**Планирование в системах реального времени**

В системах реального времени **существенную роль играет время**. Чаще всего одно или несколько внешних физических устройств генерируют входные сигналы, и **компьютер должен адекватно на них реагировать в течение заданного промежутка времени**.

Системы реального времени делятся на =

1. **жесткие системы** реального времени, что означает наличие жестких сроков для каждой задачи ***(в них обязательно надо укладываться),***
2. и **гибкие системы** реального времени, в которых нарушения временного графика нежелательны, но допустимы.

В обоих случаях реализуется **раз­деление программы на несколько процессов, каждый из которых предсказуем**. Эти **процессы чаще всего бывают короткими и завершают свою работу в течение секунды**. Когда появляется внешний сигнал, именно планировщик должен обеспечить соблюдение графика.

Внешние события, на которые система должна реагировать, можно разделить на=

1. **периодические** (возникающие через регулярные интервалы времени)
2. и н**епериодические** (возникающие непредсказуемо).

**Возможно наличие нескольких пери­одических потоков событий**, которые система должна обрабатывать. В зависимости от времени, затрачиваемого на обработку каждого из событий, **может оказаться, что система не в состоянии своевременно обработать все события.**

2.

**Поток** (thread) — это, сущность операционной системы, процесс выполнения на процессоре набора инструкций, точнее говоря программного кода. Общее назначение потоков — параллельное выполнение на процессоре двух или более различных задач.

1. **Реализация потоков на уровне ядра**. Проще говоря, это классическая 1:1 модель. Под эту категорию подпадают:
   * Потоки Win32.
   * Реализация Posix Threads в Linux — Native Posix Threads Library (NPTL). Дело в том, что до версии ядра 2.6 pthreads в Linux был целиком и полностью реализован в режиме пользователя (LinuxThreads). LinuxThreads реализовывалf модель 1:1 следующим образом: при создании нового потока, библиотека осуществляла системный вызов clone, и создавало новый процесс, который тем не менее разделял единое адресное пространство с родительским. Это породило множество проблем, к примеру потоки имели разные идентификаторы процесса, что противоречило некоторым аспектам стандарта Posix, которые касаются планировщика, сигналов, примитивов синхронизации. Также модель вытеснения потоков, работала во многих случаях с ошибками, по этому поддержку pthread решено было положить на плечи ядра. Сразу две разработки велись в данном направлении компаниями IBM и Red Hat. Однако, реализация IBM не снискала должной популярности, и не была включена ни в один из дистрибутивов, потому IBM приостановила дальнейшую разработку и поддержку библиотеки (NGPT). Позднее NPTL вошли в библиотеку glibc.
   * Легковесные ядерны потоки (Leight Weight Kernel Threads — LWKT), например в DragonFlyBSD. Отличие этих потоков, от других потоков режима ядра в том, что легковесные ядерные потоки могут вытеснять другие ядерные потоки . Об этом говорит сайт https://intellect.icu . В DragonFlyBSD существует множество ядерных потоков, например поток обслуживания аппаратных прерываний, поток обслуживания программных прерываний и т.д. Все они работают с фиксированным приоритетом, так вот LWKT могут вытеснять эти потоки (preempt). Конечно это уже более специфические вещи, про которые можно говорить бесконечно, но приведу еще два примера. В Windows все потоки ядра выполняются либо в контексте потока инициировавшего системный вызов/IO операцию, либо в контексте потока системного процесса system. В Mac OS X существует еще более интересная система. В ядре есть лишь понятие task, т.е. задачи. Все операции ядра выполняются в контексте kernel\_task. Обработка аппаратного прерывания, к примеру, происходит в контексте потока драйвера, который обслуживает данное прерывание.
2. **Реализация потоков в пользовательском режиме**. Так как, системный вызов и смена контекста — достаточно тяжелые операции, идея реализовать поддержку потоков в режиме пользователя витает в воздухе давно. Множество попыток было сделано, однако данная методика популярности не обрела:
   * GNU Portable Threads — реализация Posix Threads в пользовательском режиме. Основное преимущество — высокая портабельность данной библиотеки, проще говоря она может быть легко перенесена на другие ОС. Проблему вытиснения потоков в данной библиотеке решили очень просто — потоки в ней не вытесняются :) Ну и конечно ни о какой мультмпроцессорности речь идти не может. Данная библиотека реализует модель N:1.
   * Carbon Threads, которые я упоминал уже не раз, и RealBasic Threads.
3. **Гибридная реализация**. Попытка использовать все преимущества первого и второго подхода, но как правило подобные мутанты обладают гораздо бОльшими недостатками, нежели достоинствами. Один из примеров: реализация Posix Threads в NetBSD по модели N:M, которая была посже заменена на систему 1:1. Более подробно вы можете прочесть в публикации Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.

А теперь мы опишем эти способы со всеми их достоинствами

и недостатками.

**Первый способ** — это поместить весь набор потоков **в пользовательском пространстве.**

И об этом наборе ядру ничего не известно. Что касается ядра, оно управляет обычными,

однопотоковыми процессами. Первое и самое очевидное преимущество состоит в том,

что набор потоков на пользовательском уровне может быть реализован в операционной

системе, которая не поддерживает потоки. Под эту категорию подпадают все операци-

онные системы, даже те, которые еще находятся в разработке. При этом подходе потоки

реализованы с помощью библиотеки.

У всех этих реализаций одна и та же общая структура (рис. 2.10, а). Потоки запускают-

ся поверх системы поддержки исполнения программ (run-time system), которая пред-

ставляет собой набор процедур, управляющих потоками. Четыре из них: pthread\_create,

pthread\_exit, pthread\_join и pthread\_yield — мы уже рассмотрели, но обычно в наборе

есть и другие процедуры.

Когда потоки управляются в пользовательском пространстве, каждому процессу не-

обходимо иметь собственную таблицу потоков, чтобы отслеживать потоки, имеющиеся

в этом процессе. Эта таблица является аналогом таблицы процессов, имеющейся в ядре, за исключением того, что в ней содержатся лишь свойства, принадлежащие каждому

потоку, такие как счетчик команд потока, указатель стека, регистры, состояние и т. д.

Таблица потоков управляется системой поддержки исполнения программ. Когда поток

переводится в состояние готовности или блокируется, информация, необходимая для

возобновления его выполнения, сохраняется в таблице потоков, точно так же, как ядро

хранит информацию о процессах в таблице процессов.

Когда поток совершает какие-то действия, которые могут вызвать его локальную

блокировку, например ожидание, пока другой поток его процесса не завершит какую-

нибудь работу, он вызывает процедуру системы поддержки исполнения программ.

Эта процедура проверяет, может ли поток быть переведен в состояние блокировки.

Если может, она сохраняет регистры потока (то есть собственные регистры) в таблице

потоков, находит в таблице поток, готовый к выполнению, и перезагружает регистры

машины сохраненными значениями нового потока. Как только будут переключены

указатель стека и счетчик команд, автоматически возобновится выполнение ново-

го потока. Если машине дается инструкция сохранить все регистры и следующая

инструкция — загрузить все регистры, то полное переключение потока может быть

осуществлено за счет всего лишь нескольких инструкций. Переключение потоков,

осуществленное таким образом, по крайней мере на порядок, а может быть, и больше,

быстрее, чем перехват управления ядром, что является веским аргументом в пользу

набора потоков, реализуемого на пользовательском уровне.

Но у потоков есть одно основное отличие от процессов. Когда поток на время оста-

навливает свое выполнение, например когда он вызывает thread\_yield, код процедуры

thread\_yield может самостоятельно сохранять информацию о потоке в таблице пото-

ков. Более того, он может затем вызвать планировщик потоков, чтобы тот выбрал для

выполнения другой поток. Процедура, которая сохраняет состояние потока, и пла-

нировщик — это всего лишь локальные процедуры, поэтому их вызов намного более

эффективен, чем вызов ядра. Помимо всего прочего, не требуется перехват управленияния к отсутствующей странице и операционная система обращается к диску и получает

отсутствующие инструкции (и их соседей). Это называется ошибкой вызова отсутству-

ющей страницы. Процесс блокируется до тех пор, пока не будет найдена и считана не-

обходимая инструкция. Если ошибка обращения к отсутствующей странице возникает

при выполнении потока, ядро, которое даже не знает о существовании потоков, как

и следовало ожидать, блокирует весь процесс до тех пор, пока не завершится дисковая

операция ввода-вывода, даже если другие потоки будут готовы к выполнению.

Использование набора потоков, реализованного на пользовательском уровне, связано

еще с одной проблемой: если начинается выполнение одного из потоков, то никакой

другой поток, принадлежащий этому процессу, не сможет выполняться до тех пор,

пока первый поток добровольно не уступит центральный процессор. В рамках единого

процесса нет прерываний по таймеру, позволяющих планировать работу процессов по

круговому циклу (поочередно). Если поток не войдет в систему поддержки выполнения

программ по доброй воле, у планировщика не будет никаких шансов на работу.

