Управление процессами

Понятие процесса

Для простоты возможно рассматривать **процесс** как **абстракцию, характеризующую программу во время выполнения**.

Понятие **процесса** характеризует:

- некоторую совокупность набора исполняющихся команд;
- связанных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.)
- текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы.

Для операционной системы процесс представляет собой единицу работы, **заявку на потребление системных ресурсов**.

Состояние процессов

В многозадачной (многопроцессной) системе процесс может находиться в одном из трех основных состояний:

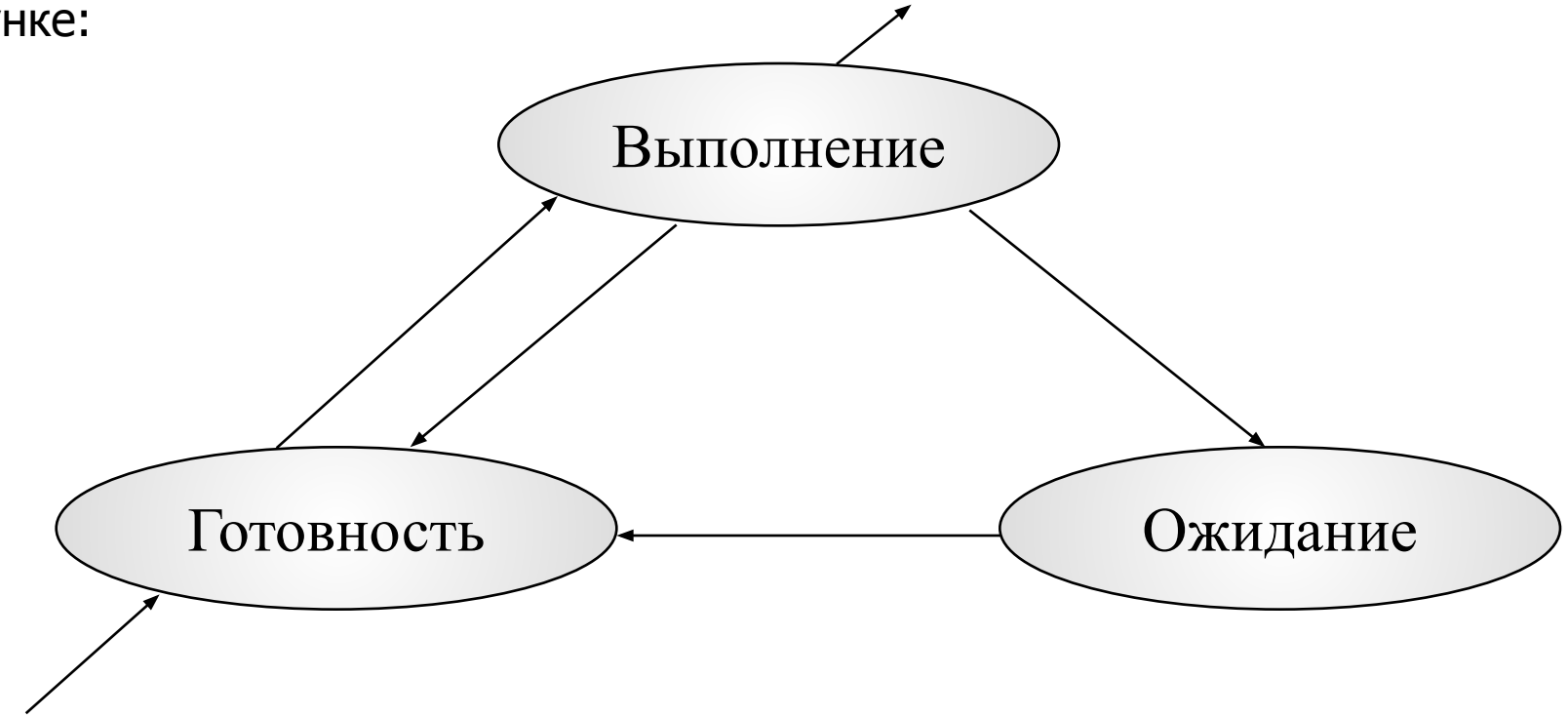
ВЫПОЛНЕНИЕ - **активное состояние** процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;

ОЖИДАНИЕ - **пассивное состояние** процесса, процесс заблокирован, он не может выполняться **по своим внутренним причинам** (он ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого процесса, освобождения какого-либо необходимого ему ресурса);

ГОТОВНОСТЬ - также **пассивное состояние** процесса, но в этом случае процесс заблокирован в связи с внешними по отношению к нему обстоятельствами: процесс имеет все требуемые для него ресурсы, он готов выполняться, однако процессор занят выполнением другого процесса.

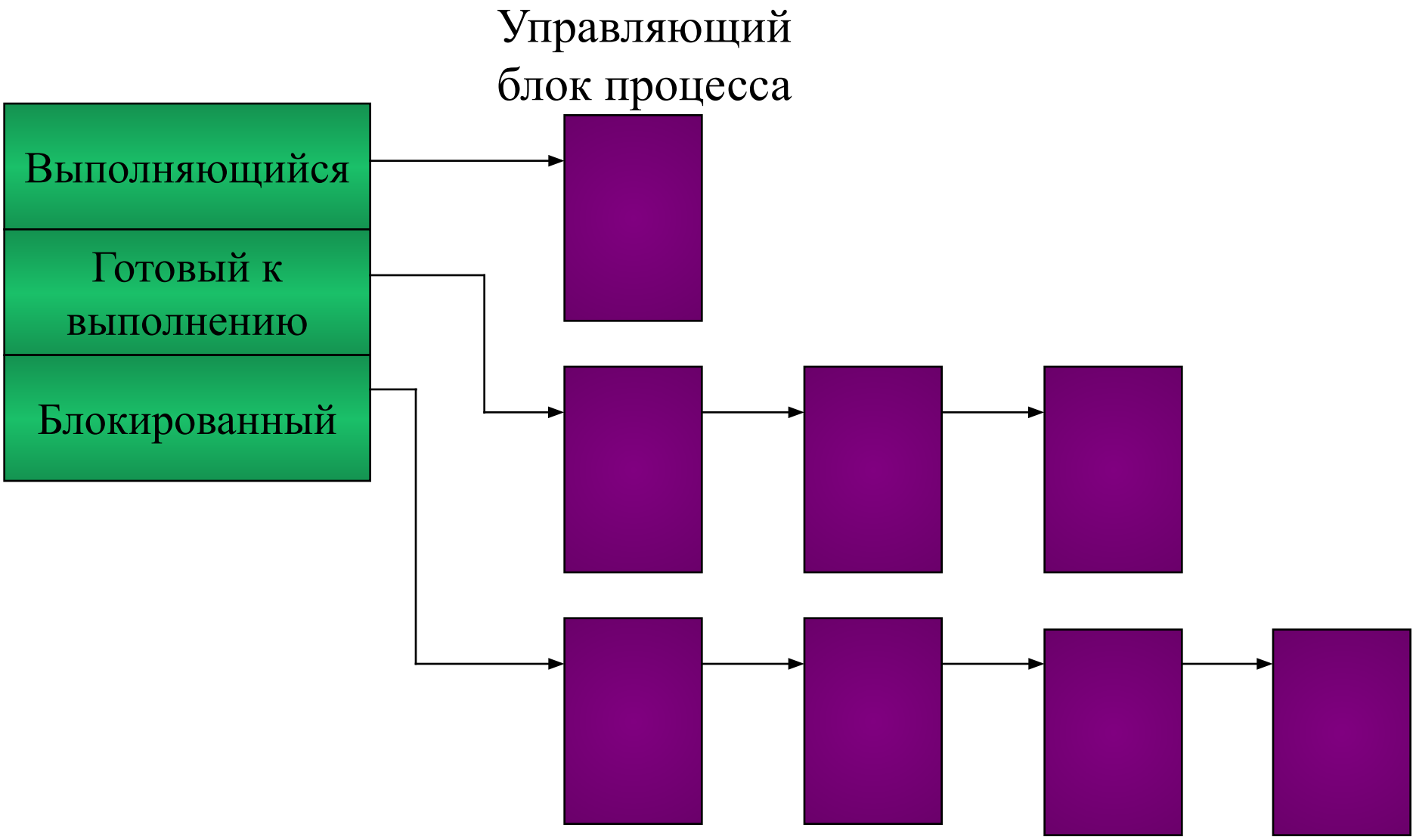
Управление процессами

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе. Типичный граф состояний процесса показан на рисунке:

