ОСРВ

Определение СРВ. Жесткие и мягкие СРВ. Отличие ОСРВ от ОС общего назначения. Системы разработки и системы исполнения.

1. Существует несколько определений систем реального времени (СРВ), большинство из которых даже противоречат друг другу.

Система реального времени это аппаратно-программный комплекс, реагирующий в предсказуемые времена на непредсказуемый поток внешних событий

Это означает, что:

А)Система должна успеть отреагировать на событие, произошедшее на объекте, в течение времени, критического для этого события. Величина критического времени для каждого события определяется объектом и самим событием, и, естественно, может быть разной, но время реакции системы должно быть предсказано (вычислено) при создании системы. Отсутствие реакции в предсказанное время считается ошибкой для систем реального времени.

Б)Система должна успевать реагировать на одновременно происходящие события. Даже если два или больше внешних событий происходят одновременно, система должна успеть среагировать на каждое из них в течение интервалов времени, критического для этих событий.

Пример: Управление роботом, берущим деталь с ленты конвейера. Деталь движется, и робот имеет лишь маленькое временное окно, когда он может ее взять. Если он опоздает, то деталь уже не будет на нужном участке конвейера, и, следовательно, работа не будет сделана, несмотря на то, что робот находится в правильном месте. Если он позиционируется раньше, то деталь еще не успеет подъехать, и робот заблокирует ей путь.

Другим примером может быть самолет, находящийся на автопилоте. Сенсорные серводатчики должны постоянно передавать в управляющий компьютер результаты измерений. Если результат какого-либо измерения будет пропущен, то это может привести к недопустимому несоответствию между реальным состоянием систем самолета и информацией о нем в управляющей программе.

Различают системы реального времени двух типов - системы жесткого реального времени и системы мягкого реального времени.

Системы жесткого реального времени не допускают никаких задержек реакции системы ни при каких условиях, так как:

- результаты могут оказаться бесполезны в случае опоздания,
- может произойти катастрофа в случае задержки реакции,
- стоимость опоздания может оказаться бесконечно велика.

Примеры систем жесткого реального времени - бортовые системы управления, системы аварийной защиты, регистраторы аварийных событий.

Системы мягкого реального времени характеризуются тем, что задержка реакции не критична, хотя и может привести к увеличению стоимости результатов и снижению производительности системы в целом.

Пример - работа сети. Если система не успела обработать очередной принятый пакет, это приведет к таймауту на передающей стороне и повторной посылке (в зависимости от протокола, конечно). Данные при этом не теряются, но производительность сети снижается.

Основное отличие между системами жесткого и мягкого реального времени можно выразить так: система жесткого реального времени никогда не опоздает с реакцией на событие, система мягкого реального времени - не должна опаздывать с реакцией на событие.

Все процессоры и ресурсы компьютера управляются системой программного обеспечения, которую мы называем **операционной системой реального времени и**, которая может быть использована для построения систем жесткого реального времени.

Большинство программного обеспечения ориентировано на «мягкое» реальное время, а задача СРВ – обеспечить уровень безопасного функционирования системы, даже если управляющая программа никогда не закончит своей работы.

ОС общего назначения, ориентированы на оптимальное распределение ресурсов компьютера между пользователями и задачами. В ОСРВ главной задачей является успеть среагировать на события, происходящие на объекте.

Одно из коренных внешних отличий это четкое разграничение систем разработки и систем исполнения.

Система исполнения в ОСРВ это набор инструментов (ядро, драйверы, исполняемые модули), обеспечивающих функционирование приложения реального времени.

Система исполнения в ОСРВ и компьютер, на котором она исполняется называют **"целевой"** системой.

Система разработки - набор средств, обеспечивающих создание и отладку приложения реального времени (компиляторы, отладчики и т.д)

Характеристики ОСРВ. Механизмы реального времени.

В связи со специфичность решаемых задач, ОСРВ должна обладать определенными свойствами.

Приблизительное время реакции в зависимости от области применения ОСРВ может быть следующее:

- математическое моделирование - несколько **микросекунд**
- радиолокация - несколько **миллисекунд**
- складской учет - несколько **секунд**
- управление производством - несколько **минут**

Видно, что времена очень разнятся и накладывают различные требования на вычислительную установку, на которой работает ОСРВ.

Интервал времени - от события на объекте и до выполнения первой инструкции в программе обработки этого события является **временем реакции системы на события.**

Из чего оно складывается? интервал времени - от возникновения запроса на прерывание и до выполнения первой инструкции обработчика определяется целиком свойствами операционной системы и архитектурой компьютера. В ОСРВ заложен параллелизм, возможность одновременной обработки нескольких событий, поэтому все ОСРВ являются многозадачными. Для того, чтобы уметь оценивать накладные расходы системы при обработке параллельных событий, необходимо знать время, которое система затрачивает на передачу управления от процесса к процессу, то есть **время переключения контекста**.

Время перезагрузки системы. Этот параметр важен для систем, от которых требуется непрерывная работа.. В таких случая важным является такое свойство системы как ее живучесть при незапланированных перезагрузках. Большинство операционных систем реального времени устойчивы к перезагрузкам и могут быть прерваны и перезагружены в любое время.

Вычислительные установки, на которых применяются ОСРВ, можно разделить на три группы:

- **«Обычные» компьютеры**.
- **Промышленные компьютеры.**
- **Встраиваемые системы.** Устанавливаются внутрь оборудования, которым они управляют. Для крупного оборудования могут по исполнению совпадать с промышленными компьютерами. Для оборудования поменьше могут представлять собой процессор с сопутствующими элементами, размещенными на одной плате с другими электронными компонентами оборудования. Для миниатюрных систем процессор с сопутствующими элементами может быть частью одной из интегральных схем оборудования.

В связи с особенностями оборудования ОСРВ (промышленные компьютеры и встаиваемые системы часто являются бездисковыми.) должны обладать следующими свойствами:

Размеры системы. Размеры ядра и обслуживающих модулей системы должны быть невелики.

Возможность исполнения системы из ПЗУ (ROM). Система должна иметь возможность осуществлять загрузку из ПЗУ. Для экономии места в ПЗУ часть системы может храниться в сжатом виде и загружаться в ОЗУ по мере необходимости. Часто система позволяет исполнять код как в ПЗУ, так и в ОЗУ. При наличии свободного места в ОЗУ система может копировать себя из медленного ПЗУ в более быстрое ОЗУ.

К дополнительным свойствам ОСРВ можно отнести следующие:

Наличие необходимых драйверов устройств. Если разрабатываемая система имеет обширную периферию, то наличие уже готовых драйверов может оказать большое влияние на выбор операционной системы.

Поддержка процессоров различной архитектуры. В связи с тем, что в промышленных компьютерах, серверах, встраиваемых системах широко распространены процессоры разной архитектуры с различной системой команд, ОСРВ по возможности должна поддерживать как можно более широкий ряд процессоров.

Специальный кроссплатформенный инструментарий разработчика. Это связано с тем, что разработка СРВ часто проводится на «обычном» компьютере, отличном по архитектуре от компьютера, на котором будет устанавливаться СРВ. При этом ОС на этих двух компьютерах также может не совпадать.

Механизмы реального времени. Которые в свою очередь делают СРВ предсказуемой: *а) Система приоритетов и алгоритмы диспетчеризации*

В многозадачных ОС общего назначения используются, как правило, различные модификации алгоритма круговой диспетчеризации, основанные на понятии непрерывного кванта времени ("time slice"), предоставляемого процессу для работы. Планировщик по истечении каждого кванта времени просматривает очередь активных процессов и принимает решение, кому передать управление, основываясь на приоритетах процессов. Приоритеты могут быть фиксированными или меняться со временем - это зависит от алгоритмов планирования в данной ОС, но рано или поздно процессорное время получат все процессы в системе.

Алгоритмы круговой диспетчеризации неприменимы в чистом виде в операционных системах реального времени. Основной недостаток - непрерывный квант времени, в течение которого процессором владеет только один процесс. Планировщики же операционных систем реального времени имеют возможность сменить процесс до истечения "time slice", если в этом возникла необходимость. Один из возможных алгоритмов планирования при этом "приоритетный с вытеснением". Мир операционных систем реального времени отличается богатством различных алгоритмов планирования: динамические, приоритетные, монотонные, адаптивные и пр., цель же всегда преследуется одна - предоставить инструмент, позволяющий в нужный момент времени исполнять именно тот процесс, который необходим.

Б) Механизмы межзадачного взаимодействия. Другой набор механизмов реального времени относится к средствам синхронизации процессов и передачи данных между ними.. К таким механизмам относятся: семафоры, мьютексы, события, сигналы, средства для работы с разделяемой памятью, каналы данных (pipes), очереди сообщений

В) *Средства для работы с таймерами.* Такие инструменты, как средства работы с таймерами, необходимы для систем с жестким временным регламентом, поэтому развитость средств работы с таймерами - необходимый атрибут операционных систем реального времени. Эти средства, как правило, позволяют:

- измерять и задавать различные промежутки времени (от 1 мкс и выше),
- генерировать прерывания по истечении временных интервалов,
- создавать разовые и циклические будильники

Архитектура ОСРВ. Классы ОСРВ.

По своей внутренней архитектуре ОСРВ можно условно разделить на монолитные ОС, ОС на основе микроядра и объектно-ориентированные ОС.

Рис 1. ОСРВ с монолитной архитектурой

ОСРВ с монолитной архитектурой можно представить в виде

- Прикладного уровня: состоит из работающих прикладных процессов;
- Системного уровня: состоит из монолитного ядра операционной системы, в котором можно выделить следующие части:
  - интерфейс между приложениями и ядром (API)
  - собственно ядро системы
  - интерфейс между ядром и оборудованием (драйверы устройств).

API в таких системах играет двойную роль:

1. управление взаимодействием прикладных процессов и системы;
2. обеспечение непрерывности выполнения кода системы (т.е. отсутствие переключения задач во время исполнения кода системы).

Основным преимуществом монолитной архитектуры является относительная быстрота работы по сравнению с другими архитектурами.

Недостатки монолитной архитектуры:

1. Системные вызовы, требующие переключения уровней привилегий должны реализовывать API как прерывания или ловушки .Это сильно увеличивает время их работы.
2. Ядро не может быть прервано пользовательской задачей. Это может привести к тому, что высокоприоритетная задача может не получить управление из-за работы низко-приоритетной.
3. Сложность переноса на новый архитектуры процессора из-за значительных ассемблерных вставок.
4. Негибкость и сложность развития: изменение части ядра системы требует его полной перекомпиляции.

Модульная архитектура (на основе микроядра) появилась как попытка убрать узкое место – API и облегчить модернизацию системы и перенос ее на новые процессоры.

API обеспечивает связь прикладных процессов и специального модуля – менеджера процессов. Однако, теперь микроядро играет двойную роль:

1. управление взаимодействием частей системы (например, менеджеров процессов и файлов)
2. обеспечение непрерывности выполнения кода системы (т.е. отсутствие переключения задач во время исполнения микроядра).

Недостатки у модульной архитектуры фактически те же, что и у монолитной. Проблемы перешли с уровня API на уровень микроядра. Системный интерфейс по-прежнему не допускает переключения задач во время работы микроядра, только сократилось время пребывания в этом состоянии. API по-прежнему может быть реализован только на ассемблере, проблемы с переносимостью микроядра уменьшились (в связи с сокращением его размера), но остались.