Проблему бесконечного выполнения потоков можно решить также путем передачи

управления системе поддержки выполнения программ за счет запроса сигнала (преры-

вания) по таймеру с периодичностью один раз в секунду, но для программы это далеко не

самое лучшее решение. Возможность периодических и довольно частых прерываний по

таймеру предоставляется не всегда, но даже если она и предоставляется, общие издержки

могут быть весьма существенными. Более того, поток может также нуждаться в прерыва-

ниях по таймеру, мешая использовать таймер системе поддержки выполнения программ.

Другой наиболее сильный аргумент против потоков, реализованных на пользователь-

ском уровне, состоит в том, что программистам потоки обычно требуются именно в тех

приложениях, где они часто блокируются, как, к примеру, в многопоточном веб-сервере.

Эти потоки часто совершают системные вызовы. Как только для выполнения системно-

го вызова ядро осуществит перехват управления, ему не составит особого труда занять-

ся переключением потоков, если прежний поток заблокирован, а когда ядро займется

решением этой задачи, отпадет необходимость постоянного обращения к системному

вызову select, чтобы определить безопасность системного вызова read. Зачем вообще

использовать потоки в тех приложениях, которые, по существу, полностью завязаны на

скорость работы центрального процессора и редко используют блокировку? Никто не

станет всерьез предлагать использование потоков при вычислении первых n простых

чисел или при игре в шахматы, поскольку в данных случаях от них будет мало проку.

**Реализация потоков в ядре**

Теперь давайте рассмотрим, что произойдет, если ядро будет знать о потоках и управ-

лять ими. Как показано на рис. 2.10, б, здесь уже не нужна система поддержки ис-

полнения программ. Также здесь нет и таблицы процессов в каждом потоке. 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 путем обновления та-

блицы потоков в ядре.

В таблице потоков, находящейся в ядре, содержатся регистры каждого потока,

состояние и другая информация. Вся информация аналогична той, которая ис-

пользовалась для потоков, создаваемых на пользовательском уровне, но теперь она

содержится в ядре, а не в пространстве пользователя (внутри системы поддержкиисполнения программ). Эта информация является подмножеством той информа-

ции, которую поддерживают традиционные ядра в отношении своих однопоточных

процессов, то есть подмножеством состояния процесса. 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 системе поддержки исполнения

программ. Когда поток блокируется, ядро по своему выбору может запустить либо

другой поток из этого же самого процесса (если имеется готовый к выполнению по-

ток), либо поток из другого процесса. Когда потоки реализуются на пользовательском

уровне, система поддержки исполнения программ 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).

Поскольку создание и уничтожение потоков в ядре требует относительно более весо-

мых затрат, некоторые системы с учетом складывающейся ситуации применяют более

правильный подход и используют свои потоки повторно. При уничтожении потока он

помечается как неспособный к выполнению, но это не влияет на его структуру данных,

имеющуюся в ядре. Чуть позже, когда должен быть создан новый поток, вместо этого

повторно активируется старый поток, что приводит к экономии времени. Повторное

использование потоков допустимо и на пользовательском уровне, но для этого нет

достаточно веских оснований, поскольку издержки на управление потоками там зна-

чительно меньше.

Для потоков, реализованных на уровне ядра, не требуется никаких новых, неблокиру-

ющих системных вызовов. Более того, если один из выполняемых потоков столкнется

с ошибкой обращения к отсутствующей странице, ядро может с легкостью проверить

наличие у процесса любых других готовых к выполнению потоков и при наличии та-

ковых запустить один из них на выполнение, пока будет длиться ожидание извлечения

запрошенной страницы с диска. Главный недостаток этих потоков состоит в весьма

существенных затратах времени на системный вызов, поэтому, если операции над по-

токами (создание, удаление и т. п.) выполняются довольно часто, это влечет за собой

более существенные издержки.

Хотя потоки, создаваемые на уровне ядра, и позволяют решить ряд проблем, но спра-

виться со всеми существующими проблемами они не в состоянии. Что будет, к при-

меру, 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 запланиро-

вано следующим. Если он собирается вызвать команду exec, чтобы запустить новую

программу, то, наверное, правильным выбором будет наличие только одного потока.

Но если он продолжит выполнение, то лучше всего было бы, наверное, воспроизвести

все имеющиеся потоки.

Другой проблемой являются сигналы. Стоит вспомнить, что сигналы посылаются

процессам, а не потокам, по крайней мере, так делается в классической модели. Какой

из потоков должен обработать поступающий сигнал? Может быть, потоки должны за-

регистрировать свои интересы в конкретных сигналах, чтобы при поступлении сигнала

он передавался потоку, который заявил о своей заинтересованности в этом сигнале?

Тогда возникает вопрос: что будет, если на один и тот же сигнал зарегистрировались

два или более двух потоков? И это только две проблемы, создаваемые потоками, а ведь

на самом деле их значительно больше.

Гибридная реализация

В попытках объединить преимущества создания потоков на уровне пользователя и на

уровне ядра была исследована масса различных путей. Один из них (рис. 2.11) заклю-

чается в использовании потоков на уровне ядра, а затем нескольких потоков на уровне

пользователя в рамках некоторых или всех потоков на уровне ядра. При использова-

нии 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. У некоторых из этих потоков могут быть несколько потоков на пользова-

тельском уровне, которые расходятся от их вершины. Создание, удаление и планирова-

ние выполнения этих потоков осуществляется точно так же, как и у пользовательских

потоков, принадлежащих процессу, запущенному под управлением операционной

системы, не способной на многопоточную работу. В этой модели каждый поток на

уровне ядра обладает определенным набором потоков на уровне пользователя, которые

используют его по очереди.

1. **Прерывание** (hardware interrupt) – это событие, генерируемое внешним (по отношению к процессору ) устройством. Посредством аппаратных прерываний аппаратура либо информирует центральный процессор о том, что произошло какое-либо событие, требующее немедленной реакции (например, пользователь нажал клавишу), либо сообщает о завершении асинхронной операции ввода-вывода (например, закончено чтение данных с диска в основную память). Важный тип аппаратных прерываний – прерывания таймера, которые генерируются периодически через фиксированный промежуток времени. Прерывания таймера используются операционной системой при планировании процессов. Каждый тип аппаратных прерываний имеет собственный номер, однозначно определяющий источник прерывания. Аппаратное прерывание – это асинхронное событие, то есть оно возникает вне зависимости от того, какой код исполняется процессором в данный момент. Обработка аппаратного прерывания не должна учитывать, какой процесс является текущим.

**Типы прерываний**

Прерывания могут возникать вследствие внешних событий (*внешние прерывания*), генерироваться специальными инструкциями (*программные прерывания*), либо из-за ошибок, возникающих в процессе выполнения программы (*исключений*). Независимо от типа прерывания микропроцессор обрабатывает их одинаково.

**Внешние прерывания** возникают вне микропроцессора. Запросы прерываний поступают по специальным входным линиям микросхемы микропроцессора. Внешние прерывания инициируются аппаратурой. Они могут быть вызваны сигналом микросхемы таймера, сигналом от принтера, нажатием клавиши на клавиатуре и множеством других причин. Внешние прерывания не координируются с работой программного обеспечения. Различают *немаскируемые* и *маскируемые* внешние прерывания.

**Немаскируемые прерывания.** Микропроцессор извещается о возникновении немаскируемого внешнего прерывания с помощью сигнала, поступающего по его линии NMI (Non Mask Interrupt). Получив сигнал, микропроцессор после выполнения очередной инструкции производит переход по вектору 2, постоянно назначенному для немаскируемых прерываний.

Обычно немаскируемое прерывание используется для извещения о возникновении тех или иных неисправностей, делающих невозможным нормальное продолжение работы, например, при обнаружении сбоя оперативной памяти.

Когда происходит немаскируемое прерывание, микропроцессор блокирует обработку новых запросов немаскируемых прерываний до тех пор, пока не будет выполнена инструкция возврата из прерывания IRET. Таким образом, немаскируемые прерывания не могут быть «вложенными».

Немаскируемое прерывание не может быть *маскировано* (игнорировано) средствами самого микропроцессора. Однако обычно имеются внешние по отношению к микропроцессору схемы, позволяющие запретить немаскируемые прерывания путём блокировки прохождения сигнала по линии NMI.

**Маскируемые прерывания.** Сигнал о запросе маскируемого прерывания поступает в микропроцессор по линии INT. Выполнение текущей инструкции завершается обычным образом и в стеке сохраняется адрес следующей инструкции.

Маскируемые прерывания могут быть запрещены средствами самого микропроцессора – для этого в регистре флагов должен быть сброшен флаг прерываний IF. В таком случае микропроцессор игнорирует сигнал на линии INT и продолжает выполнять инструкции в обычном порядке. Однако когда этот флаг установлен и поступает запрос маскируемого прерывания, микропроцессор завершает выполнение текущей инструкции, после чего передаёт управление подпрограмме обработки прерывания.

Ассемблерная команда CLI сбрасывает флаг прерываний IF, а команда STI вновь его устанавливает. Вызовы этих команд должны быть сбалансированы, и всегда завершаться вызовом команды STI. В противном случае, система окажется недоступной для управления. Кроме того, не следует сбрасывать флаг прерываний IF на время, превышающее период следования импульсов от таймера-счетчика, т.к. иначе операции, зависящие от времени, будут выполняться некорректно.