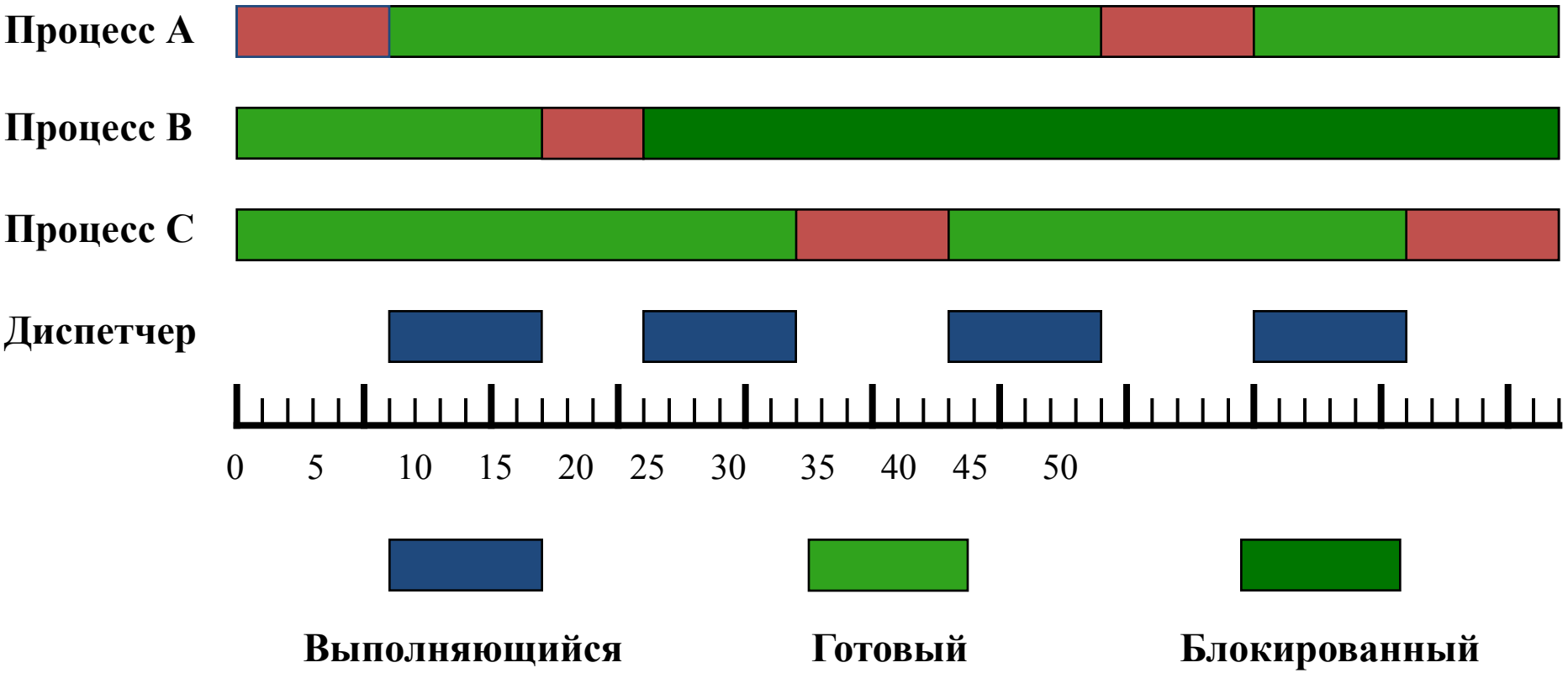


В состоянии ВЫПОЛНЕНИЕ в однопроцессорной системе может находиться только один процесс, а в каждом из состояний ОЖИДАНИЕ и ГОТОВНОСТЬ - несколько процессов, эти процессы образуют **очереди соответственно ожидающих и готовых процессов**.