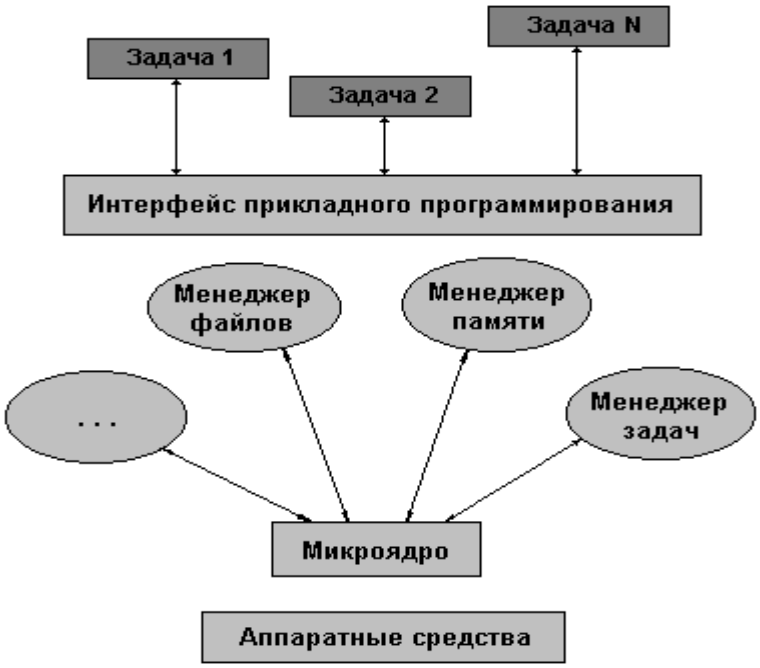


Рис 2. ОСРВ на основе микроядра

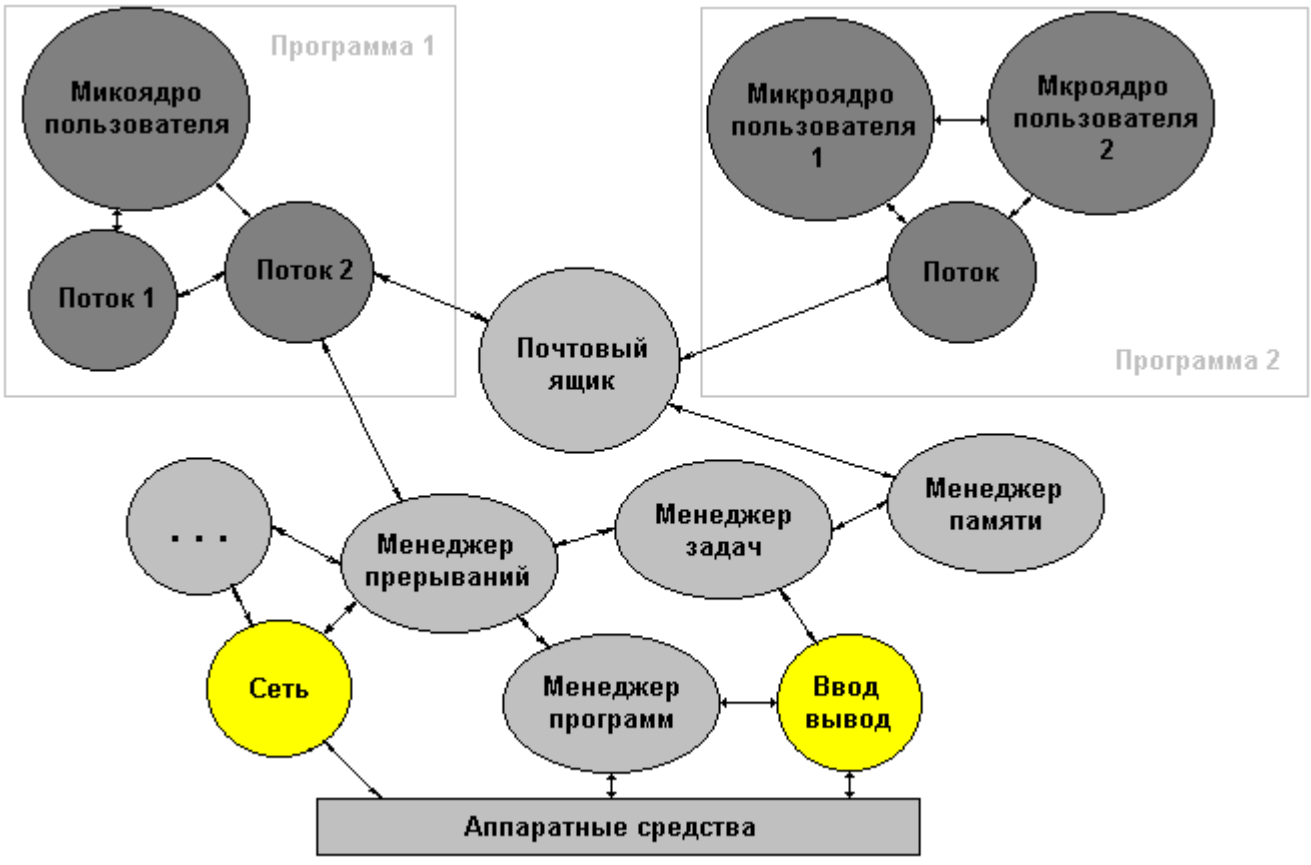


Рис 3. Объектно-ориентированная ОСРВ

Сочетание объекно-ориентированных приложений и процедурных операционных систем имеет ряд недостатков:

- В едином работающем комплексе (приложение + ОСРВ) разные компоненты используют разные подходы к разработке программного обеспечения.
- Не используются все возможности объектно-ориентированного подхода.
- Возникают некоторые потери производительности из-за разного типа интерфейсов в ОСРВ и приложении.

Естественно, возникает идея строить саму ОСРВ, используя объектно-ориентированный подход.

Объектная архитектура на основе объектов-микроядер. В этой архитектуре API отсутствует вообще. Взаимодействие между компонентами системы (микроядрами) и пользовательскими процессами осуществляется посредством обычного вызова функций, поскольку и система, и приложения написаны на одном языке (Для ОСРВ SoftKernel это C++).Это обеспечивает максимальную скорость системных вызовов.

Фактическое равноправие всех компонент системы обеспечивает возможность переключения задач в любое время, т.е. система полностью preemptible.

Объектно-ориентированный подход обеспечивает модульность, безопасность, легкость модернизации и повторного использования кода.

Роль API играет компилятор и динамический редактор объектных связей (linker). При старте приложения динамический linker загружает нужные ему микроядра (т.е., в отличие от предыдущих систем, не все компоненты самой операционной системы должны быть загружены в оперативную память). Если микроядро уже загружено для другого приложения, оно повторно не загружается, а использует код и данные уже имеющегося ядра. Это позволяет уменьшить объем требуемой памяти.

Поскольку разные приложения разделяют одни микроядра, то они должны работать в одном адресном пространстве. Следовательно, система не может использовать виртуальную память и тем самым работает быстрее (так как исключаются задержки на трансляцию виртуального адреса в физический).

Поскольку все приложения и сами микроядра работают в одном адресном пространстве, то они загружаются в память, начиная с неизвестного на момент компиляции адреса. Следовательно, приложения и микроядра не должны зависеть от начального адреса (как по коду, так и поданным (последнее обеспечить значительно сложнее)). Это свойство автоматически обеспечивает возможность записи приложений и модулей в ПЗУ, с их последующим исполнением как в самом ПЗУ, так и оперативной памяти.

Микроядра по своим характеристикам напоминают структуры, используемые в других операционных системах. Однако есть и свои различия.

- **Микроядра и модули.** Многие ОС поддерживают динамическую загрузку компонент системы, называемых модулями. Однако, модули не поддерживают объектно-ориентрованный подход (напомним, микроядро фактически является представителем некоторого класса). Далее, обмен информацией с модулями происходит посредством системных вызовов, что достаточно дорого.
- **Микроядра и драйверы.** Многие ОС поддерживают возможность своего расширения посредством драйверов (специальных модулей, обычно служащих для поддержки оборудования). Однако, драйверы часто должны быть статически связаны с ядром (т.е. образовывать с ним связанный загрузочный образ еще до загрузки) и должны работать в привилегированном (суперпользовательском) режиме. Далее, как модули они не поддерживают объектно-ориентированный подход и доступны приложению только посредством системных вызовов.
- **Микроядра и DLL.** Многие системы оформляют библиотеки, из которых берутся функции при динамическом связывании, в виде специальных модулей, называемых DLL (Dynamically Linked Libraries – динамически связываемые библиотеки). DLL обеспечивает разделение своего кода и данных для всех работающих приложений, в то время как для микроядер можно управлять доступом для каждого конкретного приложения. DLL не поддерживает объектно-ориентированный подход, код DLL не является позиционно-независимым и не может быть записан в ПЗУ.

Системы реального времени можно разделить на 4 класса.

1-й класс: исполнительные СРВ.

Это программы для программируемых микропроцессоров, встраиваемых в различные устройства, очень небольшие и обычно написаны на языке низкого уровня типа ассемблера или PLM. Внутрисхемные эмуляторы пригодны для отладки, но высокоуровневые средства разработки и отладки программ не применимы. Операционная среда обычно недоступна.

Признаки систем этого типа - различные платформы для систем разработки и исполнения. Приложение реального времени разрабатывается на host- компьютере (компьютере системы разработки), затем компонуется с ядром и загружается в целевую систему для исполнения. Как правило, приложение реального времени - это одна задача и параллелизм здесь достигается с помощью нитей (threads).

Системы этого типа обладают рядом достоинств, среди которых главное - скорость и реактивность системы. Главная причина высокой реактивности систем этого типа - наличие только нитей(потоков) и, следовательно, маленькое время переключения контекста между ними (в отличие от процессов).

С этим главным достоинством связан и ряд недостатков: зависание всей системы при зависании нити, проблемы с динамической подгрузкой новых приложений

2-й класс: минимальное ядро системы реального времени.

На более высоком уровне находятся системы реального времени, обеспечивающие минимальную среду исполнения. Предусмотрены лишь основные функции, а управление памятью и диспетчер часто недоступны. Ядро представляет собой набор программ, выполняющих типичные, необходимые для встроенных систем низкого уровня функции, такие, как операции с плавающей запятой и минимальный сервис ввода/вывода. Прикладная программа разрабатывается в инструментальной среде, а выполняется, как правило, на встроенных системах.

В этот класс входят системы с монолитным ядром, где и содержится реализация всех механизмов реального времени этих операционных систем.

Системы этого класса, как правило, модульны, хорошо структурированы, имеют наиболее развитый набор специфических механизмов реального времени, компактны и предсказуемы. Наиболее популярные системы этого класса: OS9, QNX.

Одна из особенностей систем этого класса - высокая степень масштабируемости. На базе этих ОС можно построить как компактные системы реального времени, так и большие системы серверного класса.

3-й класс: ядро системы реального времени и инструментальная среда. Этот класс систем обладает многими чертами ОС с полным сервисом. Разработка ведется в инструментальной среде, а исполнение - на целевых системах. Этот тип систем обеспечивает гораздо более высокий уровень сервиса для разработчика прикладной программы. Сюда включены такие средства, как дистанционный символьный отладчик, протокол ошибок и другие средства CASE. Часто доступно параллельное выполнение программ.

4-й класс: ОС с полным сервисом. Такие ОС могут быть применены для любых приложений реального времени. Разработка и исполнение прикладных программ ведутся в рамках одной и той же системы.

Область применения расширений реального времени - большие системы реального времени, где требуется визуализация, работа с базами данных, доступ в интернет и пр.

Функции ядра ОСРВ. Процессы. Состояния процесса. Жизненный цикл процесса. Потоки. Приоритеты процессов.

Ядро может обеспечивать сервис пяти типов:

Синхронизация ресурсов. Метод синхронизации требует ограничить доступ к общим ресурсам (данным и внешним устройствам). Наиболее распространенный тип примитивной синхронизации - двоичный семафор, обеспечивающий избирательный доступ к общим ресурсам. Так, процесс, требующий защищенного семафором ресурса, вынужден ожидать до тех пор, пока семафор не станет доступным, что свидетельствует об освобождении ожидаемого ресурса, и, захватив ресурс, установить семафор. В свою очередь, другие процессы также будут ожидать доступа к ресурсу вплоть до того момента, когда семафор возвратит соответствующий ресурс системе распределения ресурсов. Системы, обладающие большей ошибкоустойчивостью, могут иметь счетный семафор. Этот вид семафора разрешает одновременный доступ к ресурсу лишь определенному количеству процессов.

Межзадачный обмен. Часто необходимо обеспечить передачу данных между программами внутри одной и той же системы Кроме того, во многих приложениях возникает необходимость взаимодействия с другими системами через сеть. Внутренняя связь может быть осуществлена через систему передачи сообщений. Внешнюю связь можно организовать либо через датаграмму (наилучший способ доставки), либо по линиям связи (гарантированная доставка). Выбор того или иного способа зависит от протокола связи.

Разделение данных. В прикладных программах, работающих в реальном времени, наиболее длительным является сбор данных. Данные часто необходимы для работы других программ или нужны системе для выполнения каких-либо своих функций. Во многих системах предусмотрен доступ к общим разделам памяти. Широко распространена организация очереди данных. Применяется много типов очередей, каждый из которых обладает собственными достоинствами.