В отличие от немаскируемых прерываний, маскируемым не назначается какой-либо фиксированный вектор прерывания. Вместо этого микропроцессор, начиная процедуру выполнения маскируемого прерывания, выдаёт сигнал INTA, получив который, устройство, запросившее прерывание, передает микропроцессору один байт информации, являющийся номером вектора прерывания, которое должно быть выполнено. Для обработки маскируемых прерываний можно использовать до 256 векторов.

Компания Intel зарезервировала векторы 0–31 для внутренних нужд микропроцессора. Поэтому настоятельно не рекомендуется использовать инструкции INT с номерами 0–31, а также назначать этим векторам маскируемые аппаратные прерывания.

**Программные прерывания**представляют обычные подпрограммы, которые вызываются прикладными программами для обработки нажатий клавиш на клавиатуре, событий от таймера, вывода сообщений на экран и других действий. Однако эти подпрограммы относятся не к прикладной программе пользователя, а к операционной системе.

Программные прерывания могут быть вложенными. Они инициируются командой INT с номером от 0 до 0FFh. Базовая система ввода-вывода BIOS и операционная система MS-DOS предоставляют множество функций в виде программных прерываний. Некоторые программные прерывания обеспечивают несколько функций. Например, большинство полезных для пользователя функций MS-DOS закреплено за прерыванием 21h. Номер конкретной функции должен быть указан в регистре AH, а подфункции – в регистре AL.

Аппаратное прерывание может получить управление при выполнении программного. При этом не возникает конфликтов, т.к. каждая подпрограмма обработки прерывания сохраняет значения всех используемых регистров и затем восстанавливает их при выходе. Схема обработки программного прерывания показана на рис. 41

Рис. 41. Схема обработки программного прерывания

**Прерывания по исключению**разделяются на несколько видов – исключения по ошибке деления, по точке остановка, по переполнению и по трассировке.

*Прерывание по ошибке деления* возникает при выполнении инструкции DIV или IDIV, если результат превышает максимально допустимую величину, а также при попытке деления на ноль. В результате управление передаётся обработчику прерывания по вектору 0.

*Прерывание по точке останова* возникает, если в программе встретилась инструкция INT 3. Выполнение инструкции приводит к передаче управления обработчику прерывания по вектору 3. Система команд микропроцессора 8086 обеспечивает два формата инструкции INT 3 – однобайтовый (код команды CC) и двухбайтовый (код команды CD 03). Функционально они не отличаются, но двухбайтовый формат принято относить к программным прерываниям, а однобайтовый – к исключениям.

*Прерывание по переполнению* происходит, если при выполнении инструкции INTO установлен флаг переполнения OF. Когда флаг сброшен, инструкция INTO не производит никаких действий. Обработчик прерывания по переполнению использует вектор 4.

*Прерывания по трассировке* происходят после выполнения каждой инструкции, когда в регистре флагов установлен флаг трассировки TF. В этом случае, завершив выполнение инструкции, микропроцессор передаёт управление обработчику прерывания по вектору 1.

Особенностью микропроцессоров семейства 8086 является то, что после установки флага трассировки связанное с ним прерывание происходит не после инструкции, в ходе выполнения которой этот флаг был установлен, а по завершении следующей за ней инструкции.

Например, если при завершении обработчика прерывания по трассировке с помощью команды IRET в восстановленном содержимом регистра флагов флаг TF окажется установлен, то новое прерывание по трассировке произойдёт не сразу после выполнения инструкции IRET, а по завершении команды, на которую инструкцией IRET будет возвращено управление.

1. **Механизм обработки прерываний**

Цифровые устройства, построенные на основе архитектуры IBM PC, используют механизм прерываний. *Прерывания* приостанавливают выполнение программы для выполнения специальных системных действий. Необходимость прерываний обусловлено двумя причинами: преднамеренный запрос на выполнение операций ввода-вывода на различные внешние устройства и обработка непредвиденных программных ошибок.

Обработка прерываний микропроцессором производится в три этапа:

* прерывание выполнения текущей программы;
* переход к выполнению подпрограммы обработки прерывания;
* возврат управления прерванной программе.

Любая программа, загруженная на выполнение, занимает в памяти отдельное от других программ адресное пространство. Регистры являются разделяемыми ресурсами для микропроцессора, поэтому их содержимое нужно сохранять при переключении задач. Закончив выполнение инструкции, в ходе которого возник запрос на прерывание, в стеке сохраняется содержимое регистра флагов и регистровой пары CS:IP.

В регистре флагов сбрасываются флаги разрешения прерывания IF и трассировки TF. Этим обеспечивается блокировка возникновения новых маскируемых внешних прерываний (см. раздел 2.2) и запрещаются пошаговые прерывания по выполнении каждой инструкции.

Затем в регистровую пару CS:IP загружается адрес *подпрограммы обработки прерывания ISR* (Interrupt Subroutine), которой и передается управление. Адреса подпрограмм прерываний называют *векторами*, а их совокупность – *таблицей векторов прерываний IVT* (Interrupt Vector Table).

Каждый вектор размещается в 4 байтах. Старшее слово адреса содержит значение сегмента CS, младшее – смещение IP. Первые 1024 байта оперативной памяти содержат векторы прерываний. Всего доступно 256 векторов. Вектор для прерывания 0 начинается с ячейки памяти 0000:0000, для прерывания 1 – с ячейки 0000:0004, 2 – с ячейки 0000:0008 и т.д. Каждый элемент таблицы указывает на подпрограмму обработки указанного типа прерывания. В табл. 91 описаны вектора таблицы прерываний.

После выполнения подпрограммы обработки прерывания следует восстановить состояния прерванной программы. Для этого всегда последней командой подпрограммы обработки прерывания указывается машинная команда IRET. По этой команде микропроцессор извлекает из стека последовательно три слова и помещает их, соответственно в регистры IP, CS и регистр флагов.

В результате после выполнения подпрограммы обработки прерываний регистровая пара CS:IP будет содержать адрес следующей команды прерванной программы, а регистр флагов – состояние флагов на момент передачи управления ISR.

1. **Системный вызов**

**Системный вызов** является механизмом, который обеспечивает интерфейс между процессом и операционной системой. Это программный метод, при котором компьютерная программа запрашивает сервис у ядра ОС.

Системный вызов предлагает услуги операционной системы пользовательским программам через API (интерфейс прикладного программирования). Системные вызовы являются единственными точками входа в систему ядра.

Типы системных вызовов

Вот пять типов системных вызовов, используемых в ОС:

* Контроль процесса
* Управление файлами
* Управление устройством
* Информационное обслуживание
* Связи

Системные вызовы в операционной системе

Пример системного вызова

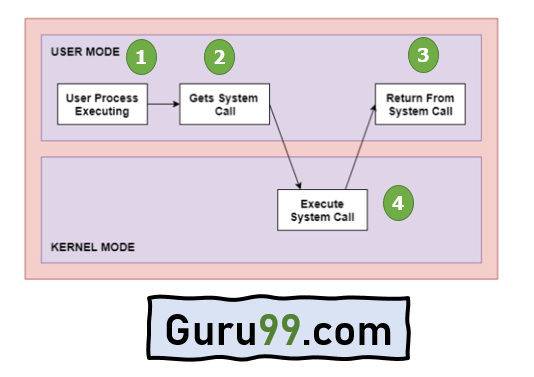
Например, если нам нужно написать программный код для чтения данных из одного файла, скопируйте эти данные в другой файл. Первой информацией, которая требуется программе, является имя двух файлов, входных и выходных файлов.

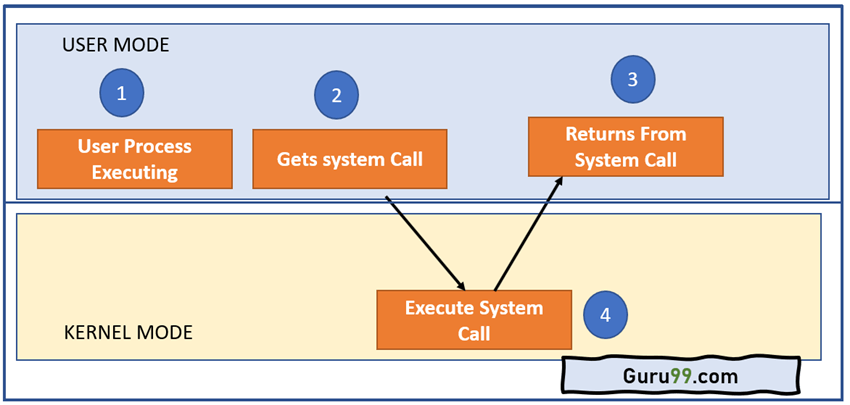
В интерактивной системе этот тип выполнения программы требует некоторых системных вызовов ОС.

* Первый звонок – написать подсказку на экране
* Во-вторых, чтобы прочитать с клавиатуры, символы, которые определяют два файла.