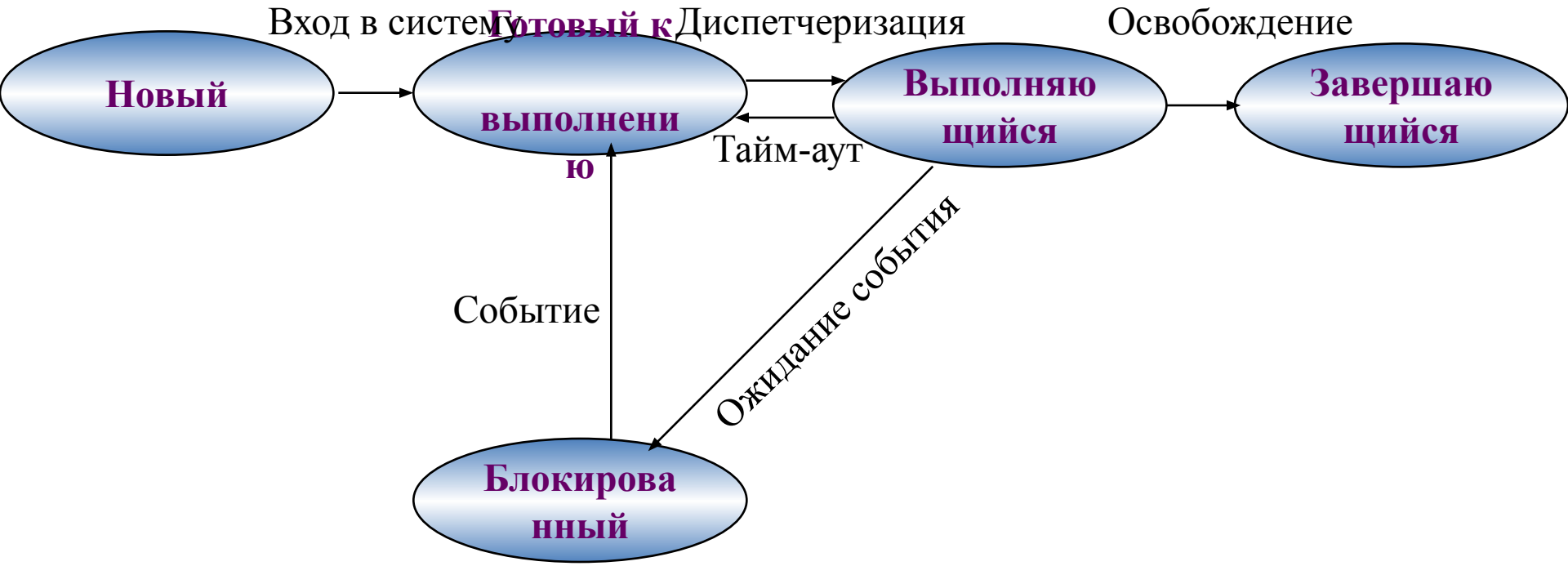
Управление процессами



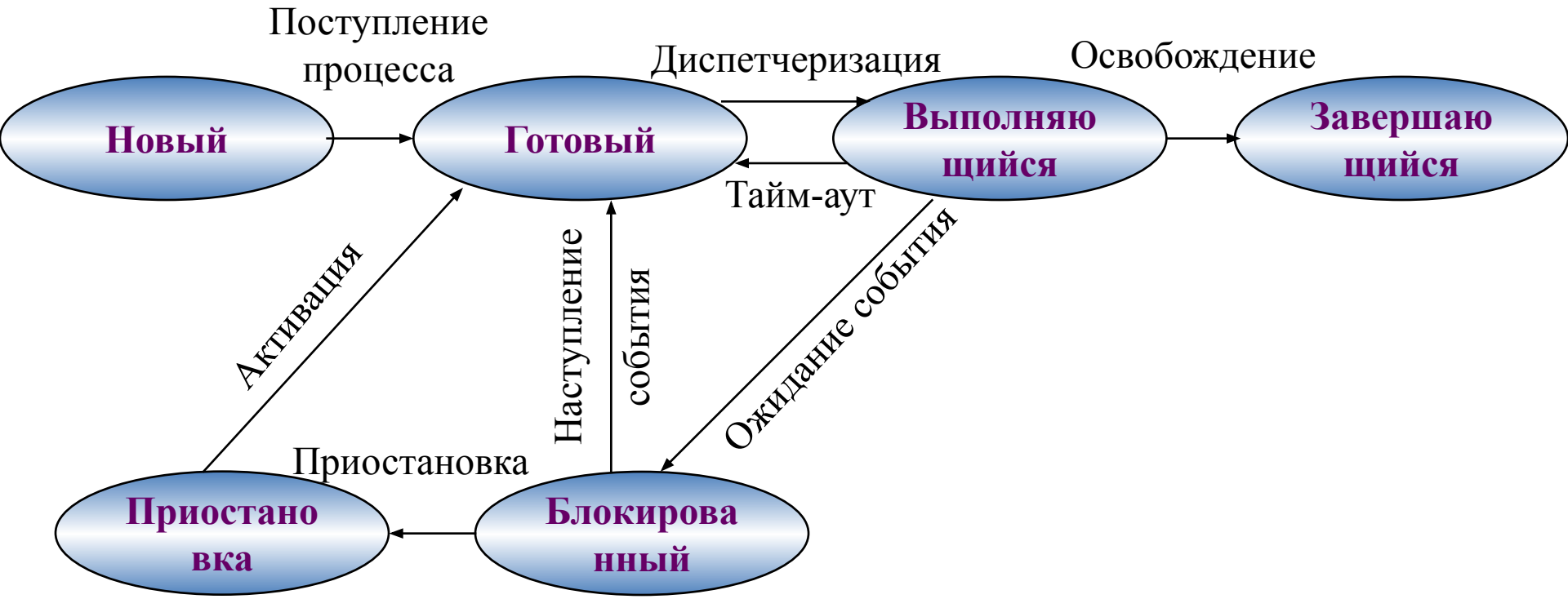
Состояния процессов



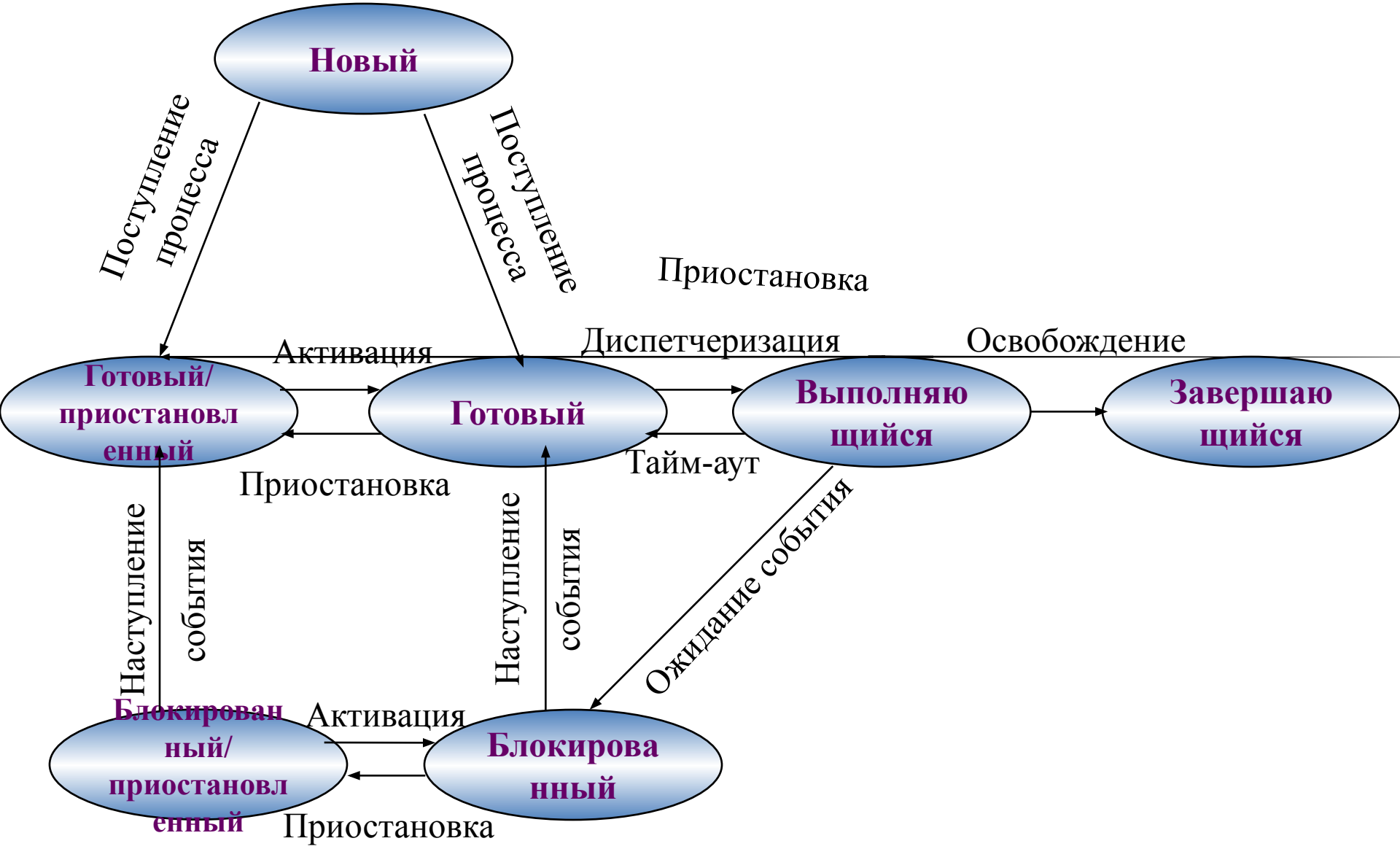
Модель с пятью состояниями



Модель с шестью состояниями



Модель с двумя приостановленными состояниями



Управляющий блок и контекст процесса

Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных. Эта структура содержит информацию, специфическую для данного процесса:

- **состояние**, в котором находится процесс;
- **программный счетчик процесса** или, другими словами, адрес команды, которая должна быть выполнена для него следующей;
- **содержимое регистров процессора**;
- **данные, необходимые для планирования** использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т. д.);
- **учетные данные** (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом);
- **сведения об устройствах ввода-вывода, связанных с *процессом*** (например, какие устройства закреплены за процессом, таблицу открытых файлов).

Эта информация находится в **управляющем блоке процесса**. Информацию, которая находится в управляющем блоке процесса, можно разбить на три основные категории:

- информация по идентификации процесса;
- информация по состоянию процесса;
- информация, используемая при управлении процессом.

Управление процессами

Управляющий блок процесса - это самая важная структура данных из всех имеющихся в операционной системе.

В управляющий блок каждого процесса входит вся необходимая операционной системе информация о нем. Информация в этих блоках считывается и/или модифицируется почти каждым модулем операционной системы, включая те, которые связаны с планированием, распределением ресурсов, обработкой прерываний, а также осуществлением контроля и анализа.

Можно сказать, что **состояние операционной системы задается совокупностью управляющих блоков процессов**.

Управление процессами

Содержимое всех регистров процессора (включая значение программного счетчика) иногда называют **регистровым контекстом процесса**, а все остальное – **системным контекстом процесса**. Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его работой в операционной системе, совершая над ним операции.

С точки зрения пользователя, наибольший интерес представляет содержимое адресного пространства процесса, возможно, наряду с регистровым контекстом определяющее последовательность преобразования данных и полученные результаты. **Код и данные, находящиеся в адресном пространстве процесса** иногда называют его **пользовательским контекстом**.

Совокупность регистрового, системного и пользовательского контекстов процесса для краткости принято называть просто **контекстом процесса**. В любой момент времени процесс полностью характеризуется своим контекстом.

Операции над процессами

Создание процессов

Операционная система может принять решение создать процесс, в следующих случаях:

- при инициализации системы;
- при запуске пользователем прикладного процесса;
- при запуске ОС служебной или системной задачи.

При создании нового процесса ОС должна:

1. **Присвоить новому процессу уникальный идентификатор.**
2. **Выделить пространство для процесса.**
3. **Инициализировать управляющий блок процесса.**
4. **Установить необходимые связи.**
5. **Создать или расширить другие структуры данных.**

Переключение процессов

Когда нужно переключать процессы

Переключение процесса может произойти в любой момент, когда управление от выполняющегося процесса переходит к операционной системе. В таблице перечислены возможные причины, по которым управление может перейти к операционной системе.

Механизмы прерывания процесса		
Механизм	**Причина**	**Использует**
Прерывание	Внешняя по отношению к выполнению текущей команды	Отклик на внешнее асинхронное событие
Ловушка	Связана с выполнением текущей команды	Обработку ошибки или исключительной ситуации
Вызов супервизора	Запрос приложения операционной системы	Вызов функции

Фактически имеются системные прерывания двух видов. Первый вид - обычные прерывания, а второй - ловушки (trap).

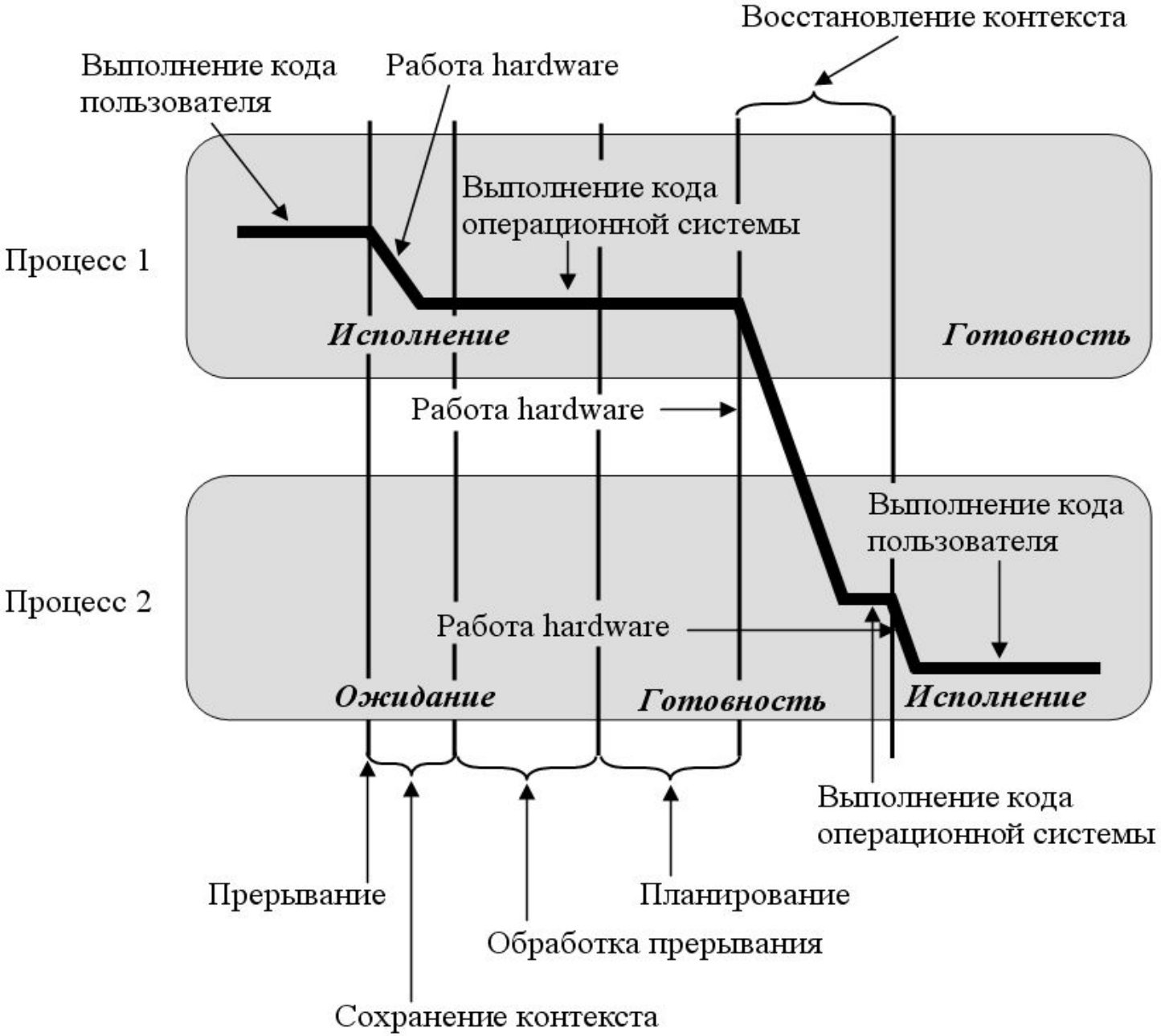
Прерывания первого вида происходят из-за событий определенного типа, **не связанных** с выполняющимся процессом и являющихся внешними по отношению к нему (таким событием может быть, например, завершение операции ввода-вывода). **Ловушки связаны с ошибкой или исключительной ситуацией**, возникшей в результате выполнения текущего процесса.