Обработка запросов внешних устройств. Каждая прикладная программа в реальном времени связана с внешним устройством определенного типа. Ядро должно обеспечивать службы ввода/вывода, позволяющие прикладным программам осуществлять чтение с этих устройств и запись на них. Для приложений реального времени обычным является наличие специфического для данного приложения внешнего устройства. Ядро должно предоставлять сервис, облегчающий работу с драйверами устройств. Например, давать возможность записи на языках высокого уровня - таких, как Си или Паскаль.

Обработка особых ситуаций. Особая ситуация представляет собой событие, возникающее во время выполнения программы. Она может быть синхронной, если ее возникновение предсказуемо, как, например, деление на нуль. А может быть и асинхронной, если возникает непредсказуемо, как, например, падение напряжения. Предоставление возможности обрабатывать события такого типа позволяет прикладным программам реального времени быстро и предсказуемо отвечать на внутренние и внешние события. Существуют два метода обработки особых ситуаций - использование значений состояния для обнаружения ошибочных условий и использование обработчика особых ситуаций для прерывания ошибочных условий и их корректировки.

Кроме того, важнейшей функцией ядра является **диспетчеризация** (планирование). Планировщик должен определять, какому процессу должно быть передано управление, а также должен определить время, выделяемое каждому процессу.

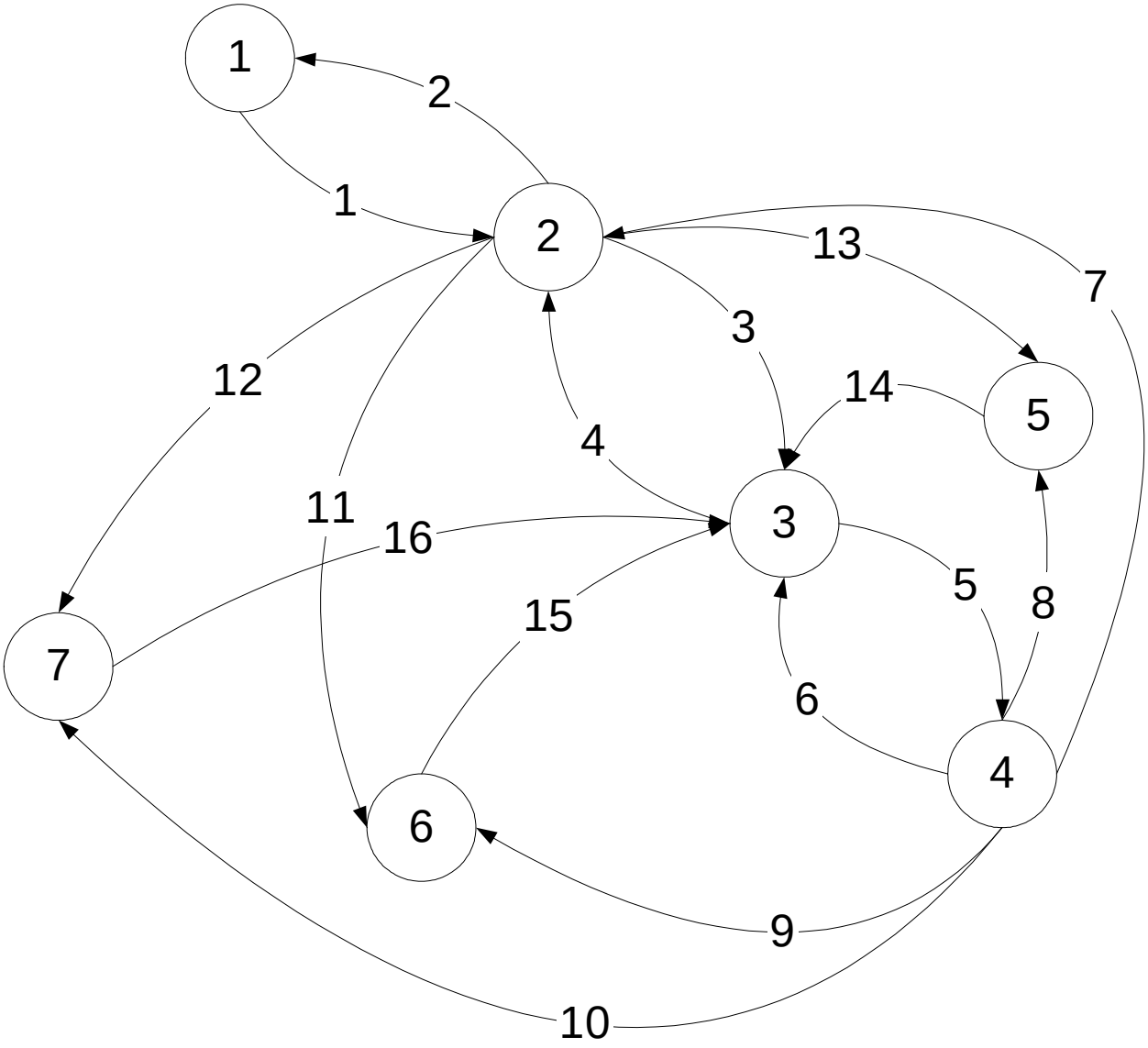
Рассмотрим подробнее, что такое процесс. **Процесс** – это динамическая сущность программы, ее код в процессе своего выполнения. Имеет:

- собственные области памяти под код и данные, включая значения регистров и счетчика команд
- собственный стек
- собственное отображение виртуальной памяти (в системах с виртуальной памятью) на физическую
- собственное состояние.

Процесс может находиться в одном из следующих типичных состояний:

- «остановлен» - процесс остановлен и не использует процессор (например, в таком состоянии процесс находится сразу после создания)
- «терминирован» - процесс терминирован и не использует процессор (например, процесс закончился, но еще не удален операционной системой)
- «ждет» - процесс ждет некоторого события (им может быть аппаратное или программное прерывание, сигнал или другая форма межпроцессного взаимодействия)
- «готов» - процесс не остановлен, не терминирован, не ожидает, не удален, но и не работает (например, процесс не может получить доступ к процессору, если в данный момент выполняется другой, более высокоприоритетный процесс)
- «выполняется» - процесс выполняется и использует процессор. В ОСРВ это обычно означает, что этот процесс является самым приоритетным среди всех процессов, находящихся в состоянии «готов»

Рассмотрим более подробно состояния процесса и переходы из одного состояния в другое.



Состояния:

1. не существует
2. не обслуживается
3. готов
4. выполняется
5. ожидает ресурс
6. ожидает назначенное время
7. ожидает события

Переходы:

1. переход 1-2 создание процесса
2. переход 2-1 уничтожение процесса
3. переход 2-3 активизация процесса диспетчером
4. переход 3-2 деактивизация процесса
5. переход 3-4 загрузка на выполнение процесса диспетчером
6. переход 4-3 требование обслуживания от процессора другим процессом (preemption – приоритетное переключение)
7. переход 4-2 завершение процесса
8. переход 4-5 блокировка процесса до освобождения требуемого ресурса
9. переход 4-6 блокировка процесса до истечения заданного времени
10. переход 4-7 блокировка процесса до прихода события
11. переход 2-6 активизация процесса приводит к ожиданию временной задержки
12. переход 2-7 активизация процесса приводит к ожиданию события
13. переход 2-5 активизация процесса приводит к ожиданию освобождения ресурса

14. переход 5-3 активизация процесса из-за освобождения ожидавшегося ресурса
15. переход 6-3 активизация процесса по истечении заданного времени
16. переход 7-3 активизация процесса из-за прихода ожидавшегося события

Таким образом, каждый процесс имеет свой жизненный цикл, состоящий из 4 стадий:

1. создание
2. загрузка
3. выполнение
4. завершение.

Создание процесса обычно состоит из присвоения новому процессу идентификатора процесса и подготовки информации, которая определяет окружение процесса.

Загрузка процесса означает загрузку в память кода процесса.

После того, как код программы загружен, процесс готов к выполнению. Он начинает конкурировать с другими процессами за ресурсы процессора. Процесс может выполняться, а может блокироваться по тем или иным причинам.

Завершение процесса означает освобождение всех ресурсов, выделенных процессу – файловых дескрипторов, памяти и т.д.

Модель потока базируется на двух независимых концепциях: группировании ресурсов и выполнении программы. С одной стороны, процесс можно рассматривать как способ группирования родственных ресурсов в одну группу. У процесса есть адресное пространство, содержащее текст программы и данные, а также другие ресурсы. Ресурсами могут быть открытые файлы, обработчики сигналов, учетная информация и многое другое. С другой стороны, процесс можно рассматривать как поток исполняемых команд, или просто **поток**. У потока есть счетчик команд, отслеживающий порядок выполнения действий. У него есть регистры, в которых хранятся текущие переменные. У него есть стек, содержащий протокол выполнения процесса, где на каждую процедуру, вызванную, но еще не вернувшуюся, отведен отдельный фрейм. У процесса есть свое состояние. Потоки одного процесса выполняются в одном адресном пространстве, используют одни и те же глобальные переменные, ресурсы.. Таким образом, процессы используются для группирования ресурсов, а потоки являются объектами, поочередно выполняющимися на процессоре.

Каждому процессу и каждому потоку в ОСРВ приписывается некоторое число, называемое **приоритетом**. Чем больше это число, тем важнее процесс или поток. Приоритет может быть **фиксированным** (назначается при создании процесса и не меняется в течение его жизни). В зависимости от алгоритмов планирования приоритет также может изменяться в течение жизни.

Планирование задач. Алгоритмы планирования без переключения и с переключением. Схемы назначения приоритетов. FIFO диспетчеризация. Карусельная диспетчеризация. Адаптивная диспетчеризация.

Определение. Планировщик задач - это модуль (программа), отвечающий за разделение времени имеющихся процессоров между выполняющимися задачами. Отвечает за коммутацию задач из состояния блокировки в состояние готовности, и за выбор задачи (задач - по числу процессоров) из числа готовых для исполнения процессором.

Ключевым вопросом планирования является выбор момента принятия решения:

Во-первых, когда создается новый процесс, необходимо решить, какой процесс запустить, родительский или дочерний. Поскольку оба процесса находятся в состоянии готовности, эта ситуация не выходит за рамки обычного и планировщик может запустить любой из двух процессов.

Во-вторых, планирование необходимо, когда процесс завершает работу. Этот процесс уже не существует, следовательно, необходимо из набора готовых процессов выбрать и запустить следующий.

В-третьих, когда процесс блокируется по какой-либо причине, необходимо выбрать и запустить другой процесс.

В-четвертых, решение по диспетчеризации должно приниматься после разблокировки процесса.

В-пятых, планировщик может принимать решение по истечении кванта времени, отпущенному процессу.

Алгоритмы планирования можно разделить на две категории согласно их поведению после прерываний. Алгоритмы планирования **без переключений**, иногда называемого также **неприоритетным** планированием, выбирают процесс и позволяют ему работать

вплоть до блокировки либо вплоть до того момента, когда процесс сам не отдаст процессор. Процесс не будет прерван, даже если он работает часами. Соответственно, решения планирования не принимаются по прерываниям от таймера. После обработки прерывания таймера управление всегда возвращается приостановленному процессу.

Напротив, алгоритмы планирования **с переключениями**, называемого также **приоритетным** планированием, выбирают процесс и позволяют ему работать некоторое максимально возможное время. Если к концу заданного интервала времени процесс все еще работает, он приостанавливается и управление переходит к другому процессу. Приоритетное планирование требует прерываний по таймеру, происходящих в конце отведенного периода времени (решения планирования могут, например, приниматься при каждом прерывании по таймеру, или при каждом *k*-ом прерывании), чтобы передать управление планировщику.