Как работает системный вызов?

Вот шаги для системного вызова:

[](https://coderlessons.com/wp-content/uploads/images/gur/d1e6243965a85b7cfb921cac42529b77.png)

Архитектура системного вызова

Как вы можете видеть на приведенной выше диаграмме.

**Шаг 1)** Процессы, выполняемые в пользовательском режиме до тех пор, пока системный вызов не прервет его.

**Шаг 2)** После этого системный вызов выполняется в режиме ядра в приоритетном порядке.

**Шаг 3)** По завершении выполнения системного вызова управление возвращается в режим пользователя.,

**Шаг 4)** Выполнение пользовательских процессов возобновляется в режиме ядра.

Зачем вам системные вызовы в ОС?

Ниже приведены ситуации, которые требуют системных вызовов в ОС:

* Чтение и запись из файлов требуют системных вызовов.
* Если файловая система хочет создать или удалить файлы, требуются системные вызовы.
* Системные вызовы используются для создания и управления новыми процессами.
* Сетевые подключения требуют системных вызовов для отправки и получения пакетов.
* Доступ к аппаратным устройствам, таким как сканер, принтер, требуется системный вызов.

Контроль процесса

Эти системные вызовы выполняют задачу создания процесса, завершения процесса и т. Д.

Функции:

* Конец и Прервать
* Загрузить и выполнить
* Создать процесс и завершить процесс
* Ожидание и подписанное событие
* Выделить и освободить память

Управление файлами

Системные вызовы управления файлами обрабатывают задания по обработке файлов, такие как создание файла, чтение, запись и т. Д.

Функции:

* Создать файл
* Удалить файл
* Открыть и закрыть файл
* Читать, писать и перемещать
* Получить и установить атрибуты файла

Управление устройством

Управление устройствами выполняет работу с устройствами, такими как чтение из буферов устройств, запись в буферы устройств и т. Д.

функции

* Запрос и релиз устройства
* Логически подключать / отключать устройства
* Получить и установить атрибуты устройства

Информационное обслуживание

Он обрабатывает информацию и ее передачу между ОС и программой пользователя.

Функции:

* Получить или установить время и дату
* Получить атрибуты процесса и устройства

Связь:

Эти типы системных вызовов специально используются для межпроцессного взаимодействия.

Функции:

* Создание, удаление коммуникационных соединений
* Отправить, получить сообщение
* Помогите ОС перенести информацию о статусе
* Подключите или отсоедините удаленные устройства

Важные системные вызовы, используемые в ОС

Подождите()

В некоторых системах процесс должен ждать, пока другой процесс завершит свое выполнение. Этот тип ситуации возникает, когда родительский процесс создает дочерний процесс, и выполнение родительского процесса остается приостановленным, пока не выполнится его дочерний процесс.

Приостановка родительского процесса автоматически происходит с помощью системного вызова wait (). Когда дочерний процесс завершает выполнение, элемент управления возвращается к родительскому процессу.

вилка ()

Процессы используют этот системный вызов для создания процессов, которые являются их копиями. С помощью этого системного вызова родительский процесс создает дочерний процесс, и выполнение родительского процесса будет приостановлено до выполнения дочернего процесса.

Exec ()

Этот системный вызов выполняется, когда исполняемый файл в контексте уже запущенного процесса заменяет старый исполняемый файл. Однако исходный идентификатор процесса остается, поскольку новый процесс не создается, но стек, данные, заголовок, данные и т. Д. Заменяются новым процессом.

убийство():

Системный вызов kill () используется ОС для отправки сигнала завершения процессу, который побуждает процесс завершиться. Однако системный вызов kill не обязательно означает уничтожение процесса и может иметь различные значения.

Выход():

Системный вызов exit () используется для прекращения выполнения программы. Этот вызов, особенно в многопоточной среде, определяет, что выполнение потока завершено. ОС восстанавливает ресурсы, которые использовались процессом после использования системного вызова exit ().

**6.межпроцессорное взаимодействие 7. Средства межпроцессного взаимодействия.**

**Межпроцессное взаимодействие** ([англ.](https://ru.m.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) ***i****nter-****p****rocess****c****ommunication*, **IPC**) — обмен данными между [потоками](https://ru.m.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F) одного или разных [процессов](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Реализуется посредством механизмов, предоставляемых [ядром ОС](https://ru.m.wikipedia.org/wiki/%D0%AF%D0%B4%D1%80%D0%BE_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) или [процессом](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), использующим механизмы [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) и реализующим новые возможности IPC. Может осуществляться как на одном компьютере, так и между несколькими компьютерами [сети](https://ru.m.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C).

Из механизмов, предоставляемых ОС и используемых для IPC, можно выделить:

* механизмы [обмена сообщениями](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BC%D0%B5%D0%BD_%D1%81%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D1%8F%D0%BC%D0%B8);
* механизмы [синхронизации](https://ru.m.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%85%D1%80%D0%BE%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0));
* механизмы [разделения памяти](https://ru.m.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B5%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C);
* механизмы [удалённых вызовов](https://ru.m.wikipedia.org/wiki/Remote_Procedure_Call) (RPC).

Межпроцессное взаимодействие, наряду с механизмами [адресации памяти](https://ru.m.wikipedia.org/wiki/%D0%90%D0%B4%D1%80%D0%B5%D1%81%D0%B0%D1%86%D0%B8%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8), является основой для разграничения [адресного пространства](https://ru.m.wikipedia.org/wiki/%D0%90%D0%B4%D1%80%D0%B5%D1%81%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE) между [процессами](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))[[1]](https://ru.m.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5#cite_note-1).

## таблицаметодов межпроцессного взаимодействия

|  |  |
| --- | --- |
| **Метод** | **Реализуется**[**ОС**](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0)**или**[**процессом**](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) |
| [Файл](https://ru.m.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB) | Все [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). |
| [Сигнал](https://ru.m.wikipedia.org/wiki/%D0%A1%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB%D1%8B_(UNIX)) | Большинство [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0); в некоторых [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), например, в [Windows](https://ru.m.wikipedia.org/wiki/Windows), сигналы доступны только в библиотеках, реализующих [стандартную библиотеку языка Си](https://ru.m.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_%D0%A1%D0%B8), и не могут использоваться для IPC. |
| [Сокет](https://ru.m.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BA%D0%B5%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81)) | Большинство [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). |
| [Канал](https://ru.m.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B2%D0%B5%D0%B9%D0%B5%D1%80_(UNIX)) | Все [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.m.wikipedia.org/wiki/POSIX). |
| [Именованный канал](https://ru.m.wikipedia.org/wiki/%D0%98%D0%BC%D0%B5%D0%BD%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%B0%D0%BD%D0%B0%D0%BB) | Все [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.m.wikipedia.org/wiki/POSIX). |
| [Неименованный канал](https://ru.m.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B8%D0%BC%D0%B5%D0%BD%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%B0%D0%BD%D0%B0%D0%BB) | Все [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.m.wikipedia.org/wiki/POSIX). |
| [Семафор](https://ru.m.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) | Все [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.m.wikipedia.org/wiki/POSIX). |
| [Разделяемая память](https://ru.m.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B5%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) | Все [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.m.wikipedia.org/wiki/POSIX). |
| [Обмен сообщениями](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BC%D0%B5%D0%BD_%D1%81%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D1%8F%D0%BC%D0%B8) (без разделения) | Используется в парадигме [MPI](https://ru.m.wikipedia.org/wiki/Message_Passing_Interface), [Java](https://ru.m.wikipedia.org/wiki/Java) [RMI](https://ru.m.wikipedia.org/wiki/RMI), [CORBA](https://ru.m.wikipedia.org/wiki/CORBA) и других. |
| [Проецируемый в память файл](https://ru.m.wikipedia.org/wiki/Mmap) (mmap) | Все [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.m.wikipedia.org/wiki/POSIX). При использовании [временного файла](https://ru.m.wikipedia.org/wiki/%D0%92%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) возможно возникновение [гонки](https://ru.m.wikipedia.org/wiki/%D0%A1%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5_%D0%B3%D0%BE%D0%BD%D0%BA%D0%B8). [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) [Windows](https://ru.m.wikipedia.org/wiki/Windows) также предоставляет этот механизм, но посредством [API](https://ru.m.wikipedia.org/wiki/API), отличающегося от [API](https://ru.m.wikipedia.org/wiki/API), описанного в стандарте [POSIX](https://ru.m.wikipedia.org/wiki/POSIX). |
| [Очередь сообщений](https://ru.m.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_%D1%81%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D0%B9) (Message queue) | Большинство [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). |
| [Почтовый ящик](https://ru.m.wikipedia.org/wiki/Mailslot) | Некоторые [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0).  Расшифровка таблицы - средства |

**Файл** - именованная область данных на [носителе информации](https://ru.m.wikipedia.org/wiki/%D0%9D%D0%BE%D1%81%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D0%B8), используемая как базовый объект взаимодействия с данными в [операционных системах](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0).