Управление процессами

Операционная система может быть активизирована в результате вызова супервизора (supervisor call), который исходит от выполняемой программы.
В случае **переключения процессов** должны быть выполнены следующие действия:

1. Сохранение контекста процессора, включая содержимое счетчика команд и других регистров.
2. Обновление управляющего блока выполняющегося в данное время процесса. Сюда входит изменение состояния процесса.
3. Помещение управляющего блока данного процесса в соответствующую очередь (очередь готовых к выполнению процессов; процессов, блокированных событием; очередь готовых приостановленных процессов).
4. Выбор следующего процесса для выполнения; это вопрос – **планирования процессов**.
5. Обновление управляющего блока выбранного процесса. Для этого процесса нужно установить состояние выполнения.
6. Обновление структур данных по управлению памятью.
7. Восстановление контекста процессора в исходное состояние(когда выбранный процесс был последний раз переключен из состояния выполнения). Это происходит путем загрузки содержимого программного счетчика и других регистров процессора.

Управление процессами



Уничтожение процессов

Операционная система может принять решение **уничтожить** процесс, в следующих случаях:

- при завершении работы системы;
- при возникновении критической ошибки;
- при завершении работы процесса.

При уничтожении процесса ОС должна:

1. **Исключить процесс из очередей.**
2. **Освободить пространство памяти, занимаемое процессом.**
3. **Уничтожить управляющий блок процесса.**

Процессы и потоки

Многопоточность

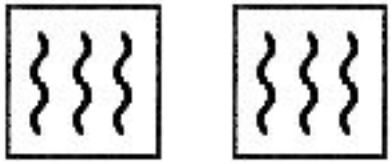
Многопоточностью (multithreading) называется способность операционной системы поддерживать в рамках одного процесса выполнение нескольких потоков.

Традиционный подход, при котором каждый процесс представляет собой единый поток выполнения, называется однопоточным подходом.

Один процесс, один поток

Один процесс, несколько потоков

Несколько процессов, по одному потоку в процессе



Несколько процессов, несколько потоков в процессе

Управление процессами

MS DOS является примером операционной системы, способной поддерживать не более одного однопоточного пользовательского процесса.

Другие операционные системы, такие, как разнообразные **разновидности UNIX**, поддерживают процессы множества пользователей, но в каждом из этих процессов может содержаться только один поток.

Примером системы, в которой один процесс может расщепляться на несколько потоков, является **среда выполнения Java**.

Подход **использования нескольких процессов, каждый из которых поддерживает выполнение нескольких потоков** принят в таких операционных системах, как **OS/2, Windows (NT и выше), Linux, Solaris** и др.

Управление процессами

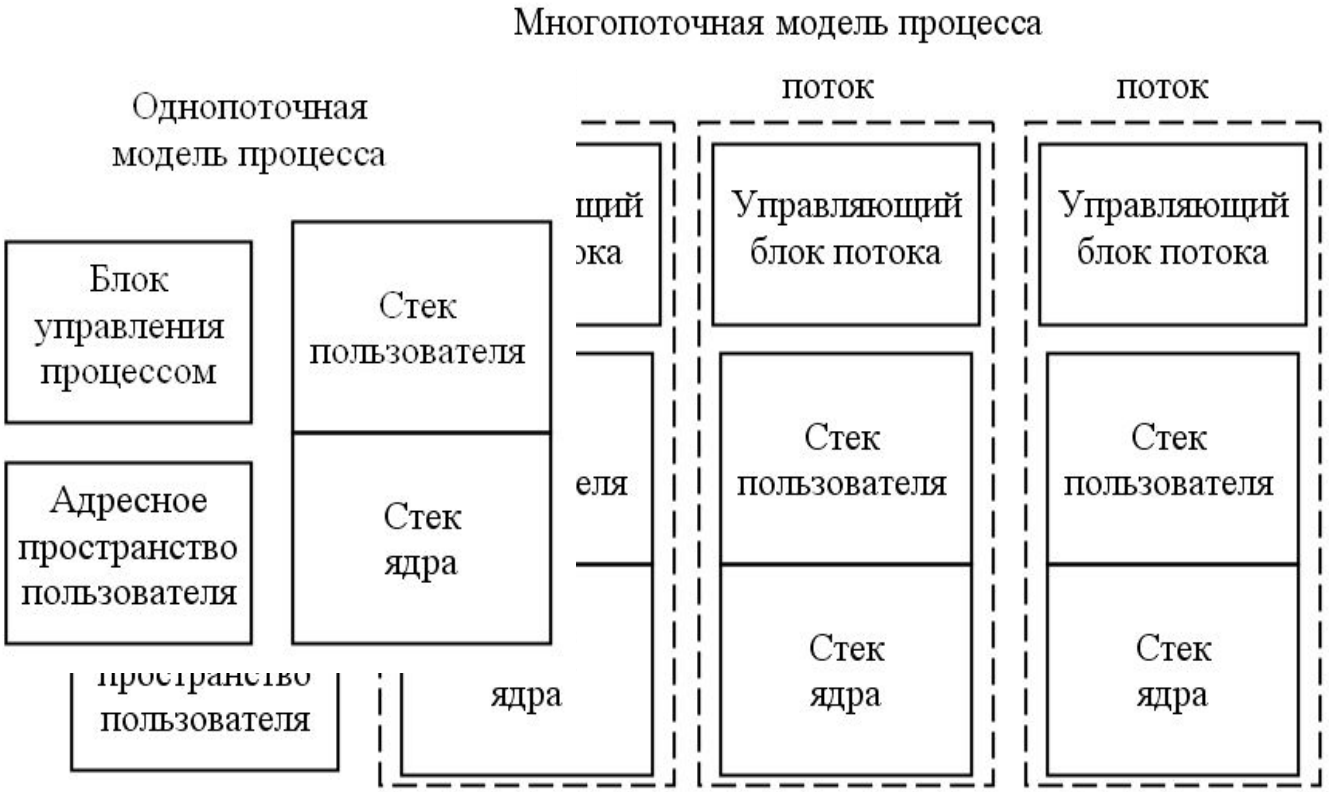
В многопоточной среде процесс определяется как структурная единица распределения ресурсов, а также структурная единица защиты.

В однопоточной модели процесса , как отмечалось выше в его представление входит:

- управляющий блок процесса;
- пользовательское адресное пространство;
- стеки ядра и пользователя(с помощью которых осуществляются вызовы процедур и возвраты из них при выполнении процесса).

Управление процессами

В многопоточной среде с каждым процессом тоже связаны управляющий блок и адресное пространство, но теперь **для каждого потока создаются свои отдельные стеки, а также свой управляющий блок**, в котором содержатся значения регистров, приоритет и другая информация о состоянии потока:



Основные преимущества использования потоков с точки зрения производительности.

1. **Создание нового потока в уже существующем процессе занимает намного меньше времени, чем создание совершенно нового процесса.**

2. **Поток можно завершить намного быстрее, чем процесс.**

3. **Переключение потоков в рамках одного и того же процесса происходит намного быстрее.**

4. **При использовании потоков повышается эффективность обмена информацией между двумя выполняющимися программами(потоками)**. (В большинстве операционных систем обмен между независимыми процессами происходит с участием ядра, в функции которого входит обеспечение защиты и механизма, необходимого для осуществления обмена).

Планирование и диспетчеризация осуществляются на основе потоков; таким образом, **большая часть информации о состоянии процесса**, имеющей отношение к его выполнению, **поддерживается в структурах данных на уровне потоков**.

Однако есть несколько действий, которые затрагивают все потоки процесса и которые операционная система должна поддерживать именно на этом уровне:

Если процесс приостанавливается, то **при этом предполагается, что его адресное пространство будет выгружено из основной памяти**.