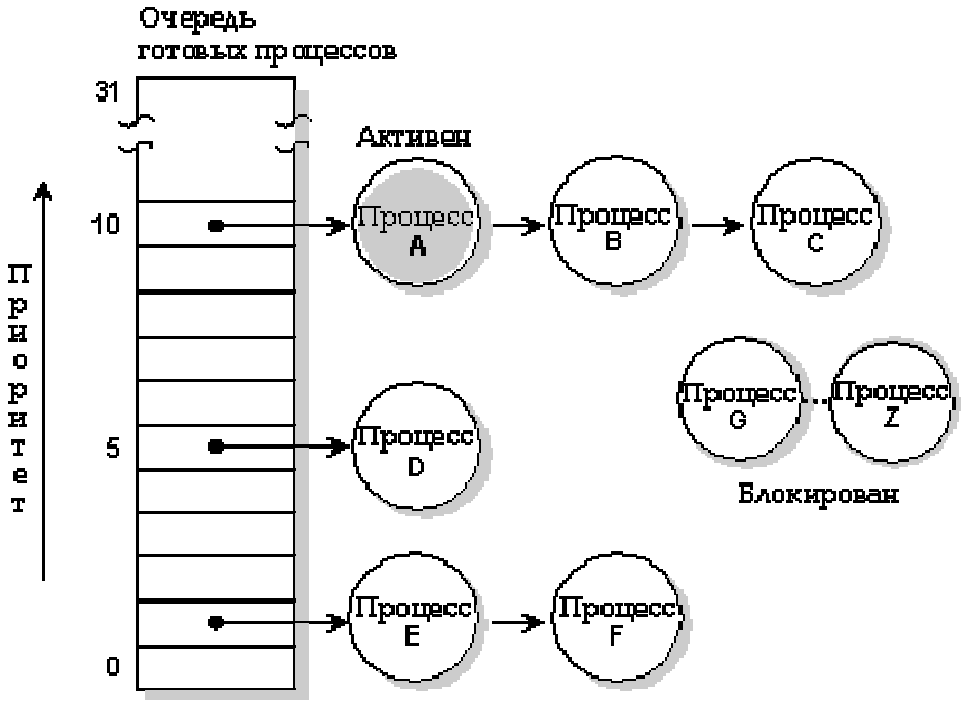
Напомним, что **приоритетом** называется число, приписанное операционной системой (а именно, планировщиком задач) каждому процессу и задаче. Существуют несколько схем назначения приоритетов.

• **Фиксированные** приоритеты - приоритет задаче назначается при ее создании и не меняется в течение ее жизни. Эта схема с различными дополнениями применяется в большинстве систем реального времени. В схемах планирования ОСРВ часто требуется, чтобы приоритет каждой задачи был уникальным, поэтому часто ОСРВ имеют большое число приоритетов (обычно 255 и более).

• **Турнирное** определение приоритета - приоритет последней исполнявшейся задачи понижается.

• Определение приоритета по алгоритму **round robin** - приоритет задачи определяется ее начальным приоритетом и временем ее обслуживания. Чем больше задача обслуживается процессором, тем меньше ее приоритет (но не опускается ниже некоторого порогового значения). Эта схема в том или ином виде применяется в большинстве UNIX систем.

Отметим, что в разных системах различные алгоритмы планирования задач могут вводить новые схемы изменения приоритетов. Например, в системе OS-9 приоритеты ожидающих задач увеличиваются для избежания слишком больших времен ожидания.

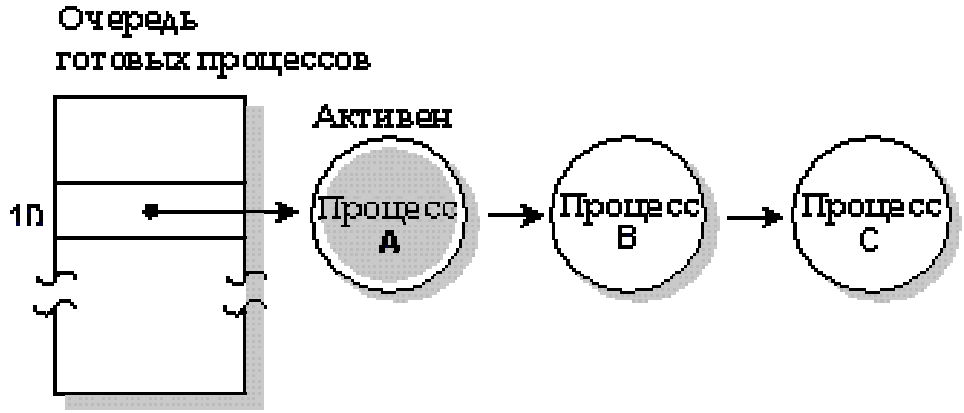


Как видно из рисунка, процессы A-F находятся в состоянии готовности. Процессы G-Z блокированы. Процессы A, B, C имеют наивысший приоритет. Поэтому именно эти процессы будут разделять процессор в соответствии с алгоритмами диспетчеризации. Рассмотрим их.

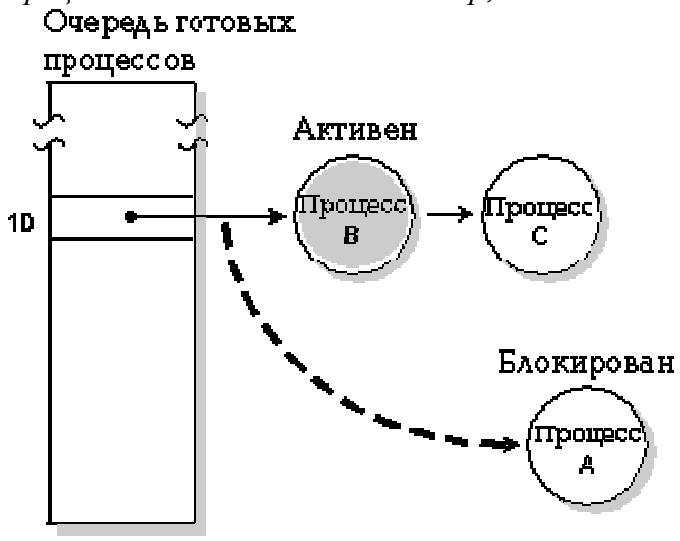
«Первым пришел – первым обслужен» (алгоритм FIFO). Является алгоритмом планирования без переключений. Процессам предоставляется доступ к процессору в том порядке, в котором они его запрашивают. При FIFO диспетчеризации процесс продолжает выполнение, пока не наступит момент, когда он:

- добровольно уступает управление (заканчивается, блокируется и т.п.);
- вытесняется процессом с более высоким приоритетом.

Заметим, что при отсутствии второго условия возможен случай, когда высокоприоритетная задача будет ожидать окончания работы низкоприоритетной.



Процесс А выполняется до тех пор, пока не блокируется.



Процесс А блокируется, процесс В выполняется.

«Кратчайшая задача – первая». Этот алгоритм без переключений предполагает, что временные отрезки работы известны заранее. В этом алгоритме первым выбирается не самая первая, а **самая короткая задача**. Приведем пример.

A	B	C	D

a

B	C	D	A

b

Пусть процессы A,B,C,D имеют время выполнения 8,4,4,4 единиц времени (например, секунд). По алгоритму FIFO они должны быть запущены в том же порядке, в котором они стоят в очереди (вариант а). Тогда время выполнения процесса А будет равно 8, В – 12, С – 16 и D – 20. Среднее время выполнения для этих 4 процессов будет равно 14. Рассмотрим теперь очередь, отсортированную по времени выполнения (вариант b). Теперь время выполнения процесса В будет равно 4, С – 8, D – 12, А – 20. Среднее время в данном варианте будет равно 11.

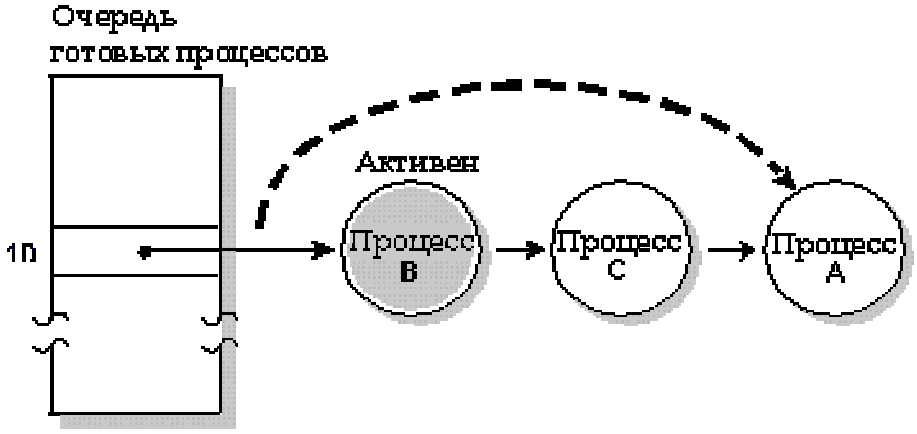
Следует отметить, что данная схема работает только в случае одновременного наличия задач. Если не все процессы доступны с самого начала, можно привести пример, когда алгоритм ухудшает среднее время выполнения.

«Наименьшее оставшееся время выполнения». Является версией предыдущего алгоритма с переключениями. В соответствии с этим алгоритмом планировщик каждый раз выбирает процесс с наименьшим оставшимся временем выполнения. В этом случае также необходимо знать заранее время выполнения каждого процесса. Когда поступает новый процесс, его полное время выполнения сравнивается с оставшимся временем выполнения текущего процесса. Если время выполнения нового процесса меньше, текущий процесс

приостанавливается и управление передается новому процессу. Эта схема позволяет быстро обслуживать короткие процессы.

«Карусельная диспетчеризация (циклическое планирование)». При карусельной диспетчеризации процесс продолжает выполнение, пока не наступит момент, когда он:

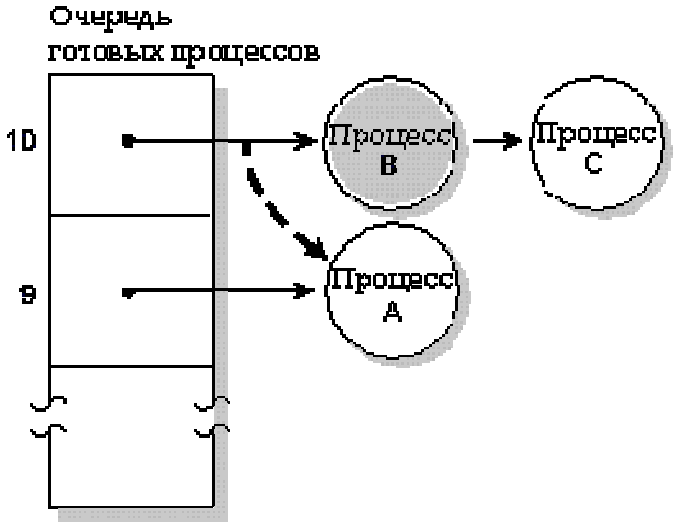
- добровольно уступает управление (т.е. блокируется);
- вытесняется процессом с более высоким приоритетом;
- использовал свой квант времени (*timeslice*). После того, как процесс использовал свой квант времени, управление передается следующему процессу, который находится в состоянии готовности и имеет такой же уровень приоритета.



Карусельная диспетчеризация. Процесс А выполняется до тех пор, пока он не использовал свой квант времени; затем выполняется следующий процесс, находящийся в состоянии готовности (процесс В).

«Адаптивная диспетчеризация». При адаптивной диспетчеризации процесс ведет себя следующим образом:

- Если процесс использовал свой квант времени (т.е. он не блокировался), то его приоритет уменьшается на 1. Это получило название *снижение приоритета* (priority decay). "Пониженный" процесс не будет продолжать "снижаться", даже если он использовал еще один квант времени и не блокировался - он снизится только на один уровень ниже своего исходного приоритета.
- Если процесс блокируется, то ему возвращается первоначальное значение приоритета.



Адаптивная диспетчеризация. Процесс А использовал свой квант времени; его приоритет снизился на 1. Выполняется следующий процесс в состоянии готовности (процесс В).

В системах реального времени наиболее распространенными являются алгоритмы FIFO, адаптивной диспетчеризации, карусельной диспетчеризации и их разновидности.

Планирование периодических процессов. Статический алгоритм планирования RMS. Динамический алгоритм планирования EDF.

Внешние события, на которые система реального времени должна реагировать, можно разделить на **периодические** (возникающие через регулярные промежутки времени) и **непериодические** (возникающие непредсказуемо). Возможно наличие нескольких 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. Если в систему поступает m периодических событий, событие с номером i поступает с периодом P_i и на его обработку уходит C_i секунд работы процессора, все потоки могут быть своевременно обработаны только при выполнении условия

$$\sum_{i=1}^{m} \frac{Ci}{Pi} \le 1$$

Система реального времени, удовлетворяющая этому условию, называется **поддающейся планированию** или **планируемой**. Соотношение $\frac{Ci}{Pi}$ является просто частью процессорного времени, используемого процессом i, а сама сумма – это **коэффициент использования (или коэффициент загруженности) процессора**, который, естественно, не может быть больше 1.