**Сигнал** в операционных системах [семейства Unix](https://ru.m.wikipedia.org/wiki/Unix-%D0%BF%D0%BE%D0%B4%D0%BE%D0%B1%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) — асинхронное уведомление процесса о каком-либо событии, один из основных способов [взаимодействия между процессами](https://ru.m.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5). Когда сигнал послан процессу, операционная система прерывает выполнение процесса, при этом, если процесс установил собственный *обработчик сигнала*, операционная система запускает этот обработчик, передав ему информацию о сигнале, если процесс не установил обработчика, то выполняется обработчик по умолчанию.

**Со́кет** ([англ.](https://ru.m.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *socket* — разъём) — название [программного интерфейса](https://ru.m.wikipedia.org/wiki/API) для обеспечения обмена данными между [процессами](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Процессы при таком обмене могут исполняться как на одной [ЭВМ](https://ru.m.wikipedia.org/wiki/%D0%AD%D0%92%D0%9C), так и на различных ЭВМ, связанных между собой [сетью](https://ru.m.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C). Сокет — [абстрактный](https://ru.m.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%86%D0%B8%D1%8F) объект, представляющий конечную точку соединения

**Конвейер или канал** ([англ.](https://ru.m.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *pipeline*) в терминологии [операционных систем](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) семейства [Unix](https://ru.m.wikipedia.org/wiki/Unix) — некоторое множество [процессов](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), для которых выполнено следующее [перенаправление ввода-вывода](https://ru.m.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BD%D0%B0%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5)): то, что выводит на поток стандартного вывода предыдущий процесс, попадает в поток стандартного ввода следующего процесса. Запуск конвейера реализован с помощью системного вызова pipe().

В [программировании](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) **именованный канал** или **именованный конвейер** ([англ.](https://ru.m.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *named pipe*) — один из методов [межпроцессного взаимодействия](https://ru.m.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5" \o "Межпроцессное взаимодействие), расширение понятия [конвейера](https://ru.m.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B2%D0%B5%D0%B9%D0%B5%D1%80_(UNIX)) в [Unix](https://ru.m.wikipedia.org/wiki/Unix) и подобных [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). Именованный канал позволяет различным процессам обмениваться данными, даже если программы, выполняющиеся в этих процессах, изначально не были написаны для взаимодействия с другими программами. Это понятие также существует и в [Microsoft Windows](https://ru.m.wikipedia.org/wiki/Microsoft_Windows), хотя там его [семантика](https://ru.m.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D0%BD%D1%82%D0%B8%D0%BA%D0%B0) существенно отличается. Традиционный канал — «безымянен», потому что существует анонимно и только во время выполнения процесса. Именованный канал — существует в [системе](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) и после завершения процесса. Он должен быть «отсоединён» или удалён, когда уже не используется.

**Неименованный канал** ([англ.](https://ru.m.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *anonymous pipe*) — один из методов [межпроцессного взаимодействия](https://ru.m.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5" \o "Межпроцессное взаимодействие) ([IPC](https://ru.m.wikipedia.org/wiki/Inter-process_communication)) в [операционной системе](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), который доступен связанным процессам — родительскому и дочернему. Представляется в виде области памяти на внешнем запоминающем устройстве, управляемой операционной системой, которая осуществляет выделение взаимодействующим процессам частей из этой области памяти для совместной работы. Организация данных в канале использует стратегию [FIFO](https://ru.m.wikipedia.org/wiki/FIFO), то есть информация, которая первой записана в канал, будет первой прочитана из канала.

**Семафо́р** ([англ.](https://ru.m.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *semaphore*) — примитив синхронизации[[1]](https://ru.m.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#cite_note-:8-1) работы [процессов](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) и [потоков](https://ru.m.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F), в основе которого лежит счётчик, над которым можно производить две [атомарные операции](https://ru.m.wikipedia.org/wiki/%D0%90%D1%82%D0%BE%D0%BC%D0%B0%D1%80%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F): увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счётчика является блокирующейся[[2]](https://ru.m.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#cite_note-2). Служит для построения более сложных механизмов синхронизации[[1]](https://ru.m.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)#cite_note-:8-1) и используется для синхронизации параллельно работающих задач, для защиты передачи данных через [разделяемую память](https://ru.m.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B5%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), для защиты [критических секций](https://ru.m.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B5%D0%BA%D1%86%D0%B8%D1%8F), а также для управления доступом к аппаратному обеспечению.

**Разделяемая память** ([англ.](https://ru.m.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Shared memory*) является самым быстрым средством обмена данными между [процессами](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))[[1]](https://ru.m.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B5%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C#cite_note-1).

[](https://ru.m.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B5%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C_(Shared_Memory).png)

Наглядное представление разделяемой памяти

В других средствах межпроцессового взаимодействия ([IPC](https://ru.m.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5)) обмен информацией между процессами проходит через [ядро](https://ru.m.wikipedia.org/wiki/%D0%AF%D0%B4%D1%80%D0%BE_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B), что приводит к [переключению контекста](https://ru.m.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82%D0%B0) между процессом и ядром, т.е. к потерям производительности[[2]](https://ru.m.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B5%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C#cite_note-2).

**Обмен сообщениями** в информатике — один из подходов реализации взаимодействия компонентов и систем, используемый в [параллельных вычислениях](https://ru.m.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B), [объектно-ориентированном программировании](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), также — одна из форм [межпроцессного взаимодействия](https://ru.m.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5" \o "Межпроцессное взаимодействие) в [операционных системах](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), в [микроядерных](https://ru.m.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D1%8F%D0%B4%D1%80%D0%BE" \o "Микроядро) операционных системах подход используется для обмена информацией между одним из ядер и одним или более исполняющих блоков.

Распределённые системы доступа к объектам и удалённого вызова методов, вида [ONC RPC](https://ru.m.wikipedia.org/wiki/ONC_RPC), [CORBA](https://ru.m.wikipedia.org/wiki/CORBA), [RMI](https://ru.m.wikipedia.org/wiki/RMI), [DCOM](https://ru.m.wikipedia.org/wiki/DCOM), [SOAP](https://ru.m.wikipedia.org/wiki/SOAP), [.Net\_Remoting](https://ru.m.wikipedia.org/wiki/.Net_Remoting), [QNX Neutrino RTOS](https://ru.m.wikipedia.org/wiki/QNX), [OpenBinder](https://ru.m.wikipedia.org/w/index.php?title=OpenBinder&action=edit&redlink=1" \o "OpenBinder (страница отсутствует)), [D-Bus](https://ru.m.wikipedia.org/wiki/D-Bus) и им подобные являются системами обмена сообщениями.

**ПРОЕЦИРУЕМЫЙ В ПАМЯТЬ ФАЙЛ mmap** — [POSIX](https://ru.m.wikipedia.org/wiki/POSIX)-совместимый [системный вызов](https://ru.m.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2) [Unix](https://ru.m.wikipedia.org/wiki/Unix), позволяющий выполнить [отображение файла или устройства на память](https://ru.m.wikipedia.org/wiki/%D0%9E%D1%82%D0%BE%D0%B1%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_%D1%84%D0%B0%D0%B9%D0%BB%D0%B0_%D0%BD%D0%B0_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C). Является методом ввода-вывода через [отображение файла на память](https://ru.m.wikipedia.org/wiki/%D0%9E%D1%82%D0%BE%D0%B1%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_%D1%84%D0%B0%D0%B9%D0%BB%D0%B0_%D0%BD%D0%B0_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) и естественным образом реализует выделение страниц по запросу, поскольку изначально содержимое файла не читается с диска и не использует физическую память вообще. Реальное считывание с диска производится в «ленивом» режиме, то есть при осуществлении доступа к определённому месту.

В [Linux](https://ru.m.wikipedia.org/wiki/Linux), [Mac OS X](https://ru.m.wikipedia.org/wiki/Mac_OS_X) и [BSD](https://ru.m.wikipedia.org/wiki/BSD) mmap может создавать несколько типов отображений.

**Очередь сообщений** (или **почтовый ящик**) — в [информатике](https://ru.m.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) — программно-инженерный [компонент](https://ru.m.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), используемый для [межпроцессного](https://ru.m.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5" \o "Межпроцессное взаимодействие) или [межпотокового](https://ru.m.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F" \o "Поток выполнения) взаимодействия внутри одного [процесса](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Для [обмена сообщениями](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BC%D0%B5%D0%BD_%D1%81%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D1%8F%D0%BC%D0%B8) используется [очередь](https://ru.m.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

**Мэйлслот** — один из механизмов [межпроцессного взаимодействия](https://ru.m.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5" \o "Межпроцессное взаимодействие), обеспечивающий однонаправленную передачу информации и позволяющий производить [широковещательную рассылку](https://ru.m.wikipedia.org/wiki/%D0%A8%D0%B8%D1%80%D0%BE%D0%BA%D0%BE%D0%B2%D0%B5%D1%89%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BA%D0%B0%D0%BD%D0%B0%D0%BB) сообщений по [сети](https://ru.m.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C).