Поскольку **все потоки процесса используют одно и то же адресное пространство**, все **они должны одновременно перейти в состояние приостановленных**. Соответственно прекращение процесса приводит к прекращению всех составляющих его потоков.

Планирование процессора

Типы планирования:

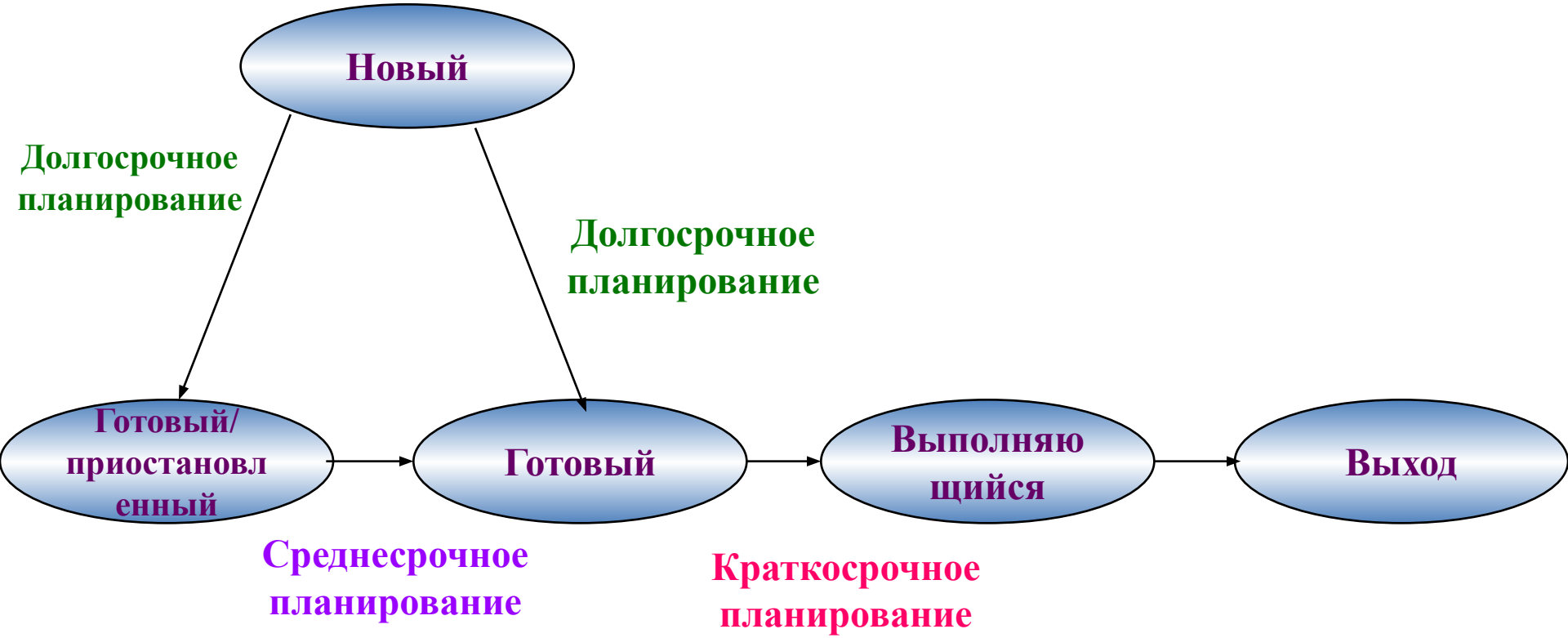
Долгосрочное планирование – решение о добавлении процесса в пул выполняемых процессов.

Среднесрочное планирование – решение о добавлении процесса к числу процессов частично или полностью размещенных в основной памяти.

Краткосрочное планирование – решение о том, какой из готовых процессов будет выполняться процессором.

Планирование ввода-вывода – решение о том, какой из запросов процессов на операции ввода-вывода будет обработан свободным устройством ввода-вывода.

Планирование процессора



АЛГОРИТМЫ ПЛАНИРОВАНИЯ

Критерии краткосрочного планирования:

Пользовательские, связанные с производительностью	
Время оборота	**Интервал времени между подачей процесса и его завершением**. Включает время выполнения, а также время, затраченное на ожидание ресурсов, в том числе и процессора. Критерий вполне применим для пакетных заданий.
Время отклика	В интерактивных процессах это **время, истекшее между подачей запроса и началом получения ответа на него**. Критерий — наиболее подходящий с точки зрения пользователя. Стратегия планирования должна пытаться сократить время получения ответа при максимизации количества интерактивных пользователей, время отклика для которых не выходит за заданные пределы.
Предельный срок	При указании предельного срока завершения процесса планирование должно подчинить ему все прочие цели максимизации количества процессов, завершающихся в срок.

Пользовательские, иные	
Предсказуемость	Данное задание должно выполняться примерно за одно и то же количество времени и с одной и той же стоимостью, независимо от загрузки системы.

Планирование процессора

Системные, связанные с производительностью	
Пропускная способность	Стратегия планирования должна пытаться **максимизировать количество процессов, завершающихся за единицу времени**, что является мерой количества выполненной системой работы. Очевидно, что эта величина зависит от средней продолжительности процесса; однако на нее влияет и используемая стратегия планирования.
Использование процессора	Этот показатель представляет собой **процент времени, в течение которого процессор оказывается занят**. Для дорогих совместно используемых систем этот критерий достаточно важен; в однопользовательских же и некоторых других системах (типа систем реального времени) этот критерий менее важен по сравнению с рядом других.

Системные, иные	
Беспристраст-ность	При отсутствии дополнительных указаний от пользователя или системы **все процессы должны рассматриваться как равнозначные** и ни один процесс не должен подвергнуться голоданию.
Использование приоритетов	**Если процессам назначены приоритеты, стратегия планирования должна отдавать предпочтение процессам с более высоким приоритетом**.
Баланс ресурсов	**Стратегия планирования должна поддерживать занятость системных ресурсов**. Предпочтение должно быть отдано процессу, который недостаточно использует важные ресурсы. Этот критерий включает использование долгосрочного и среднесрочного планирования.

Планирование процессора

Функция выбора определяет, какой из готовых к выполнению процессов будет выбран следующим для выполнения.

Режим решения определяет, в какие моменты времени выполняется функция выбора.

Режимы решения подразделяются на две основные категории:

Невытесняющие: В этом случае находящийся в состоянии выполнения процесс продолжает выполнение до тех пор, пока он не завершится или пока не окажется в заблокированном состоянии ожидания завершения операции ввода-вывода или запроса некоторого системного сервиса.

Вытесняющие: Выполняющийся в настоящий момент процесс может быть прерван и переведен операционной системой в состояние готовности к выполнению. Решение о вытеснении может приниматься при запуске нового процесса по прерыванию, которое переводит заблокированный процесс в состояние готовности к выполнению, или периодически — на основе прерываний таймера.

1. Первым поступил — первым обслужен(FCFS)

Простейшая стратегия планирования "первым поступил — первым обслужен" (first-come-first-served — FCFS) известна также как схема "первым пришел — первым вышел", или схема строгой очередности.

Как только процесс становится готовым к выполнению, он присоединяется к очереди готовых процессов. При прекращении выполнения текущего процесса для выполнения выбирается процесс, который находился в очереди дольше других.

2. Круговое планирование(RR)

Стратегия кругового (карусельного) планирования (round robin — RR) - таймер генерирует прерывания через определенные интервалы времени. При каждом прерывании исполняющийся в настоящий момент процесс помещается в очередь готовых к выполнению процессов, и начинает выполняться очередной процесс, выбираемый в соответствии со стратегией FCFS. Эта методика известна также как **квантование времени** (time slicing), поскольку перед тем как оказаться вытесненным, каждый процесс получает квант времени для выполнения.

3. Наиболее короткий процесс следующий(SPN)

Если в очереди появляется короткий процесс, то он выполняется первым. Автоматически система не может определить какой процесс короче. Такой метод используется в пакетной обработке, т.к. оператор может указать ориентировочное время выполнения задачи. Так же может быть использована статистика, которая накапливается в системе. Однако возникает голодание более длинных процессов.

4. Наименьшее остающееся время(SRT)

Стратегия наименьшего остающегося времени (shortest remaining time — SRT) представляет собой **вытесняющую версию стратегии SPN**.

В этом случае планировщик выбирает процесс с наименьшим **ожидаемым временем** до окончания процесса.

При присоединении нового процесса к очереди готовых к исполнению процессов может оказаться, что его оставшееся время в действительности меньше, чем оставшееся время выполняемого в настоящий момент процесса. Планировщик, соответственно, может применить вытеснение при готовности нового процесса. Как и при использовании стратегии SPN, **планировщик для корректной работы функции выбора должен оценивать время выполнения процесса.** В этом случае также имеется риск голодания длинных процессов.

5. Наивысшее отношение отклика(HRRN)

Рассмотрим соотношение

$$R = \frac{w+s}{s}, \text{ где}$$

R — отношение отклика; **w** — время, затраченное процессом на ожидание;

s — ожидаемое время обслуживания.

Таким образом, правило стратегии планирования наивысшего отношения отклика (highest response ratio next — HRRN) можно сформулировать так: при завершении или блокировании текущего процесса для выполнения из очереди готовых процессов выбирается тот, который имеет наибольшее значение R.