В качестве примера рассмотрим систему с тремя периодическими сигналами с периодами 100, 200, 500 мс соответственно. Если на обработку этих сигналов уходит 50, 30, 100 мс, система является поддающейся планированию, поскольку 0,5+0,15+0,2<1. Даже при добавлении четвертого сигнала с периодом в 1 с системой все равно можно будет управлять при помощи планирования, пока время обработки сигнала не будет превышать 150 мс. Эти расчеты не являются абсолютно верными, так как не учитывают время переключения контекста и не учитывает возникновение непериодических событий.

Алгоритмы планирования заданий могут быть разделены на статические и динамические. *Статические* алгоритмы определяют приемлемый план выполнения заданий по их априорным характеристикам, динамический алгоритм модифицирует план во время исполнения заданий. Издержки на статическое планирование низки, но оно крайне нечувствительно и требует полной предсказуемости той системы реального времени, на которой оно установлено. *Динамическое* планирование связано с большими издержками, но способно адаптироваться к меняющемуся окружению.

Алгоритмы планирования будем рассматривать на примере 3 периодических процессов *A*, *B*, *C*. Предположим, что процесс *A* запускается с периодом 30 мс и временем обработки 10 мс. Процесс *B* имеет период 40 мс и время обработки 15 мс. Процесс *C* запускается каждые 50 мс и обрабатывается за 5 мс. Суммарно эти процессы потребляют 0,808 процессорного времени, что меньше единицы. Соответственно, система в данном примере поддается планированию.

На рис. 1 представлена временная диаграмма работы процессов. Видно, что необходимо применить некоторый алгоритм планирования, так как в определенные моменты времени имеется сразу несколько готовых к выполнению процессов.

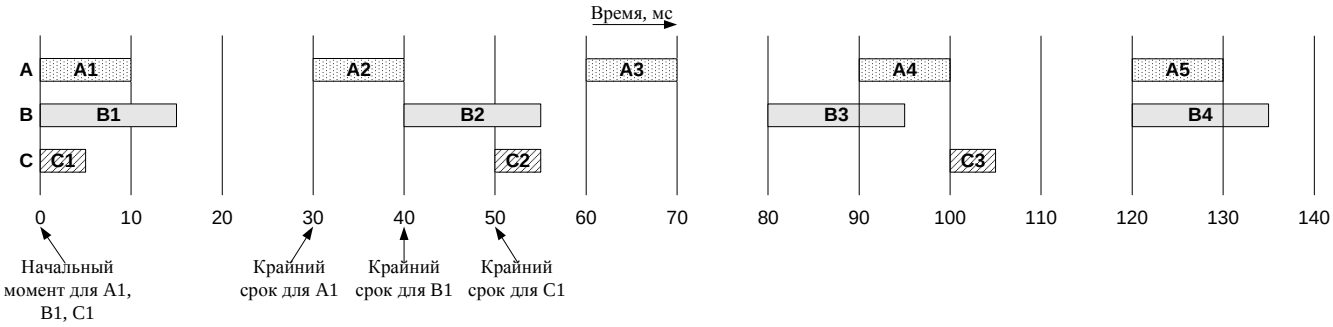


Рис. 1. Три периодических процесса с разным периодом и временем обработки

Классическим примером статического алгоритма планирования реального времени для прерываемых периодических процессов является алгоритм **RMS** (Rate Monotonic Scheduling –

планирование с приоритетом, пропорциональным частоте). Этот алгоритм может использоваться для процессов, удовлетворяющих следующим условиям:

1. Каждый периодический процесс должен быть завершен за время его периода
2. Ни один процесс не должен зависеть от любого другого процесса
3. Каждому процессу требуется одинаковое процессорное время на каждом интервале
4. У непериодических процессов нет жестких сроков
5. Прерывание процесса происходит мгновенно, без накладных расходов.

Алгоритм **RMS** работает, назначая каждому процессу фиксированный приоритет, обратно пропорциональный периоду и, соответственно, прямо пропорциональный частоте возникновения событий процесса. Например, в примере рис. 1 процесс *A* запускается каждые 30 мс (33 раза в секунду) и получает приоритет 33. Процесс *B* запускается каждые 40 мс (25 раз в секунду) и получает приоритет 25. Процесс *C* запускается каждые 50 мс (20 раз в секунду) и получает приоритет 20. Отметим, что реализация алгоритма требует, чтобы у всех процессов были разные приоритеты.

Во время работы планировщик всегда запускает готовый к работе процесс с наивысшим приоритетом, прерывая при необходимости работающий процесс с меньшим приоритетом. Таким образом, в нашем примере процесс *A* может прервать процессы *B* и *C*, процесс *B* может прервать *C*. Процесс C всегда вынужден ждать, пока процессор не освободится.

На рис. 2 показана работа алгоритма планирования для процессов *A*, *B*, *C*. Изначально все три процесса готовы к работе. Выбирается процесс с максимальным приоритетом – *A*. Ему разрешается работать в течение 10 мс, требующихся процессу до завершения. Когда процесс *A* освобождает процессор, начинает работать процесс *B*, а затем процесс *C*. Вместе эти процессы потребляют 30 мс, поэтому, когда процесс *C* заканчивает работу, снова запускается процесс *A*. Этот цикл повторяется до тех пор, пока в момент времени 70 мс у системы начинается период простоя. В момент времени 80 мс процесс *B* переходит в состояние готовности и запускается. Однако в момент времени 90 мс процесс *A*, обладающий более высоким приоритетом, также переходит в состояние готовности. Поэтому он прерывает выполнение процесса *B* и работает до момента времени 100 мс, пока не закончит свою работу. В этот момент времени система должна выбрать между процессом *B*, который не закончил обработку, и *C*, который находится в состоянии готовности. Выбирается процесс *B*, имеющий больший приоритет.

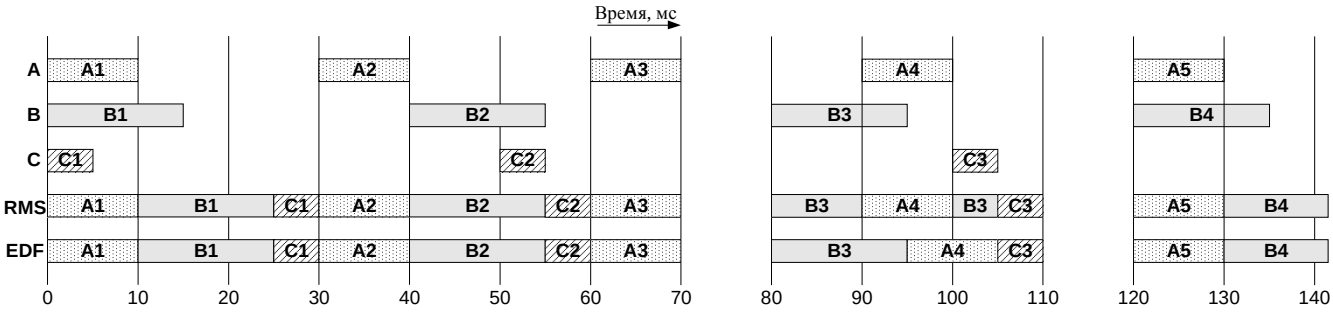


Рис. 2. Пример алгоритмов планирования RMS и EDF.

Алгоритм **RMS** может быть использован только при не слишком высокой загруженности процессора. Предположим, что процесс *A* имеет продолжительность работы не 10 мс, а 15 мс. Коэффициент использования процессора в таком случае равен 0,975. Теоретически для данного случая должен иметься метод планирования. Работа алгоритма показана на рис. 3.

На этот раз процесс *B* завершает обработку к моменту времени 30 мс. В этот же момент процесс *A* снова приходит в состояние готовности. К тому времени, когда он заканчивает работу, снова готов процесс *B* и поскольку у него приоритет больше, чем у *C*, то запускается процесс *B*. Процесс *C* пропускает свой критический срок, алгоритм **RMS** терпит неудачу.

Теоретически было показано, что данный алгоритм гарантированно работает в любой системе периодических процессов при условии

$$\sum_{i=1}^{m} \frac{Ci}{Pi} \le m(2^{1/m} - 1)$$

Таким образом, для 3 процессов максимальная загруженность процессора равна 0,780. Хотя в нашем примере (рис. 2) загруженность процессора составила 0,808, алгоритм **RMS** еще работал, хотя с другими периодами и временем обработки при том же коэффициенте загруженности процессора мог потерпеть неудачу. При увеличении коэффициента загруженности на алгоритм **RMS** не было надежды.

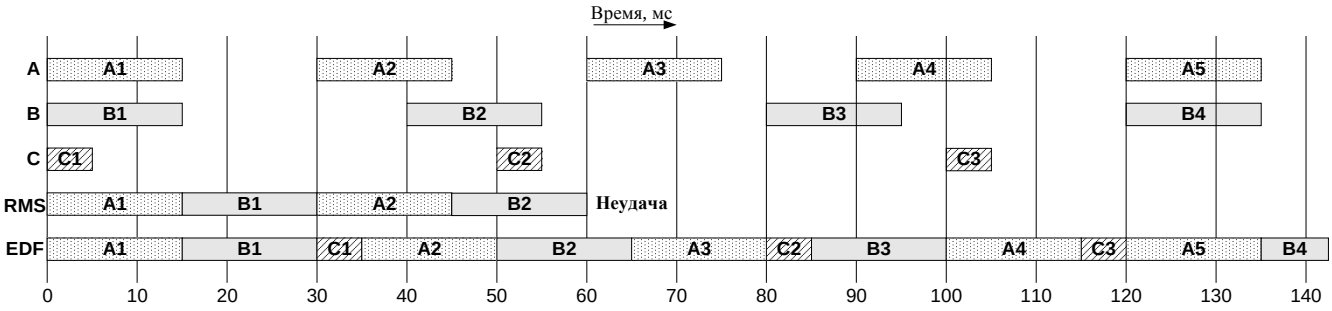


Рис. 3. Пример, в котором алгоритм RMS не может быть использован

Другим популярным алгоритмом планирования является алгоритм **EDF** (Earliest Deadline First – процесс с ближайшим сроком завершения в первую очередь). Алгоритм **EDF** представляет собой динамический алгоритм, не требующий от процессов периодичности. Он также не требует и постоянства временных интервалов использования процессора. Каждый раз, когда процессу требуется процессорное время, он объявляет о своем присутствии и о своем сроке выполнения задания. Планировщик хранит список процессов, сортированный по срокам выполнения заданий. Алгоритм запускает первый процесс в списке, то есть тот, у которого самый близкий по времени срок выполнения. Когда новый процесс переходит в состояние готовности, система сравнивает его срок выполнения со сроком выполнения текущего процесса. Если у нового процесса график более жесткий, он прерывает работу текущего процесса.

Пример работы алгоритма **EDF** показан на рис. 2. Вначале все процессы находятся в состоянии готовности. Они запускаются в порядке своих крайних сроков. Процесс *A* должен быть выполнен к моменту времени 30, процесс *B* должен закончить работу к моменту времени 40, процесс *C* – 50. Таким образом, процесс *A* запускается первым. Вплоть до момента времени 90 выбор алгоритма **EDF** не отличается от **RMS**. В момент времени 90 процесс *A* переходит в состояние готовности с тем же крайним сроком выполнения 120, что и у процесса *B*. Планировщик имеет право выбрать любой из процессов, но поскольку с прерыванием процесса *B* не связано никаких накладных расходов, лучше предоставить возможность продолжать работу этому процессу.

Рассмотрим рис. 3. В момент времени 30 между процессами *A* и *C* возникает спор. Поскольку срок выполнения процесса *C* равен 50 мс, а процесса *A* – 60 мс, планировщик выбирает процесс *C*. Этим данный алгоритм отличается от алгоритма **RMS**, в котором побеждает процесс *A*, как обладающий большим приоритетом. В момент времени 90 процесс *A* переходит в состояние готовности в 4 раз. Предельный срок процесса *A* такой же, что и у текущего процесса, поэтому у планировщика появляется выбор – прервать работу процесса *B* или нет. Поскольку необходимости прерывать процесс *B* нет, то он продолжает работу.

Алгоритм EDF работает с любым набором процессов, для которых возможно планирование. Платой за это является использование более сложного алгоритма.

Межпроцессное взаимодействие. Ресурсы, их характеристики. Состязание процессов. Критические области. Взаимное исключение. Проблемы взаимодействия процессов.