**8.Логическая организация взаимодействия процессов;**

Различают два способа адресации: *прямую* и *непрямую*. В случае *прямой адресации взаимодействующие процессы* непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, указывают имена своих партнеров по взаимодействию, то такая схема адресации называется ***симметричной прямой адресацией***. **Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные.**сли только один из *взаимодействующих процессов*, например передающий, указывает имя своего партнера по *кооперации*, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например ожидает получения информации от произвольного источника, то такая схема адресации называется ***асимметричной прямой адресацией***.При***непрямой адресации*** данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных, имеющий свой адрес, откуда они могут быть затем изъяты каким-либо другим процессом.

**9, 10 сигнальные и канальные Средства связи**

**Категории средств обмена информацией**

Процессы могут взаимодействовать друг с другом, только обмениваясь информацией. По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории.

* Сигнальные. Передается минимальное количество информации – один бит, "да" или "нет". Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом. Неправильная реакция на сигнал или его игнорирование могут привести к трагическим последствиям. Вспомним профессора Плейшнера из кинофильма "Семнадцать мгновений весны". Сигнал тревоги – цветочный горшок на подоконнике – был ему передан, но профессор проигнорировал его. И к чему это привело?
* Канальные. "Общение" процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.
* Разделяемая память. Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием разделяемой памяти занимается операционная система (если, конечно, ее об этом попросят). "Общение" процессов напоминает совместное проживание студентов в одной комнате общежития. Возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, но требует повышенной осторожности (если вы переложили на другое место вещи вашего соседа по комнате, а часть из них еще и выбросили). Использование разделяемой памяти для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как сигнальным и канальным средствам коммуникации для этого необходимы специальные системные вызовы. Разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

Взаимодействуя друг с другом процессы обмениваются информацией. По объему передаваемой информации и степени возможного воздействия на поведение другого процесса средства обмена делятся на три вида.

* Сигнальные. Данный вид обмена передает минимальное количество информации – один бит, "да" или "нет". Используется, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Процесс, получивший сигнал должен знать, что означает полученный сигнал и как на него реагировать. Неправильная реакция на сигнал или его игнорирование могут привести к непредсказуемым последствиям.
* Канальные. Операционная система предоставляет линии связи, через которые происходит обмен информации между процессами. Объем передаваемой информации в единицу времени ограничен пропускной способностью линии связи.

**Сигнал** в операционных системах [семейства Unix](https://ru.m.wikipedia.org/wiki/Unix-%D0%BF%D0%BE%D0%B4%D0%BE%D0%B1%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) — асинхронное уведомление процесса о каком-либо событии, один из основных способов [взаимодействия между процессами](https://ru.m.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BD%D0%BE%D0%B5_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BE%D0%B4%D0%B5%D0%B9%D1%81%D1%82%D0%B2%D0%B8%D0%B5). Когда сигнал послан процессу, операционная система прерывает выполнение процесса, при этом, если процесс установил собственный *обработчик сигнала*, операционная система запускает этот обработчик, передав ему информацию о сигнале, если процесс не установил обработчика, то выполняется обработчик по умолчанию.

**Конвейер или канал** ([англ.](https://ru.m.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *pipeline*) в терминологии [операционных систем](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) семейства [Unix](https://ru.m.wikipedia.org/wiki/Unix) — некоторое множество [процессов](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), для которых выполнено следующее [перенаправление ввода-вывода](https://ru.m.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BD%D0%B0%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5)): то, что выводит на поток стандартного вывода предыдущий процесс, попадает в поток стандартного ввода следующего процесса. Запуск конвейера реализован с помощью системного вызова pipe()

11

**+Алгоритмы синхронизации. Взаимоисключение (не нашла). Условие Бернстайна. Критическая секция**

**Аппаратная поддержка взаимоисключений**

Наличие аппаратной поддержки *взаимоисключений* позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Мы уже обращались к общепринятому hardware для решения задачи реализации *взаимоисключений*, когда говорили об использовании механизма запрета/разрешения прерываний.

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как *атомарные операции*. Давайте обсудим, как концепции таких команд могут использоваться для реализации *взаимоисключений*.

**Критическая секция**

Важным понятием при изучении способов синхронизации процессов является понятие критической секции (critical section) программы. Критическая секция – это часть программы, исполнение которой может привести к возникновению race condition для определенного набора программ. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций программ. Реализация взаимоисключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам, участвующим во взаимодействии, критическая секция начинает выполняться как атомарная операция. Давайте рассмотрим следующий пример, в котором псевдопараллельные взаимодействующие процессы представлены действиями различных студентов (таблица 5.1):

Здесь критический участок для каждого процесса – от операции "Обнаруживает, что хлеба нет" до операции "Возвращается в комнату" включительно. В результате отсутствия взаимоисключения мы из ситуации "Нет хлеба" попадаем в ситуацию "Слишком много хлеба". Если бы этот критический участок выполнялся как атомарная операция – "Достает два батона хлеба", то проблема образования излишков была бы снята.

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 5.1. | | | |
| **Время** | **Студент 1** | **Студент 2** | **Студент 3** |
| 17-05 | Приходит в комнату |  |  |
| 17-07 | Обнаруживает,что хлеба нет |  |  |
| 17-09 | Уходит в магазин |  |  |
| 17-11 |  | Приходит в комнату |  |
| 17-13 |  | Обнаруживает, что хлеба нет |  |
| 17-15 |  | Уходит в магазин |  |
| 17-17 |  |  | Приходит в комнату |
| 17-19 |  |  | Обнаруживает,что хлеба нет |
| 17-21 |  |  | Уходит в магазин |
| 17-23 | Приходит в магазин |  |  |
| 17-25 | Покупает 2 батона на всех |  |  |
| 17-27 | Уходит из магазина |  |  |
| 17-29 |  | Приходит в магазин |  |
| 17-31 |  | Покупает 2 батона на всех |  |
| 17-33 |  | Уходит из магазина |  |
| 17-35 |  |  | Приходит в магазин |
| 17-37 |  |  | Покупает 2 батона на всех |
| 17-39 |  |  | Уходит из магазина |
| 17-41 | Возвращается в комнату |  |  |
| 17-43 |  |  |  |
| 17-45 |  |  |  |
| 17-47 |  | Возвращается в комнату |  |
| 17-49 |  |  |  |
| 17-51 |  |  |  |
| 17-53 |  |  | Возвращается в комнату |

Сделать процесс добывания хлеба атомарной операцией можно было бы следующим образом: перед началом этого процесса закрыть дверь изнутри на засов и уходить добывать хлеб через окно, а по окончании процесса вернуться в комнату через окно и отодвинуть засов. Тогда пока один студент добывает хлеб, все остальные находятся в состоянии ожидания под дверью (таблица 5.2).

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 5.2. | | | |
| **Время** | **Студент 1** | **Студент 2** | **Студент 3** |
| 17-05 | Приходит в комнату |  |  |
| 17-07 | Достает два батона хлеба |  |  |
| 17-43 |  | Приходит в комнату |  |
| 17-47 |  |  | Приходит в комнату |

Итак, для решения задачи необходимо, чтобы в том случае, когда процесс находится в своем критическом участке, другие процессы не могли войти в свои критические участки. Мы видим, что критический участок должен сопровождаться прологом ( entry section ) – "закрыть дверь изнутри на засов" – и эпилогом ( exit section ) – "отодвинуть засов", которые не имеют отношения к активности одиночного процесса. Во время выполнения пролога процесс должен, в частности, получить разрешение на вход в критический участок, а во время выполнения эпилога – сообщить другим процессам, что он покинул критическую секцию.

В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

while (some condition) {

entry section

critical section

exit section

remainder section

}

Здесь под remainder section понимаются все атомарные операции, не входящие в критическую секцию.

Оставшаяся часть этой лекции посвящена различным способам программной организации пролога и эпилога критического участка в случае, когда очередность доступа к критическому участку не имеет значения.

**Достаточные условия Бернстайна**

Можно ли до получения результатов определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна. Изложим их применительно к программам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных – это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы R(P) (R от слова read) суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова write) суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

P: x=u+v

y=x\*w

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменная *x* присутствует как в R(P), так и в W(P).

Теперь сформулируем **условия Бернстайна.**

Если для двух данных активностей P и Q:

* пересечение W(P) и W(Q) пусто,
* пересечение W(P) с R(Q) пусто,
* пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q *детерминировано*.

Если эти условия *не соблюдены*, возможно, параллельное выполнение P и Q детерминировано, а может быть, и нет.

Случай двух активностей естественным образом обобщается на их большее количество.

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, *исключает*для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется **взаимоисключением** (mutual exclusion)

**12.Требования, предъявляемые к алгоритмам**

Организация взаимоисключения для критических участков, конечно, позволит избежать возникновения race condition, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов. Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих критические участки, если они могут проходить их в произвольном порядке.

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test ) являются атомарными операциями.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс Pi исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях. Это условие получило название условия взаимоисключения (mutual exclusion).
4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress) .
5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (bound waiting) .