6. Снижение приоритета

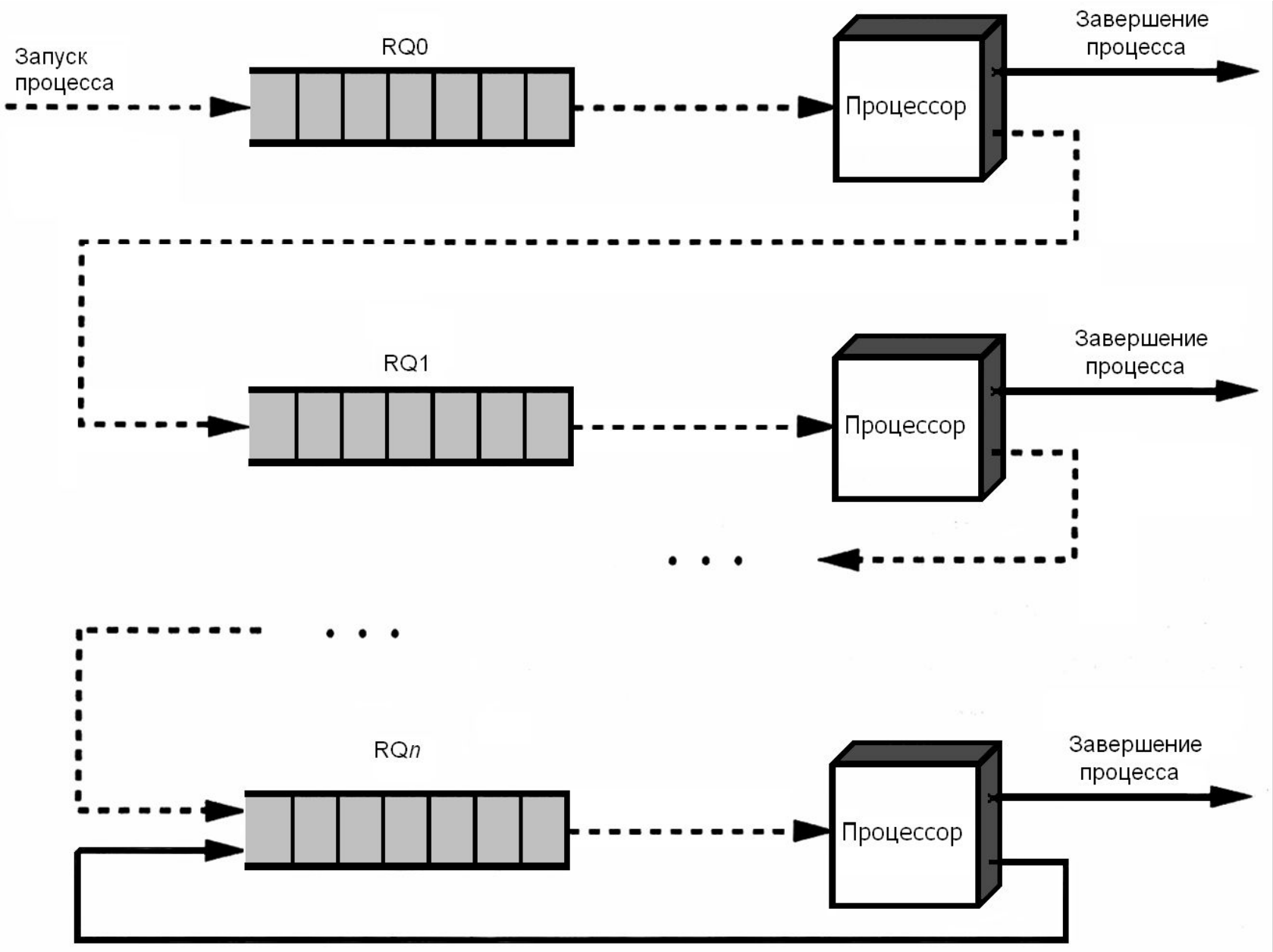
Если у нас нет никаких указаний об относительной продолжительности процессов, то мы не можем использовать ни одну из стратегий — SPN, SRT или HRRN. Еще один путь предоставления преимущества коротким процессам состоит в применении штрафных санкций к долго выполняющимся процессам. Другими словами, раз уж мы не можем работать с оставшимся временем выполнения, мы будем работать с затраченным временем.

Вот как этого можно достичь. **Выполняется вытесняющее (по квантам времени) планирование с использованием динамического механизма**. При входе процесса в систему он помещается в очередь RQ0. После первого выполнения и возвращения в состояние готовности процесс помещается в очередь RQ1. В дальнейшем при каждом вытеснении этого процесса он вносится в очередь со все меньшим приоритетом.

По достижении очереди с наиболее низким приоритетом процесс уже не покидает ее, всякий раз после вытеснения попадая в нее вновь (таким образом, эта очередь, по сути, обрабатывается с использованием циклической стратегии).

Такой подход известен как многоуровневый возврат (multilevel feedback2), поскольку при блокировании или вытеснении процесса осуществляется его возврат, на очередной уровень приоритетности