Межпроцессное взаимодействие – это тот или иной способ передачи информации из одного процесса в другой. Наиболее распространенными формами взаимодействия являются:

1. **Разделяемая память** – два или более процесса могут иметь доступ к одному и тому же блоку памяти. В системах с виртуальной памятью организация такого вида взаимодействия требует поддержки со стороны операционной системы, поскольку

необходимо отобразить соответствующие блоки виртуальной памяти на один и тот же блок физической памяти.

2. **Семафоры** – два и более процесса имеют доступ к одной переменной, принимающей значение 0 или 1. Сама переменная часто находится в области данных операционной системы и доступ к ней организуется с помощью специальных функций.
3. **Сигналы** – это сообщения, доставляемые посредством операционной системы процессу. Процесс должен зарегистрировать обработчик этого сообщения у операционной системы, чтобы получить возможность реагировать на него. Часто операционная система извещает процесс сигналом о наступлении какого-либо сбоя, например, делении на 0, или о каком-либо аппаратном прерывании, например, прерывании таймера.
4. **Почтовые ящики** – это очередь сообщений (обычно – тех или иных структур данных), которые помещаются в почтовый ящик процессами и/или операционной системой. Несколько процессов могут ждать поступления сообщения в почтовый ящик и активизироваться по поступлении сообщения. Требует поддержки со стороны операционной системы.

Во многих ОСРВ компоненты операционной системы, также как и пользовательские задачи, способны принимать и передавать сообщения. Сообщения могут быть **асинхронными** и **синхронными**. В первом случае доставка сообщений задаче производится после того, как она в плановом порядке получит управление, а во втором случае циркуляция сообщений оказывает непосредственное влияние на планирование задач. Например, задача, пославшая какое-либо сообщение, немедленно блокируется, если для продолжения работы ей необходимо дождаться ответа, или если низкоприоритетная задача шлет высокоприоритетной задаче сообщение, которого последняя ожидает, то высокоприоритетная задача, если конечно, используется приоритетная многозадачность с вытеснением, немедленно получит управление. Иногда сообщения передаются через буфер определенного размера (почтовый ящик). При этом, как правило, новое сообщение затирает старое, даже если последнее не было обработано.

Однако наиболее часто используется принцип, когда каждая задача имеет свою очередь сообщений, в конец которой ставится всякое вновь полученное сообщение. Стандартный принцип обработки очереди сообщений по принципу **FIFO** не всегда оптимально соответствует поставленной задаче. В некоторых ОСРВ предусматривается такая возможность, когда сообщение от высокоприоритетной задачи обрабатывается в первую очередь (в этом случае говорят, что сообщение наследует приоритет пославшей его задачи).

Общие ресурсы.

Ресурс - это общий термин, описывающий объект (физическое устройство, область памяти), который может одновременно использоваться только одной задачей.

По своим характеристикам ресурсы разделяют на:

- **активные** – способны изменить информацию (процессор)
- **пассивные** – способны хранить информацию
- **локальные** – принадлежат одному процессу; время жизни совпадает с временем жизни процесса
- **разделяемые** – могут быть использованы несколькими процессами; существуют, пока есть хоть один процесс, который их использует
- **постоянные** – используются посредством операций «захватить» и «освободить»
- **временные** – используются посредством «создать» и «удалить».

Разделяемые ресурсы бывают:

- **не критичные** – могут быть использованы одновременно несколькими процессами (например, жесткий диск)
- **критичные** – могут быть использованы только одним процессом, и пока этот процесс не завершит работу с ресурсом, последний не доступен другим процессам (например, разделяемая память, доступная на запись).

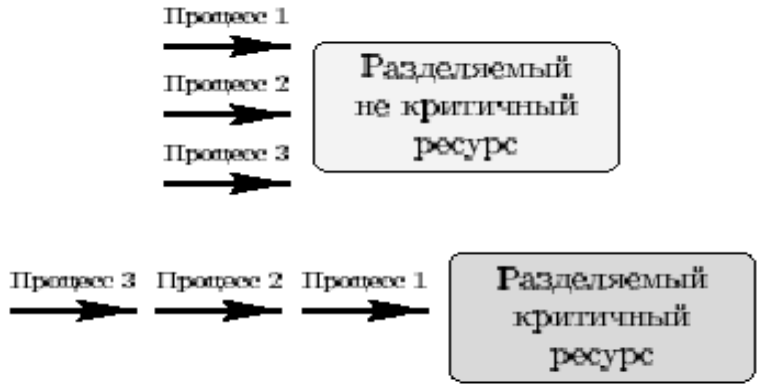


Рис. 1. Виды разделяемых ресурсов

По типу взаимодействия различают:

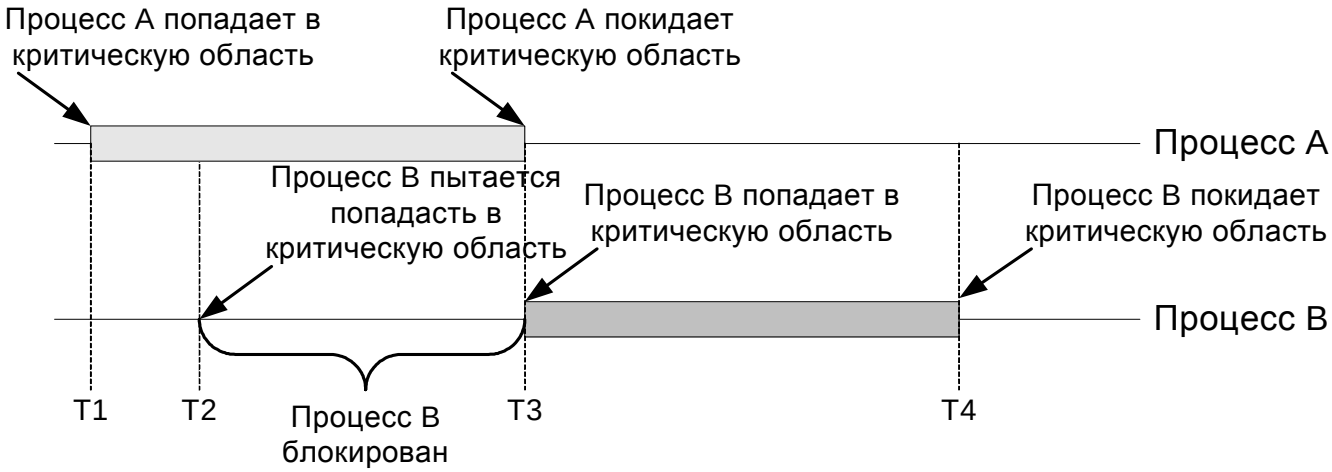
- **сотрудничающие процессы**:
  - процессы, разделяющие только коммуникационный канал, по которому один передает данные, а другой получает их;
  - процессы, осуществляющие взаимную синхронизацию: когда работает один, другой ждет окончания его работы (типично для программ, управляющих рядом технологических процессов)
- **конкурирующие процессы**:
  - процессы, использующие совместно разделяемый ресурс;
  - процессы, использующие критические секции;
  - процессы, использующие взаимные исключения.

Представим, например, что несколько процессов пытаются одновременно выводить данные на принтер. Если процессу требуется вывести на печать файл, он помещает его имя в специальный каталог спулера. Другой процесс, демон печати, периодически проверяет наличие файлов, которые нужно печатать, печатает файл и удаляет его имя из каталога. Пусть каталог спулера состоит из большого числа пронумерованных сегментов, в каждом из которых может храниться имя файла. Также есть две совместно используемые (и доступные всем процессам) переменные – *out*, указывающая на следующий файл для печати, и *in*, указывающая на следующий свободный сегмент. Процессы *A* и *B* одновременно пытаются поставить файлы в очередь на печать. Процесс *A* считывает значение переменной in и сохраняет его в локальной переменной. После этого происходит прерывание по таймеру, и процессор переключается на процесс *B*. Он, в свою очередь, также считывает значение переменной *in* и сохраняет его в своей локальной переменной. Процесс *B* сохраняет в каталоге спулера имя файла и увеличивает на 1 значение переменной *in*. После того как управление перейдет к процессу *A*, он продолжит выполнение с того места, где был прерван. Он обращается к своей локальной переменной, хранящей старое значение переменной *in*, считывает ее значение и записывает имя файла в тот же сегмент, что и процесс *B*, а затем изменяет значение *in*. Структура каталога не нарушена, но файл процесса *B* не будет напечатан. Ситуация, в которой два процесса считывают или записывают данные одновременно и конечный результат зависит от того, какой процесс был первым, называется **состоянием состязания**. Основным способом предотвращения состояния состязания является запрет использования совместно используемых данных одновременно нескольким процессам.

Критическая секция (области) – это участок программы, в котором есть обращение к совместно используемым данным. На этом участке запрещается переключение задач для обеспечения исключительного использования ресурсов процессом. Все ОСРВ предоставляют вызовы «войти в критическую секцию» и «выйти из критической секции».

Взаимное исключение (mutual exclusion, mutex) – это способ синхронизации параллельно работающих процессов, использующих разделяемый постоянный критичный ресурс. Если ресурс занят, то системный вызов «захватить ресурс» переводит процесс из состояния выполнения в состояние ожидания. Когда ресурс будет освобожден посредством системного вызова «освободить ресурс», то этот процесс вернется в состояние выполнения и продолжит работу. Ресурс при этом перейдет в состояние «занят».

В схематичном виде поведение процессов может быть представлено следующим образом. Процесс *A* попадает в критическую область в момент времени *T1*. В момент времени *T2* процесс *B* пытается попасть в критическую область, но ему это не удается, поскольку в критической области уже находится процесс *A*, а два процесса не должны одновременно находиться в критических областях. Поэтому процесс *B* приостанавливается до наступления момента времени *T3*, когда процесс *A* выходит из критической области. В момент времени *T4* процесс *B* покидает критическую область.



При взаимодействии процессов возможны следующие проблемы:

Смертельный захват, тупик (Deadlock). Обычно побочные эффекты этой ситуации называются более прозаично – «зацикливание» или «зависание». А причина этого может быть достаточно проста – «задачи не поделили ресурсы». Пусть, например, *Задача A* захватила ресурс клавиатуры и ждет, когда освободится ресурс дисплея, а в это время *Задача B*, успев захватить ресурс дисплея, ждет теперь, когда освободится клавиатура. В таких случаях рекомендуется придерживаться тактики «или все, или ничего». Другими словами, если задача не смогла получить все необходимые для дальнейшей работы ресурсы, она должна освободить все, что уже захвачено, и повторить попытку только через определенное время. Другим решением, которое уже упоминалось, является использование серверов ресурсов.

Инверсия приоритетов (Priority inversion). Представим, что у нас есть высокоприоритетная *Задача A*, среднеприоритетная *Задача B* и низкоприоритетная *Задача C*. Пусть в начальный момент времени *Задачи A* и *B* блокированы в ожидании какого-либо внешнего события. Допустим, получившая в результате этого управление *Задача C* захватила *Ресурс A*, но не успела его отдать, как была прервана *Задачей A*. В свою очередь, *Задача A* при попытке захватить *Ресурс A* будет блокирована, так как этот ресурс уже захвачен *Задачей C*. Если к этому времени *Задача B* находится в состоянии готовности, то управление после этого получит именно она, как имеющая более высокий, чем у *Задачи C*, приоритет. Теперь *Задача B* может занимать процессорное время, пока ей не надоест, а мы получаем ситуацию, когда высокоприоритетная *Задача A* не может функционировать из-за того, что необходимый ей ресурс занят низкоприоритетной *Задачей C*.

Блокировка (Lockout). Процесс ожидает ресурс, который никогда не освободится.

Голодовка (Starvation). Процесс монополизировал процессор.

Взаимоблокировки. Выгружаемые и невыгружаемые ресурсы. Условия возникновения взаимоблокировок. Граф ресурсов. Стратегии решения проблемы взаимоблокировок. Обнаружение тупиков и восстановление работоспособности системы.

В компьютерной системе существует большое количество ресурсов, каждый из которых может использоваться в конкретный момент времени только одним процессом. Часто процесс для выполнения задачи нуждается в исключительном доступе не к одному, а к нескольким ресурсам. Предположим, что имеется два процесса *A* и *B*. Процесс *A* имеет в исключительном доступе ресурс *R1*, процесс *B* – ресурс *R2*. Если при этом процессу *A* для продолжения работы потребуется ресурс *R2*, то его запрос на получение ресурса будет отклонен до тех пор, пока

ресурс не будет освобожден процессом *B*. Если же при этом процессу *B* для продолжения работы окажется необходимым иметь в исключительном доступе ресурс *R1*, то и процесс *B* окажется, как и процесс *A*, заблокированным. Такая ситуация называется **тупиком**, **тупиковой ситуацией** или **взаимоблокировкой**.