**13.Запрет прерываний. Переменная замок. Строгое чередование. Флаги и готовности. Алгоритм Петерсона. Команда Test-and-Set. Команда Swap**

**Запрет прерываний**. Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

while (some condition) {

запретить все прерывания

critical section

разрешить все прерывания

remainder section

}

Поскольку выход процесса из состояния **исполнение** без его завершения осуществляется по прерыванию, внутри критической секции никто не может вмешаться в его работу. Однако такое решение может иметь далеко идущие последствия, поскольку позволяет процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе. Допустим, что пользователь случайно или по злому умыслу запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в такой ситуации не обойтись.

Тем не менее запрет и разрешение прерываний часто применяются как пролог и эпилог к критическим секциям внутри самой операционной системы, например при обновлении содержимого PCB.

**Переменная-замок**

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из критической секции процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе " Критическая секция ").

shared int lock = 0;

/\* shared означает, что \*/

/\* переменная является разделяемой \*/

while (some condition) {

while(lock); lock = 1;

critical section

lock = 0;

remainder section

}

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию взаимоисключения, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

**Строгое чередование**

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

shared int turn = 0;

while (some condition) {

while(turn != i);

critical section

turn = 1-i;

remainder section

}

Очевидно, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение turn равно 1, и процесс P0 готов войти в критический участок, он не может сделать этого, даже если процесс P1 находится в **remainder section**.

**Флаги готовности**

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в критический участок

shared int ready[2] = {0, 0};

Когда i-й процесс готов войти в критическую секцию, он присваивает элементу массива ready[i] значение равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

while (some condition) {

ready[i] = 1;

while(ready[1-i]);

critical section

ready[i] = 0;

remainder section

}

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (deadlock). (Подробнее о тупиковых ситуациях рассказывается в лекции 7.)

**Алгоритм Петерсона**

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

shared int ready[2] = {0, 0};

shared int turn;

while (some condition) {

ready[i] = 1;

turn =1-i;

while(ready[1-i] && turn == 1-i);

critical section

ready[i] = 0;

remainder section

}

При исполнении пролога *критической секции* процесс Pi заявляет о своей готовности выполнить *критический участок* и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в *критическом участке* продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия *взаимоисключения* методом от противного. Пусть оба процесса одновременно оказались внутри своих *критических секций*. Заметим, что процесс Pi может войти в *критическую секцию*, только если ready[1-i] == 0 или turn == i. Заметим также, что если оба процесса выполняют свои *критические секции* одновременно, то значения флагов готовности для обоих процессов совпадают и равны 1. Могли ли оба процесса войти в *критические секции* из состояния, когда они оба одновременно находились в процессе выполнения цикла while? Нет, так как в этом случае переменная turn должна была бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных измениться не могут). Пусть процесс P0 первым вошел в *критический участок*, тогда процесс P1 должен был выполнить перед вхождением в цикл while по крайней мере один предваряющий оператор (turn = 0;). Однако после этого он не может выйти из цикла до окончания *критического участка* процесса P0, так как при входе в цикл ready[0] == 1 и turn == 0, и эти значения не могут измениться до тех пор, пока процесс P0 не покинет свой *критический участок*. Мы пришли к противоречию. Следовательно, имеет место *взаимоисключение*.

Докажем выполнение *условия прогресса*. Возьмем, без ограничения общности, процесс P0. Заметим, что он не может войти в свою *критическую секцию* только при совместном выполнении условий ready[1] == 1 и turn == 1. Если процесс P1 не готов к выполнению *критического участка*, то ready[1] == 0, и процесс P0 может осуществить вход. Если процесс P1 готов к выполнению *критического участка*, то ready[1] == 1 и переменная turn имеет значение 0 либо 1, позволяя процессу P0 либо процессу P1 начать выполнение *критической секции*. Если процесс P1 завершил выполнение *критического участка*, то он сбросит свой флаг готовности ready[1] == 0, разрешая процессу P0 приступить к выполнению критической работы. Таким образом, *условие прогресса* выполняется.

Отсюда же вытекает выполнение *условия ограниченного ожидания*. Так как в процессе ожидания разрешения на вход процесс P0 не изменяет значения переменных, он сможет начать исполнение своего *критического участка* после не более чем одного прохода по *критической секции* процесса P1.

**Команда Test-and-Set (проверить и присвоить 1)**

О выполнении команды *Test-and-Set* , осуществляющей проверку значения логической переменной с одновременной установкой ее значения в 1, можно думать как о выполнении функции

int Test\_and\_Set (int \*target){

int tmp = \*target;

\*target = 1;

return tmp;

}

С использованием этой атомарной команды мы можем модифицировать наш алгоритм для переменной-замка, так чтобы он обеспечивал *взаимоисключения*

shared int lock = 0;

while (some condition) {

while(Test\_and\_Set(&lock));

critical section

lock = 0;

remainder section

}

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет *условию ограниченного ожидания* для алгоритмов. Подумайте, как его следует изменить для соблюдения всех условий.

**Команда Swap (обменять значения)**

Выполнение команды *Swap* , обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией:

void Swap (int \*a, int \*b){

int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

Применяя атомарную команду *Swap* , мы можем реализовать предыдущий алгоритм, введя дополнительную логическую переменную key, локальную для каждого процесса:

shared int lock = 0;

int key;

while (some condition) {

key = 1;

do Swap(&lock,&key);

while (key);

critical section

lock = 0;

remainder section

}

**14.Механизмы синхронизации процессов.**

Концепция семафоров;

**Механизмы синхронизации: семафоры, мониторы,сообщения и мьютексы**мониторы

Для повышения производительности вычислительных систем и облегчения задачи программистов существуют специальные механизмы синхронизации. Одним из первых *механизмов*, предложенных для синхронизации поведения процессов, стали семафоры.**Семафор** представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать). Подобные переменные-семафоры могут с успехом применяться для решения различных задач организации взаимодействия процессов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

**Семафоры**

Семафоры – примитивы синхронизации предложены Дейкстрой (Dijkstra) в 1968 г. в качестве компонента операционной системы THE

•Семафор - это целочисленная неотрицательная переменная, для которой определены 2 операции: **P** (от датского слова proberen – проверять) и **V** (от verhogen – увеличивать).

•Операции **P** и **V** выполняются **атомарно**

Классическое определение этих операций выглядит следующим образом:

P(S): пока S == 0 процесс блокируется;

S = S - 1;

V(S): S = S + 1;

Эта запись означает следующее: при выполнении операции P над семафором S сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над семафором S к его значению просто прибавляется 1.

Виды семафоров **Двоичный семафор**

•S может принимать значения 0 и 1, инициализируется значением 1

•обеспечивает эксклюзивный доступ к ресурсу (например, при работе в критической секции)

•одновременно может выполняться только один поток

**Счетный семафор**

•S инициализируется значением N (число доступных единиц ресурса)

•представляет ресурсы, состоящие из нескольких однородных элементов

•позволяет потокам исполняться, пока есть неиспользуемые элементы

Одной из типовых задач, требующих организации взаимодействия процессов, является задача ***producer-consumer*** (производитель-потребитель). Пусть два процесса обмениваются информацией через буфер ограниченного размера. Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда.

**Итоги**

1.С помощью семафоров можно решить любую классическую задачу синхронизации, но по сути, семафоры - это разделяемые глобальные переменные (что является признаком плохой структуры программы)

2.Семафоры используются как для решения задачи взаимного исключения, так и для координации действий

3.Отсутствует связь между семафором и данными, доступом к которым он управляет

4.Нет контроля использования семафоров со стороны компилятора или операционной системы, соответственно - нет гарантий, что семафоры будут использованы правильно

**Проблема производителя и потребителя**

Проблема производителя-потребителя (англ .: Producer-consumer problem), также известная как проблема ограниченного буфера (англ.: Bounded-buffer problem), является классическим случаем проблем многопоточной синхронизации. Эта проблема описывает, что происходит, когда два потока, совместно использующие буфер фиксированного размера - так называемые «производитель» и «потребитель», фактически работают. Основная роль производителя - сгенерировать определенный объем данных в буфер, а затем повторить процесс. В то же время потребители используют эти данные в буфере. Ключ к этой проблеме - гарантировать, что производители не будут добавлять данные, когда буфер заполнен, а потребители не будут потреблять данные, когда буфер пуст.

Решение: (1) использовать механизм для защиты синхронизации между производителем и потребителем; (2) построить конвейер между производителем и потребителем. Первый метод более эффективен, прост в реализации и код более управляем, что является обычным режимом. Второй тип конвейерного буфера нелегко контролировать, передаваемый объект данных нелегко инкапсулировать и т. Д., И это непрактично. Таким образом, в этой статье описывается только проблема производителя / потребителя механизма синхронизации.

Суть проблемы синхронизации заключается в том, как обеспечить целостность одного и того же ресурса при одновременном доступе нескольких потоков. Обычно используемый метод синхронизации заключается в использовании механизма сигнала или блокировки, чтобы гарантировать доступ к ресурсам не более чем одному потоку в любое время. Язык Java реализует полную объективацию в многопоточном программировании и обеспечивает хорошую поддержку механизма синхронизации. В Java существует четыре метода поддержки синхронизации: первые три - это синхронные методы, а один - конвейерный.  
(1) метод wait () / notify ()  
(2) метод await () / signal ()  
(3) Метод блокировки очереди BlockingQueue  
（4）PipedInputStream / PipedOutputStream

**Решение проблемы producer-consumer с помощью семафоров**

Одной из типовых задач, требующих организации взаимодействия процессов, является задача ***producer-consumer*** (производитель-потребитель). Пусть два процесса обмениваются информацией через буфер ограниченного размера. Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда. На этом уровне деятельность потребителя и производителя можно описать следующим образом.

