реализованы как системные, с более существенными затратами, чем вызов процедуры в исполнительной системе. Когда поток блокируется, ядро по своему выбору может запустить либо другой поток из этого же самого процесса (если имеется готовый к выполнению поток), либо поток из другого процесса. Когда потоки реализуются на пользовательском уровне, исполнительная система работает с запущенными потоками собственного процесса до тех пор, пока ядро не заберет у нее центральный процессор или не останется ни одного готового к выполнению потока.

Хотя потоки, создаваемые на уровне ядра, и позволяют решить ряд проблем, но справиться со всеми существующими проблемами они не в состоянии. Что будет, к примеру, когда произойдет разветвление многопоточного процесса? Будет ли у нового процесса столько же потоков, сколько у старого, или только один поток? Во многих случаях наилучший выбор зависит от того, выполнение какого процесса запланировано следующим.

Гибридная реализация. В попытках объединить преимущества создания потоков на уровне пользователя и на уровне ядра была исследована масса различных путей. Один из них заключается в использовании потоков на уровне ядра, а затем нескольких потоков на уровне пользователя в рамках некоторых или всех потоков на уровне ядра. При использовании такого подхода программист может определить, сколько потоков использовать на уровне ядра и на сколько потоков разделить каждый из них на уровне пользователя. Эта модель обладает максимальной гибкостью.

При таком подходе ядру известно *только* о потоках самого ядра, работу которых оно и планирует.. В этой модели каждый поток на уровне ядра обладает определенным набором потоков на уровне пользователя, которые используют его по очереди.



Реализация обозначается как M:N (hybrid threading) и доступна, в частности, в последних версиях Windows.

Потоки в Win32 API

При создании процесса ОС автоматически создаёт первичный поток, в котором и выполняется код программы. Большинство программ обходятся этим потоком и не нуждаются в создании дополнительных. Когда оказывается удобным организовать программу так, чтобы в ней одновременно выполнялось несколько потоков, Win32 предоставляет такую возможность. Программа может создать, помимо первичного, любое количество потоков. Процесс считается завершённым, если завершились все потоки, созданные в нём. С этой точки зрения первичный поток не имеет никаких привилегий по сравнению с порождёнными потоками. Он может быть завершён до того, как завершатся остальные потоки, при этом процесс будет работать.

Поток представляет собой объект ядра Win32. Для создания нового потока нужно вызвать функцию *CreateThread*.

Функция возвращает хэндл созданного потока.

Для завершения запущенного потока из функции потока можно вызвать функцию *ExitThread* с одним параметром — кодом возврата. Если требуется завершить поток из другого потока, можно вызвать функцию *TerminateThread* с двумя параметрами – хэндлом останавливаемого потока и кодом возврата. Поток принудительно завершается. Использовать *TerminateThread* рекомендуется только в крайних случаях по той же причине, что и *TerminateProcess* – принудительное завершение потока может привести к нарушению логики его функционирования.

Для того, чтобы приостановить выполняющийся поток, можно вызвать функцию *SuspendThread* с одним параметром — хэндлом потока. Поток будет приостановлен — ему перестанут выделяться кванты процессорного времени.

Для запуска приостановленного потока используется функция *ResumeThread* с одним параметром — хэндлом потока. Её вызов приводит к продолжению выполнения потока, если он ранее был приостановлен функцией *SuspendThread* или если он был создан с флагом *CreateSuspended*.