Система может зайти в тупик, когда процессам предоставляются исключительные права доступа к ресурсам. С рассматриваемой здесь точки зрения ресурсы могут быть разделены на два типа: выгружаемые и невыгружаемые. **Выгружаемый ресурс** можно безболезненно забрать у владеющего им процесса. Образцом такого ресурса является память. **Невыгружаемый ресурс**, в противоположность выгружаемому, - это такой ресурс, который нельзя забрать у текущего процесса без уничтожения результатов вычислений. Если в момент записи компакт-диска внезапно забрать у процесса устройство для записи и передать его другому процессу, в результате получим испорченный компакт-диск.

Вообще говоря, взаимоблокировки касаются невыгружаемых ресурсов. Потенциальные тупиковые ситуации, в которые вовлечены выгружаемые ресурсы, обычно можно разрешить с помощью перераспределения ресурсов от одного процесса другому.

Взаимоблокировки или тупиковые ситуации формально можно определить так:

Группа процессов находится в тупиковой ситуации, если каждый процесс из группы ожидает события, которое может вызвать только другой процесс из той же группы.

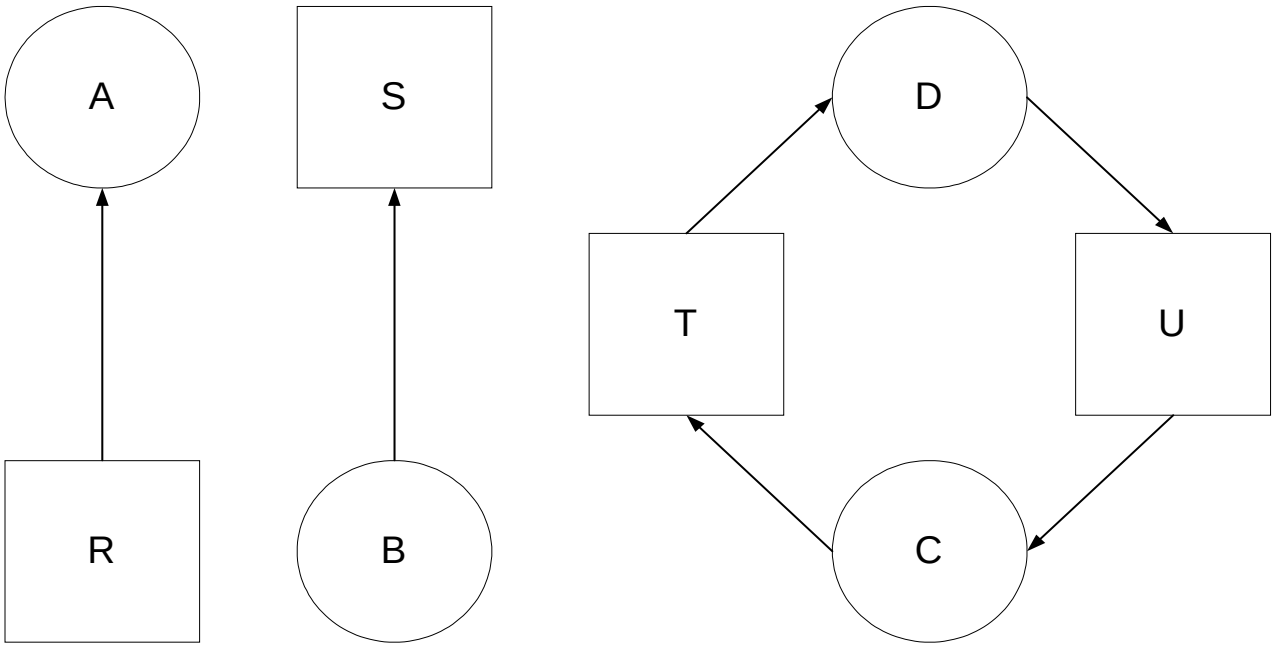
Так как все процессы находятся в состоянии ожидания при возникновении блокировки, то ни один из них не будет причиной какого-либо события, которое могло бы активировать любой другой процесс в группе, и все процессы продолжат ждать до бесконечности. В этой модели предполагается наличие только одного потока у каждого процесса и отсутствие прерываний, способных активизировать заблокированный процесс. Условие отсутствия прерываний необходимо, чтобы предотвратить ситуацию, когда тот или иной заблокированный процесс активизируется, скажем, по сигналу тревоги и затем приводит к событию, которое освободит другие процессы в группе. В большинстве случаев событием, которого ждет каждый процесс, является возврат какого-либо ресурса, в данный момент занятого другим участником группы.

Исследования показали, что для возникновения ситуации взаимоблокировки должны выполняться 4 условия:

1. **Условие взаимного исключения.** Каждый ресурс в данный момент времени или отдан одному процессу, или доступен.
2. **Условие удержания и ожидания.** Процессы, в данный момент удерживающие полученные ранее ресурсы, могут запрашивать новые ресурсы.
3. **Условие отсутствия принудительной выгрузки ресурса.** У процесса нельзя принудительным образом забрать ранее полученные ресурсы. Процесс, владеющий ими, должен сам освободить ресурсы.
4. **Условие циклического ожидания.** Должна существовать круговая последовательность из двух или более процессов, каждый из которых ждет доступа к ресурсу, удерживаемому следующим членом последовательности.

Для того, чтобы произошла взаимоблокировка, должны выполниться все четыре условия. Если хоть одно условие отсутствует, тупиковая ситуация невозможна.

Условия возникновения тупиков можно смоделировать, используя направленные графы. Графы имеют два вида узлов: процессы, показанные кружочками, и ресурсы, изображающиеся квадратиками. Ребро, направленное от узла ресурса к узлу процесса (от квадрата к кругу), означает, что ресурс ранее был запрошен процессом, получен и в данный момент используется данным процессом. Ребро, направленное от процесса к ресурсу (от круга к квадрату), означает, что процесс в данный момент блокирован и находится в состоянии ожидания доступа к этому ресурсу.



На рисунке выше показаны следующие ситуации. Ресурс *R* в настоящий момент времени отдан процессу *A*. Процесс *B* ждет ресурс *S*. Процессы *C* и *D* находятся в состоянии взаимоблокировки: процесс *C* ожидает ресурс *T*, удерживаемый в настоящий момент процессом *D*. Процесс *D* не освобождает ресурс *T*, так как *B* ждет ресурс *U*, используемый процессом *C*. Оба процесса могут ждать до бесконечности.

Вообще говоря, при столкновении с взаимоблокировками используются четыре стратегии:

1. Пренебрежение проблемой в целом. Если вы проигнорируете проблему, возможно, затем она проигнорирует вас.
2. Обнаружение и восстановление. Позволить взаимоблокировке произойти, обнаружить ее и предпринять какие-либо действия.
3. Динамическое избежание тупиковых ситуаций с помощью аккуратного распределения ресурсов.
4. Предотвращение с помощью структурного опровержения одного из четырех условий, необходимых для взаимоблокировки.

Рассмотрим каждый из этих четырех методов.

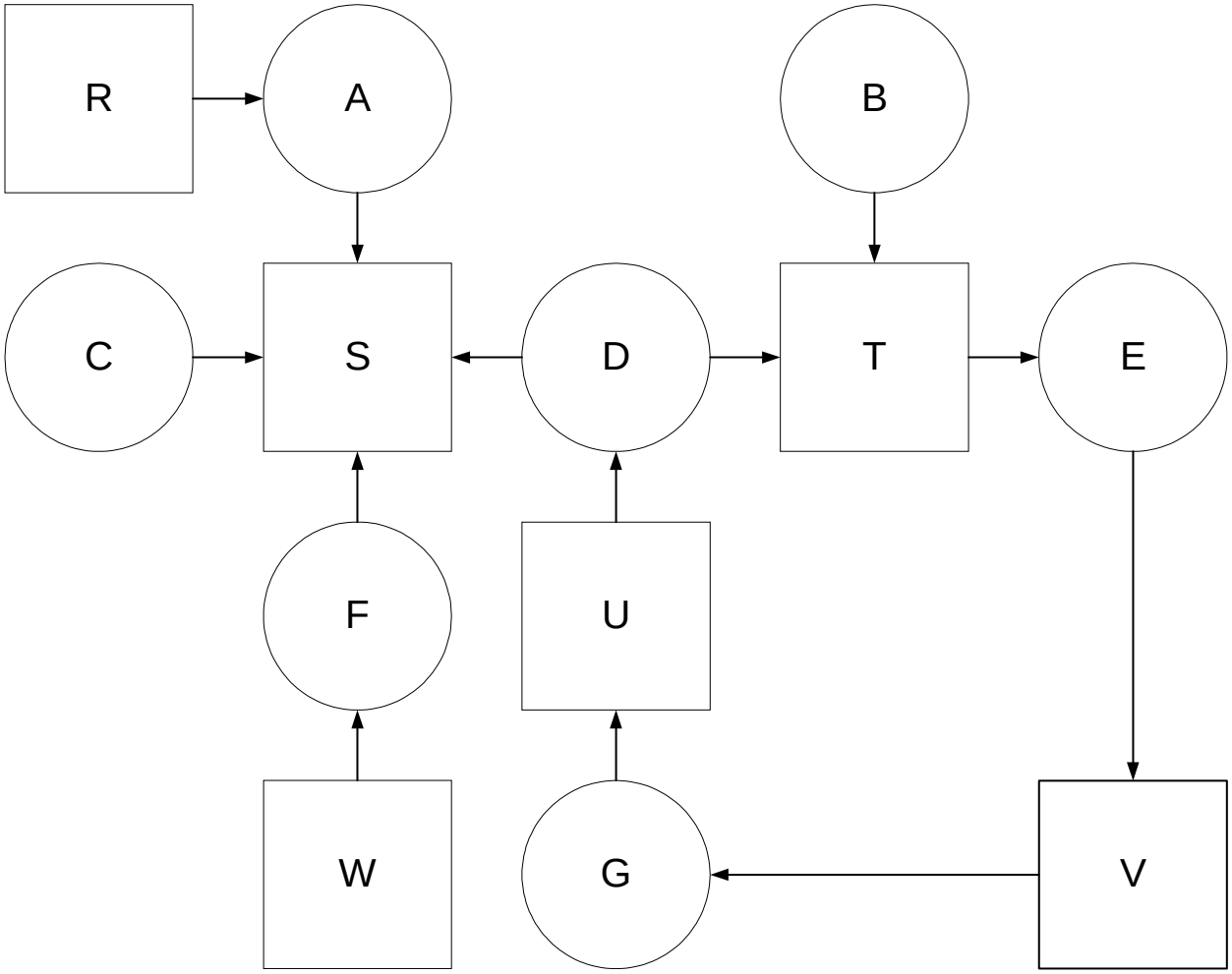
Самым простым подходом является «страусовый алгоритм»- притвориться, что проблема вообще не существует. Большая часть операционных систем, включая UNIX и Windows, игнорируют эту проблему. Они исходят из предположения, что большинство пользователей предпочтут иметь дело со случающимися время от времени взаимоблокировками, чем с правилом, по которому всем пользователям разрешается только один процесс, один открытый файл и т.д.

Вторая техника представляет собой обнаружение и восстановление. При использовании этого метода система не пытается предотвратить попадание в тупиковые ситуации. Вместо этого она позволяет взаимоблокировке произойти, старается определить, когда это случилось, и затем совершает некие действия к возврату системы к состоянию, имевшему место до того, как система попала в тупик. Рассмотрим эту технику на примере простого варианта: в системе существует только один ресурс каждого типа. Для такой системы можно сконструировать граф ресурсов. Если граф ресурсов содержит один или более циклов, то произошла взаимоблокировка и блокирован любой процесс, являющийся частью цикла. Если в графе нет циклов, система не попала в тупик.