Producer: while(1) {

produce\_item;

put\_item;

}

Consumer: while(1) {

get\_item;

consume\_item;

}

Если буфер заполнен, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации. Если буфер пуст, то потребитель должен дожидаться нового сообщения. Как можно реализовать эти условия с помощью *семафоров*? Возьмем три *семафора*: empty, full и *mutex*. *Семафор* full будем использовать для гарантии того, что потребитель будет ждать, пока в буфере появится информация. *Семафор* empty будем использовать для организации ожидания производителя при заполненном буфере, а *семафор* *mutex* – для организации взаимоисключения на критических участках, которыми являются действия put\_item и get\_item (операции "положить информацию" и "взять информацию" не могут пересекаться, так как в этом случае возникнет опасность искажения информации). Тогда решение задачи на C-подобном языке выглядит так:

Semaphore mutex = 1;

Semaphore empty = N; /\* где N – емкость буфера\*/

Semaphore full = 0;

Producer:

while(1) {

produce\_item;

P(empty);

P(mutex);

put\_item;

V(mutex);

V(full);

}

Consumer:

while(1) {

P(full);

P(mutex);

get\_item;

V(mutex);

V(empty);

consume\_item;

}

Легко убедиться, что это действительно корректное решение поставленной задачи. Попутно заметим, что *семафоры* использовались здесь для достижения двух целей: организации взаимоисключения на критическом участке и *взаимосинхронизации* скорости работы процессов.

**Проблема производителя и потребителя**

Рассмотрим проблему производителя и потребителя, также известную как проблема ограниченного буфера.

**Два процесса совместно используют буфер ограниченного размера**. ***Один из них, производитель, помещает данные в этот буфер, а другой, потребитель, считывает их оттуда.***

**Трудности начинаются в тот момент, когда производитель хочет поместить в буфер очередную порцию данных и обнаруживает, что буфер полон**.  
Для производителя **решением является ожидание, пока потребитель полностью или частично не очистит буфер**. Аналогично, если потребитель хочет **забрать данные из буфера, а буфер пуст,** потребитель уходит в состояние ожидания и выходит из него, как только производитель положит что-нибудь в буфер и разбудит его.

Нам нужна **переменная count** для от­слеживания количества элементов в буфере. Если максимальное число элементов, хранящихся в буфере, равно **N**, программа производителя должна проверить, не равно ли **N** значение count прежде, чем поместить в буфер следующую порцию данных. **Если значение count равно N, то производитель уходит в состояние ожидания; в противном случае производитель помещает данные в буфер и увеличивает значение count.**

Теперь давайте **вернемся к состоянию состязания.**  
Его возникновение возможно, поскольку **доступ к переменной count не ограничен**. Может возникнуть следующая ситуация: буфер пуст, и потребитель только что считал значение перемен­ной count, чтобы проверить, не равно ли оно нулю. В этот момент планировщик передал управление производителю, производитель поместил элемент в буфер и увеличил значение count, проверив, что теперь оно стало равно 1. Зная, что перед этим оно было равно 0 и потребитель находился в состоянии ожидания, производитель активизирует его с помощью вызова wakeup.

Но потребитель не был в состоянии ожидания, так что **сигнал активизации про­пал впустую**. Когда управление перейдет к потребителю, он вернется к считанному когда-то значению count, обнаружит, что оно равно 0, и **уйдет в состояние ожидания**.

**Рано или поздно производитель наполнит буфер и также уйдет в состояние ожидания. Оба процесса так и останутся в этом состоянии.**

Суть проблемы в данном случае состоит в том, что **сигнал активизации, при­шедший к процессу, не находящемуся в состоянии ожидания, пропадает**.  
**Если бы не это, проблемы бы не было.**  
Быстрым решением может быть **добавление бита ожидания активизации**. Если сигнал активизации послан процессу, не находящемуся в состоянии ожидания, этот бит устанавливается. Позже, когда процесс пытается уйти в состояние ожидания, бит ожидания активизации сбрасывается, но процесс остается активным. **Этот бит исполняет роль копилки сигналов активизации.**

, сообщен

Механизмы синхрони

**Мьютексы** Мьютекс – двоичный семафор, обычно используемый для организации согласованного доступа к неделимому общему ресурсу. Мьютекс может принимать значения 1 (свободен) и 0 (занят). Операции над мьютексами

•acquire(mutex) – уменьшить (занять) мьютекс

•release(mutex) – увеличить (освободить) мьютекс

•tryacquire(mutex) – часто реализуемая неблокирующая операция, выполняющая попытку уменьшить (занять) мьютекс

Мьютексы в конкретных реализациях могут иметь дополнительные свойства

•Запоминание владельца – освободить мьютекс может только поток, захвативший его

•Рекурсивность – поток может многократно захватить мьютекс (вызывать aquire()); для освобождения мьютекса поток должен соответствующее число раз вызвать release()

•Наследование приоритета – поток, захвативший мьютекс, временно наследует максимальный из приоритет потоков, ждущих освобождения данного мьютекса

**Мониторы**

Мониторы – высокоуровневый механизм взаимодействия и синхронизации процессов, обеспечивающий доступ к неразделяемым ресурсам. Предложены Хоаром (Hoare) в 1974 г. Пер Бринч Хансен первым, описал и реализовал мониторы, основывая их на идеях Хоара.

Монитор – это конструкция языка программирования, поддерживающая управляемый доступ к разделяемым данным. Монитор инкапсулирует:

•разделяемые критические данные;

•функции, использующие разделяемые данные;

•синхронизацию выполнения параллельных потоков, вызывающих указанные функции.

Доступ к переменным монитора, реализуется **только посредством** вызова предоставленных функций.

•Только **один поток** может находиться в мониторе в любой момент времени, если второй поток пытается вызвать метод монитора, он переходит в состояние ожидания до выхода из монитора первого потока.

•Код синхронизации добавляется **компилятором**.

Языки программирования, поддерживающие мониторы:

•Ада

•C# (и другие языки, использующие .NET Framework)

•Concurrent Pascal

•D

•Java (с помощью ключевого слова synchronized)

•Mesa

•Модула-3

•Ruby

•Squeak Smalltalk

•uC++

•и др.



Тупиковая ситуация при использовании монитора



Операции над условными переменными **wait(cv)** – выполняется потоком, который хочет подождать наступления события. Снимает блокировку монитора, после чего другой поток может войти в него; ожидает, пока какой-либо другой поток не подаст условный сигнал. **signal(cv)** – выполняется потоком, сигнализирующем о наступлении события. Пробуждает максимум один ожидающий поток; если отсутствуют потоки, ожидающие события, информация о приходе сигнала теряется. **broadcast(cv)** – также выполняется потоком, сигнализирующем о наступлении события. Пробуждает все ожидающие события потоки.

ац

**Мониторы Хоара и мониторы Меса *Мониторы Хоара обрабатывают вызов signal(cv) следующим образом:***

•немедленно запускается поток, ожидавший сигнала;

•поток, пославший сигнал, блокируется и остается блокированным все время, пока выполняется поток, который он вывел из состояния ожидания. ***Мониторы Меса обрабатывают вызов signal(cv) несколько другим способом:***

•ожидающий поток переводится в состояние "готов к выполнению", а поток, пославший сигнал, продолжает исполнение;

•ожидавший поток запускается при выходе потока, пославшего сигнал, из монитора или его перехода в состояние ожидания.

и

**Сообщения** Сообщения –способ межпроцессного и межпоточного взаимодействия, позволяющий потокам подавать сигналы друг другу и обмениваться данными. Используется для организации взаимодействия потоков, выполняющихся на различных узлах сети.

и

Типы доставки сообщений **Асинхронный** поток, посылающий сообщение, инициирует процесс доставки сообщения, после чего продолжает свою работу **Синхронный** поток, пославший сообщение, дожидается подтверждения его получения принимающим потоком

: с

Пример

•Имеются разделяемые данные, над которыми требуется многократно выполнить некоторую операцию.

•Есть два потока, которые выполняют данную операцию над частями разделяемых данных.

•Каждое следующее выполнение операции требует, чтобы предыдущее было полностью завершено обоими потоками.

е

Адресация **Прямая** send(P, message) – послать сообщение message процессу P receive(Q, message) – получить сообщение message от процесса Q **Непрямая** send(A, message) – послать сообщение message в буфер A receive(A, message) – получить сообщение message из буфера A мафоры, мьютексы, мониторы, сообщения. ия.

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, *исключает*для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется **взаимоисключением** (mutual exclusion). Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними взаимоисключениями уже не обойтись, нужна *взаимосинхронизация* поведения программ.