Планирование процессора

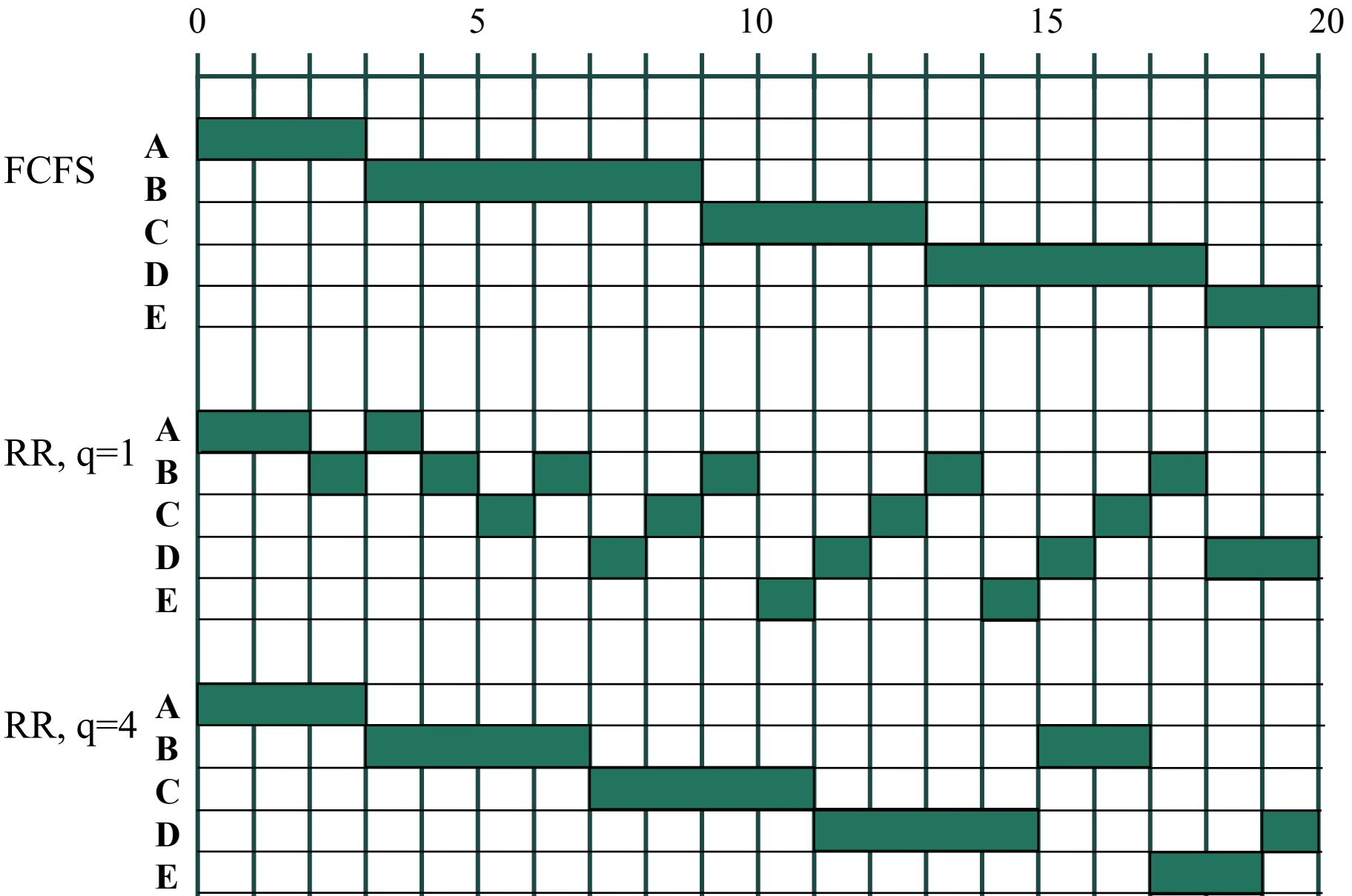


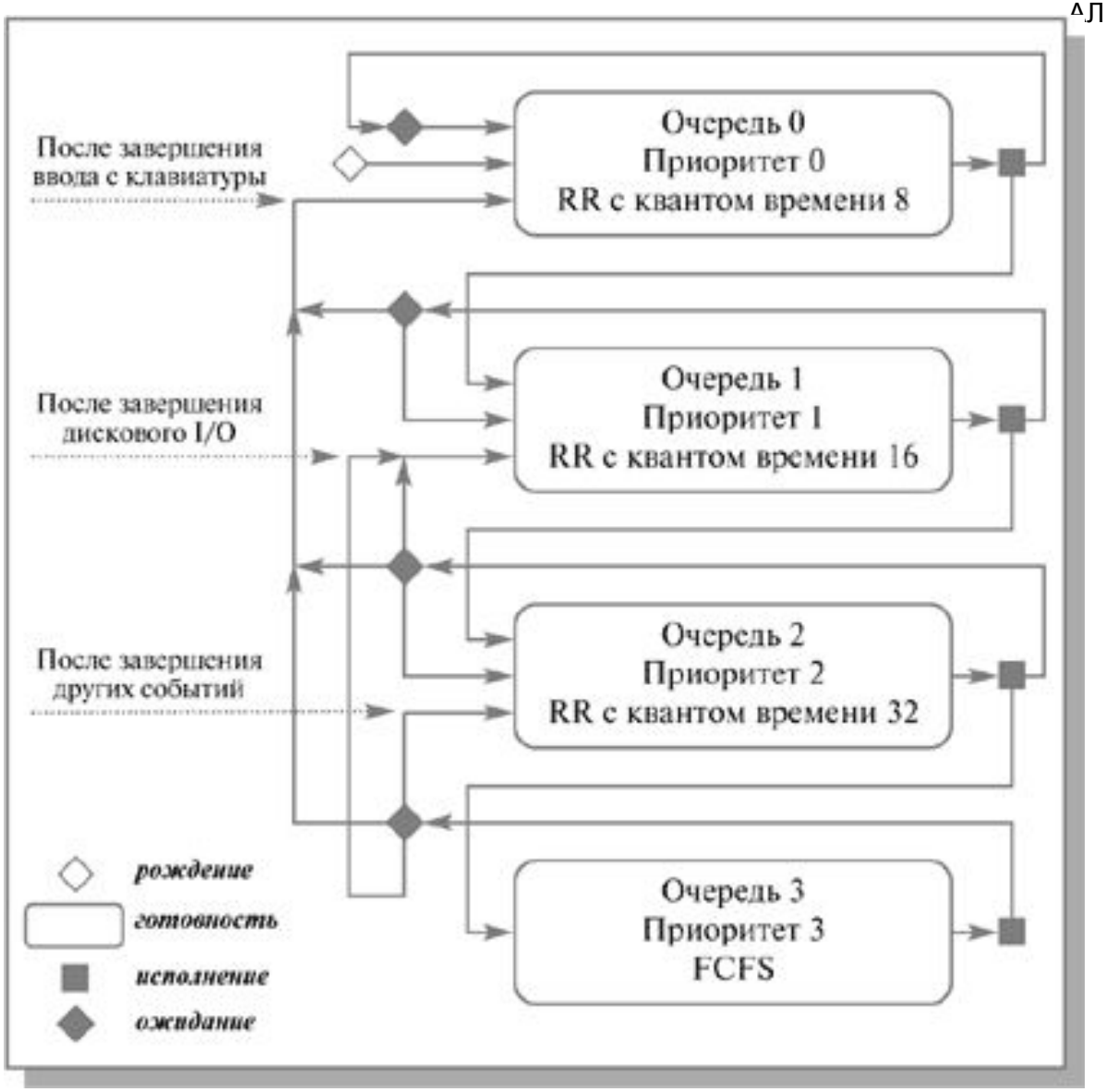
Планирование процессора

Пример планирования процессов

Процесс	Время запуска	Время обслуживания
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Пример стратегий планирования





Вариант реализации алгоритма многоуровневого возврата(FB)

Характеристики различных алгоритмов планирования

Таблица 1 - Характеристики различных стратегий планирования

	Функция выбора	Режим решения	Пропускная способность	Время отклика	Накладные расходы	Влияние на процессы	Голодание
FCFS	Max[w]	Невытесняющий	Неважна	Может быть большим	Минимальны	Плохо сказывается на коротких процессах и процессах с интенсивным вводом-выводом	-
RR	const	Вытесняющий (по времени)	Может быть низкой при малом кванте времени	Обеспечивает хорошее время отклика	Минимальны	Беспристрастна	-
SPN	Min[s]	Невытесняющий	Высокая	Обеспечивает хорошее время отклика для коротких процессов	Могут быть высокими	Плохо сказывается на длинных процессах	+
SRT	Min[s — *e*]	Вытесняющий (по решению)	Высокая	Обеспечивает хорошее время отклика для коротких процессов	Могут быть высокими	Плохо сказывается на длинных процессах	+
HRRN	Max[(w+s)/s]	Невытесняющий	Высокая	Обеспечивает хорошее время отклика для коротких процессов	Могут быть высокими	Хороший баланс	-
FB	-	Вытесняющий (по времени)	Неважна	Обеспечивает хорошее время отклика для коротких процессов	Могут быть высокими	Может привести к предпочтению процессов с интенсивным вводом-выводом	+

Выводы

- Одним из ограниченных ресурсов вычислительной системы является **процессорное время**.
- Для его распределения между многочисленными процессами ОС применяет **процедуру планирования процессов**.
- По степени длительности влияния планирования на поведение вычислительной системы различают **краткосрочное, среднесрочное и долгосрочное** планирование процессов.
- Различают **вытесняющий** и **невытесняющий** режимы решения алгоритма краткосрочного планирования.
- Конкретные алгоритмы планирования процессов зависят от поставленных перед вычислительной системой целей и класса решаемых задач.

Синхронизация процессов и потоков

Понятие синхронизации процессов и потоков

В мультипрограммной операционной системе иногда возникает необходимость в **синхронизации** процессов (или потоков).

Далее будем говорить о синхронизации потоков, имея в виду, что если операционная система не поддерживает потоки либо если происходит взаимодействие процессов, то все сказанное относится и к синхронизации процессов.

Потоки в общем случае (когда программист не предпринял специальных мер по их синхронизации) выполняются независимо - *асинхронно* друг другу.

Любое взаимодействие потоков связано с их *синхронизацией*, которая заключается в согласовании их скоростей путем **приостановки** потока до наступления некоторого события и последующей его **активизации** при наступлении этого события.

Понятие синхронизации процессов и потоков 2

Потокам(процессам) **необходимо** взаимодействовать между собой для выполнения таких задач, как:

- обмен данными;
- совместное использование данных;
- явное или неявное использование общих ресурсов.

При необходимости использовать один и тот же ресурс параллельные процессы(потоки) *конкурируют* друг с другом за получение доступа к ресурсу.

В случае конкуренции процессов за ресурс может возникать **эффект гонок** - ситуация, когда два или более процессов обрабатывают разделяемые данные, и конечный результат обработки зависит от соотношения скоростей процессов.