В качестве примера рассмотрим систему с семью процессами, обозначенными буквами от *A* до *G*, и шестью ресурсами, обозначенными буквами от *R* до *W*. Состояние системы в данный момент времени соответствует следующему списку:

- Процесс *A* занимает ресурс *R* и хочет получить ресурс *S*.
- Процесс *B* ничего не использует, но хочет получить ресурс *T*.
- Процесс *C* ничего не использует, но хочет получить ресурс *S*.
- Процесс *D* занимает ресурс *U* и хочет получить ресурсы *S* и *T*.
- Процесс *E* занимает ресурс *T* и хочет получить ресурс *V*.

- Процесс *F* занимает ресурс *W* и хочет получить ресурс *S*.
- Процесс *G* занимает ресурс *V* и хочет получить ресурс *U*.



Из построенного графа ресурсов видно, что он содержит цикл и процессы *D,E,G* заблокированы. Процессы *A,C,F* не попали в тупик, потому что любому из них можно предоставить ресурс *S*, после чего процесс, получивший ресурс, закончит работу и вернет ресурс. Затем два других процесса по очереди также могут получить ресурс и успешно выполнить свою работу.

Обнаружив тупик, необходимо ликвидировать взаимоблокировку. Рассмотрим методы восстановления работоспособности системы.

Восстановление при помощи принудительной выгрузи ресурсов. Иногда можно отобрать ресурс у его текущего владельца и отдать его другому процессу. Способность забирать ресурс у процесса, отдавать его другому процессу и затем возвращать назад так, чтобы исходный процесс этого не замечает, в значительной мере зависит от свойств ресурса. Выйти из тупика таким образом зачастую трудно или невозможно. Выбор приостанавливаемого процесса главным образом зависит от того, какой процесс владеет ресурсами, которые легко могут быть у него отняты. Во многих случаях требуется ручное вмешательство.

Восстановление через откат. Если разработчики системы знают, что есть вероятность взаимоблокировки, они могут организовать работу таким образом, чтобы процессы периодически создавали **контрольные точки**. Созданные процессом контрольные точки означают, что состояние процесса записывается в файл, в результате чего впоследствии процесс может быть возобновлен из этого файла. Контрольные точки содержат не только образ памяти, но и состояние ресурсов, то есть информацию о том, какие ресурсы в данный момент предоставлены процессу. Когда взаимоблокировка обнаружена, достаточно понять, какие ресурсы нужны процессам. Чтобы выйти из тупика, процесс откатывается к тому времени, перед которым он получил данный ресурс, для чего запускается одна из его контрольных точек. Вся работа, выполненная после этой контрольной точки, теряется. В результате процесс вновь запускается с более раннего момента, когда он не занимал тот ресурс, который теперь

предоставляется одному из процессов, попавших в тупик. Если возобновленный процесс вновь пытается получить данный ресурс, ему придется ждать того момента, когда ресурс опять станет доступен.

Восстановление путем уничтожения процессов. Простейший способ выхода из тупика заключается в уничтожении одного или нескольких процессов. Легче всего уничтожать те процессы, которые можно запустить сначала без всяких болезненных эффектов. При этом могут быть уничтожены как процессы, входящие в цикл взаимоблокировки, так и процессы, не находящиеся в цикле, но владеющие ресурсами, необходимыми заблокированным процессам.

Избежание взаимоблокировок. Предотвращение взаимоблокировок.

В большинстве систем процессы запрашиваются не все сразу, а поочередно. Система должна уметь решать, является ли предоставление ресурса безопасным или нет, и предоставить его процессу только в первом случае. Таким образом, возникает вопрос: существует ли алгоритм, который всегда может избежать ситуации взаимоблокировки, все время делая правильный выбор? Ответом будет условие «да» - мы можем избежать тупиков, но если заранее будет доступна определенная информация.

Основные алгоритмы, позволяющие предотвращать взаимоблокировки, базируются на концепции безопасных состояний. Говорят, что состояние **безопасно**, если оно не находится в тупике и существует некоторый порядок планирования, при котором каждый процесс может работать до завершения, даже если все процессы вдруг захотят немедленно получить максимальное количество ресурсов.

Проиллюстрируем сказанное на примере с одним ресурсом. Пусть в некотором состоянии процесс *A* занимает 3 экземпляра ресурса, но может потребовать 9 экземпляров. Процесс *B* в настоящий момент занял 2 экземпляра, но может потребовать еще 4. Процесс *C* владеет 2 экземплярами, и ему могут понадобиться еще 5. В системе есть всего 10 экземпляров ресурса, 7 из которых в настоящий момент распределены, а 3 остаются свободными. Текущее состояние системы показано как *«состояние а»*.

	Имеет	Max
A	3	9
B	2	4
C	2	7
Свободно 3		
Состояние а		

	Имеет	Max
A	3	9
B	4	4
C	2	7
Свободно 1		
Состояние б		

	Имеет	Max
A	3	9
B	0	-
C	2	7
Свободно 5		
Состояние в		

	Имеет	Max
A	3	9
B	0	-
C	7	7
Свободно 0		
Состояние г		

	Имеет	Max
A	3	9
B	0	-
C	0	-
Свободно 7		
Состояние д		

«Состояние а» безопасно, потому что существует такая последовательность предоставления ресурсов, которая позволяет завершиться всем процессам. Например, планировщик может просто запустить в работу процесс *B* на то время, пока он запросит и получит два дополнительных экземпляра ресурса, что приведет систему к *«состоянию б»*. Когда процесс *B* закончится, мы получим *«состояние в»*. Затем планировщик может запустить процесс *C*, что приведет нас к *«состоянию г»*. По завершении процесса *C* мы получим *«состояние д»*. В этом состоянии процесс *A* может занять все необходимые ему ресурсы и также завершиться. Таким образом, *«состояние а»* является безопасным.

	Имеет	Max
A	3	9
B	2	4
C	2	7
Свободно 3		
Состояние а		

	Имеет	Max
A	4	9
B	2	4
C	2	7
Свободно 2		
Состояние б		

	Имеет	Max
A	4	9
B	4	4
C	2	7
Свободно 0		
Состояние в		

	Имеет	Max
A	4	9
B	0	-
C	2	7
Свободно 4		
Состояние г		

Теперь предположим, что система первоначально находится в том же состоянии, но в данный момент процесс *A* запрашивает и получает еще один ресурс. Это приводит на с *«состоянию б»*. В этом состоянии планировщик уже не сможет найти последовательность,

которая гарантировала бы работу системы. Можно дать проработать процессу *B* до того момента, пока он не запросит свои ресурсы (*«состояние в»*). В итоге процесс *B* успешно завершится и система окажется в *«состоянии г»*. В этом месте система застревает: в системе осталось всего 4 свободных экземпляра ресурса, а каждому из активных процессов необходимо 5 экземпляров. Следовательно, решение о предоставлении ресурса, приводящее систему в *«состояние б»*, привело ее в небезопасное состояние. Если из этого состояния запустить процесс *A* или *C*, система не выйдет из тупика.

Следует отметить, что небезопасное состояние само по себе не является тупиком. Система еще может проработать какое-то время (в данном случае – может закончиться процесс *B*). Кроме того, возможна ситуация, что процесс *A* сможет освободить ресурс до следующего запроса, позволяя процессу *C* успешно завершиться, а системе – избежать взаимоблокировки. Таким образом, разница между безопасным и небезопасным состоянием заключается в том, что в безопасном состоянии система может гарантировать, что все процессы закончат свою работу, а в небезопасном состоянии такую гарантию дать нельзя.

Алгоритм планирования, позволяющий избегать взаимоблокировок, был разработан Дейкстрой и носит название **алгоритма банкира**. Он представляет собой расширение алгоритма обнаружения тупиков. Модель алгоритма основана на примере банкира, имеющего дело с группой клиентов, которым он выдал ряд кредитов. Алгоритм проверяет, ведет ли выполнение каждого запроса к небезопасному состоянию. Если да, запрос отклоняется. Если удовлетворение запроса приводит к безопасному состоянию, ресурс предоставляется процессу. Чтобы понять, является ли состояние безопасным, банкир проверяет, может ли он предоставить достаточно ресурсов для завершения работы какого-либо клиента. Если да, то эти ссуды считается погашенными, после чего проверяется ближайший к пределу займа клиент и т.д. Если, в конце концов, все ссуды могут быть погашены, состояние является безопасным и исходный запрос можно удовлетворить. На практике применение алгоритма банкира затруднено, потому что нечасто можно определить заранее, сколько ресурсов потребуется процессам в будущем. Кроме того, количество процессов не фиксировано и динамически меняется по мере работы. Более того, ресурсы, про которые считалось, что они доступны, внезапно могут исчезнуть.

Как видно, уклонение от взаимоблокировок, в сущности, невозможно. Тогда возникает вопрос, как реальные системы могут избегать тупиков. Для этого следует вспомнить, что взаимоблокировки невозможны, если не выполняется хотя бы одно из условий взаимоблокировки.

Условие взаимного исключения. Если в системе нет ресурсов, отданных в единоличное пользование одному процессу, система никогда не попадет в тупик. На практике невозможно отказаться от этого.

Условие удержания и ожидания. Если мы сможем уберечь процессы, занимающие некоторые ресурсы, от ожидания остальных ресурсов, мы устраним ситуацию взаимоблокировки. Один из способов достижения этой цели состоит в требовании, следуя которому любой процесс должен запрашивать все ресурсы до начала работы. Если ресурсы свободны, процесс получит все ему необходимое и сможет работать до успешного завершения. Если один или несколько ресурсов заняты, процессу ничего не представляется.

Первая проблема при этом подходе заключается в том, что многие процессы не знают, сколько ресурсов им понадобится, до тех пор, пока не начнут работу. Другая проблема – неоптимальное распределение ресурсов.

Условие отсутствия принудительной выгрузки ресурса. Устранение этого условия также практически невозможно. Например, если процесс получил принтер и печатает входные данные, насильственное изъятие принтера по причине недоступности какого-либо другого устройства в лучшем случае сложно, а в худшем – невозможно.

Условие циклического ожидания. Циклическое ожидание можно устранить несколькими способами. Один из них следующий: процессу дано право только на один ресурс в данный момент времени. Если нужен второй ресурс, процесс должен освободить первый.

Другой способ уклонения от циклического ожидания заключается в поддержке общей нумерации всех ресурсов. Тогда действует общее правило: процессы могут запрашивать ресурс когда хотят этого, но все запросы должны быть сделаны в соответствии с нумерацией ресурсов.

При выполнении такого рода правила граф ресурсов никогда не будет иметь циклов. Покажем это на примере двух процессов.

Пусть процесс *A* имеет в монопольном использовании ресурс с номером *I*, а процесс *B* – ресурс с номером *J*. Мы можем попасть в тупик, если процесс *A* запросит ресурс *J*, а процесс *B* – ресурс *I*. Предположим, что ресурсы *I* и *J* различны. Это значит, что они имеют разные номера. Если *I* больше *J*, то процессу *A* не позволяется запрашивать ресурс *J*, потому что его номер меньше, чем номер имеющегося ресурса. Если *J* меньше *I*, отклоняется запрос процесса *B* по той же причине. Так или иначе, блокировка невозможна.

При работе с несколькими процессами сохраняется та же самая логика. В каждый момент времени один из предоставленных ресурсов будет иметь наивысший номер. Процесс, использующий этот ресурс, уже никогда не запросит другие занятые ресурсы. Он или закончит свою работу, или, в худшем случае, запросит ресурс с еще большим номером. Любой такой ресурс окажется доступным. В итоге процесс завершит свою работу и освободит ресурсы. На этот момент сложится ситуация, когда ресурс с высшим номером уже занят каким-то другим процессом, который также сможет закончить работу и т.д. Таким образом, все процессы завершатся без тупика.

Вариантом этого алгоритма является схема, в которой отбрасывается требование приобретения ресурсов в строго возрастающем порядке, но сохраняется условие, что процесс не может запросить ресурсы с меньшим номером, чем уже у него имеющиеся.

Несмотря на то, что систематизация ресурсов с помощью их нумерации устраняет проблему взаимоблокировок, бывает ситуации, когда невозможно найти порядок, удовлетворяющий всех. Когда ресурсы включают в себя области таблицы процессов, дисковое пространство для подкачки данных, записи базы данных и т.д., число потенциальных ресурсов может быть настолько огромным, что никакая систематизация не сможет работать.

Тупики без ресурсов. Взаимоблокировки могут происходить и без какого-либо участия ресурсов. Например, два процесса могут заблокировать друг друга, если каждый ждет, когда другой выполнит определенные действия. Такое часто случается, когда операции над объектами синхронизации (например, операции над семафорами) были выполнены в неправильном порядке.