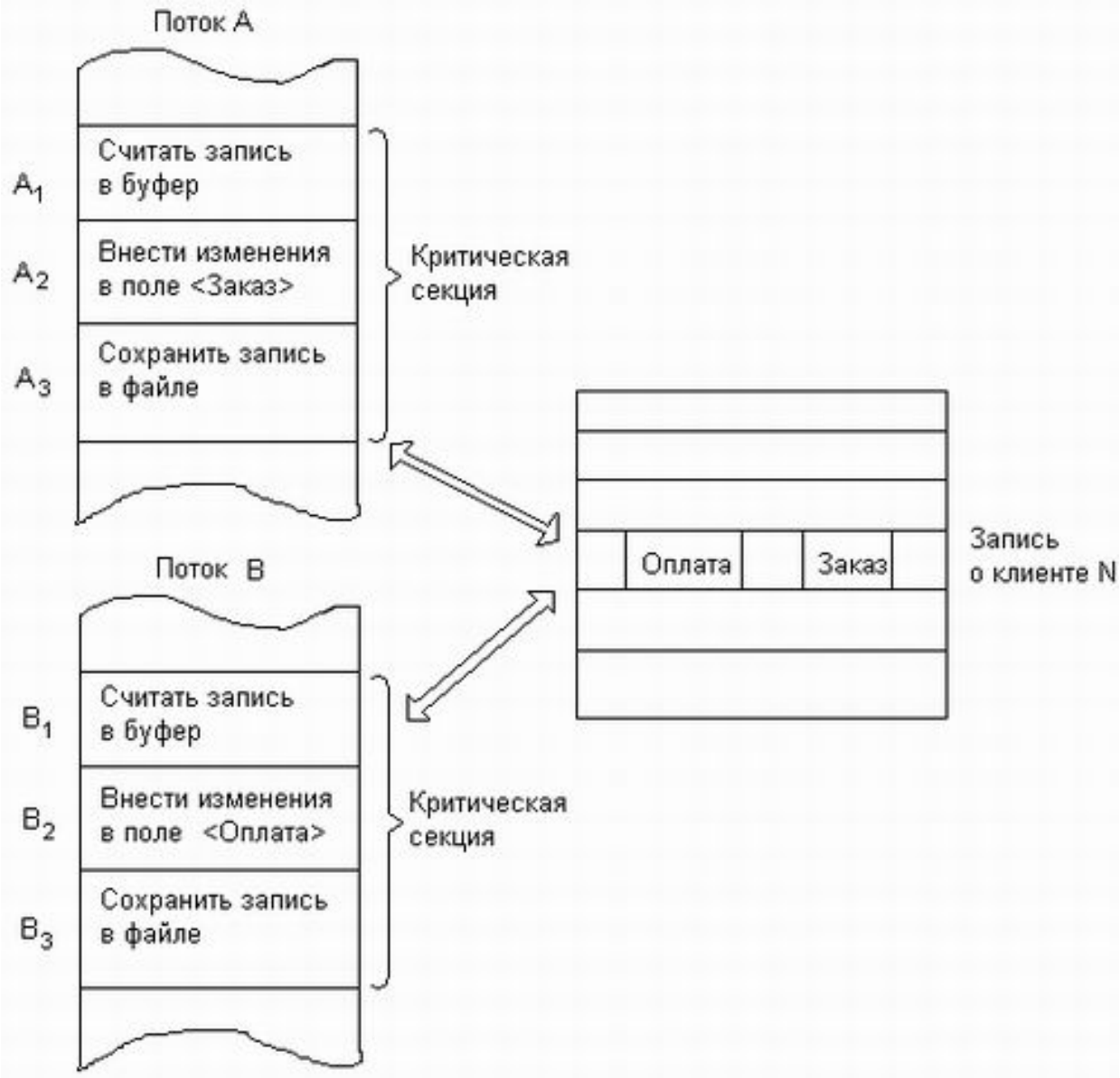
Пример возникновения эффекта гонок

Рассмотрим, задачу ведения базы данных клиентов некоторого предприятия. Каждому клиенту отводится отдельная запись в базе данных, в которой среди прочих полей имеются поля **Заказ** и **Оплата**.

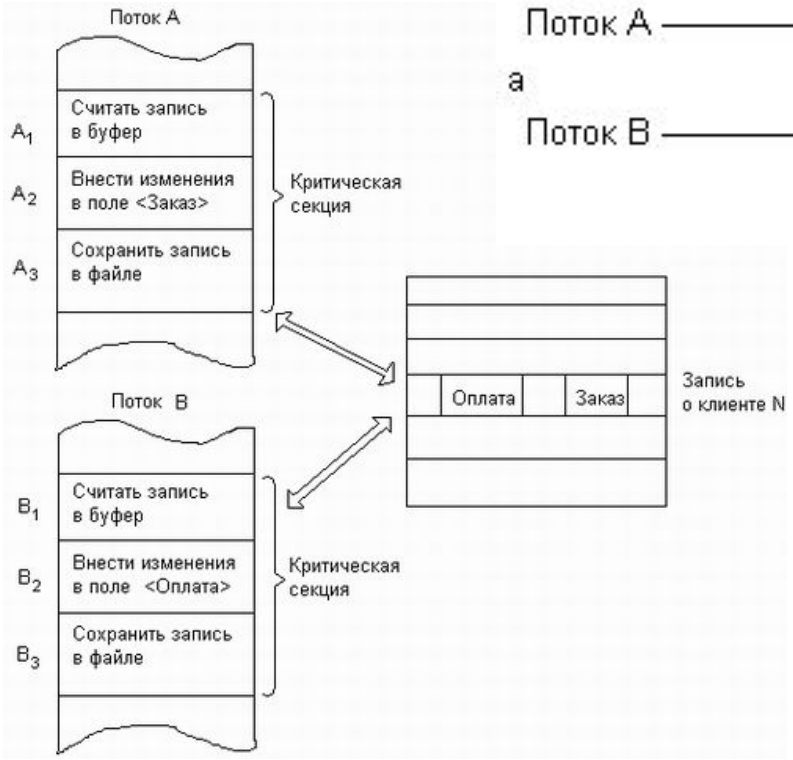
Программа, ведущая базу данных, реализована как единый процесс, имеющий несколько потоков, в том числе поток А, который **заносит в базу данных информацию о заказах**, поступивших от клиентов, и поток В, который **фиксирует в базе данных сведения об оплате клиентами выставленных счетов**. Оба эти потока совместно работают над общим файлом базы данных, используя однотипные алгоритмы, включающие три шага:

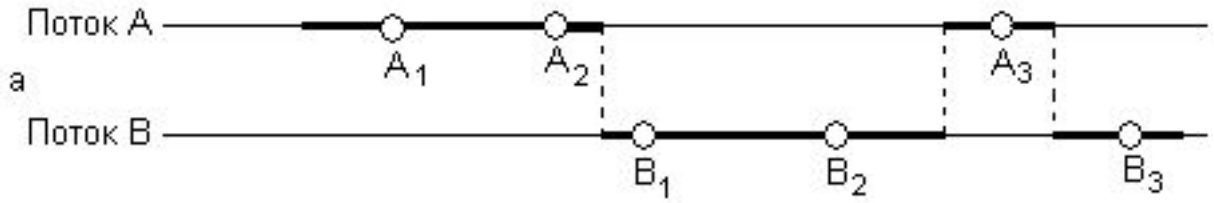
1. Считать из файла базы данных в буфер запись о клиенте с заданным идентификатором(уникальным номером).
2. Внести новое значение в поле Заказ (для потока А) или Оплата (для потока В).
3. Вернуть модифицированную запись в файл базы данных.

Пример возникновения эффекта гонок



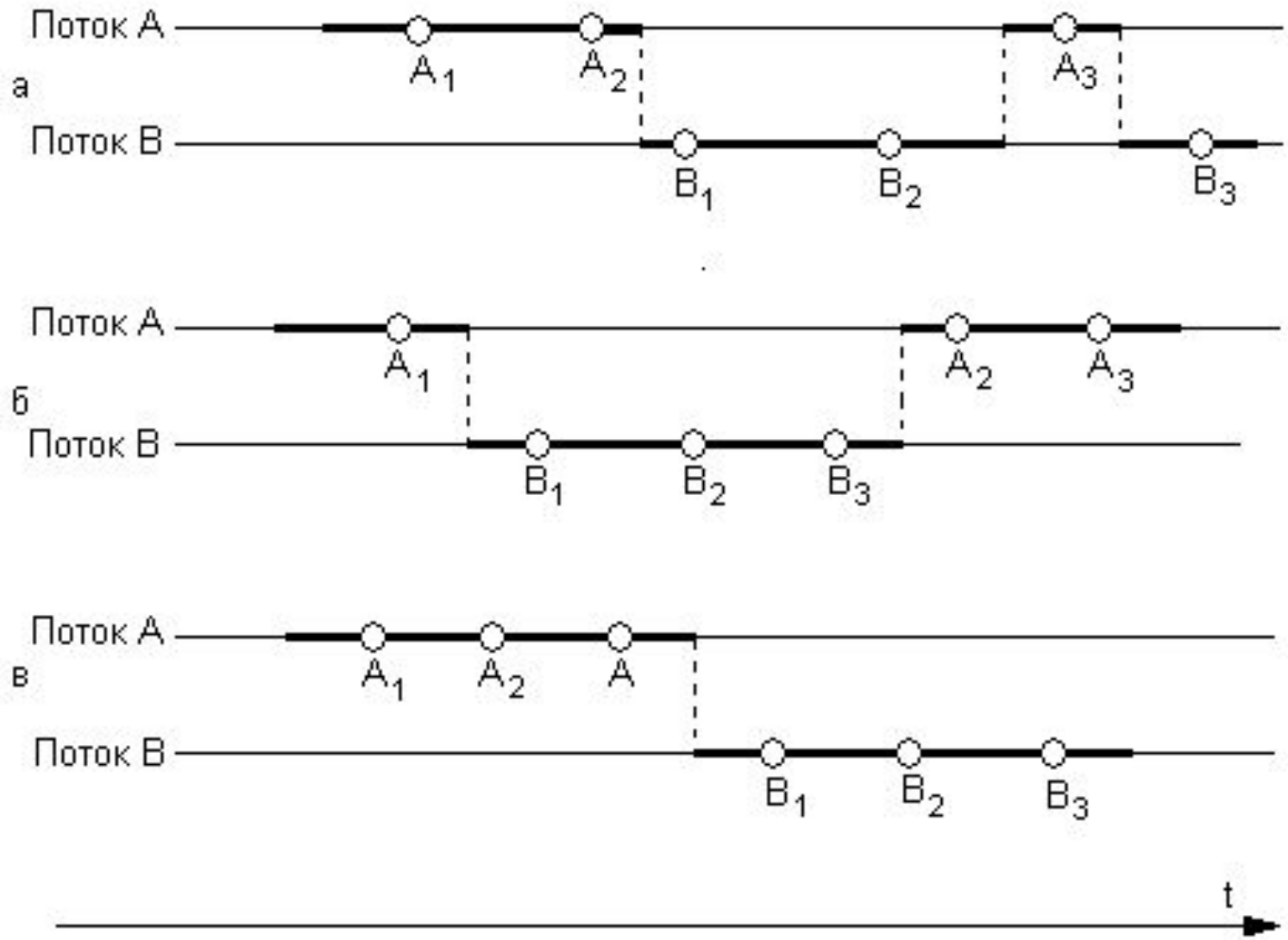
Пример возникновения эффекта гонок





В случае прерывания потока A на шаге A3 и активизации потока B, последний при выполнении шага B3 запишет в базу данных поверх только что обновленной записи о клиенте N свой вариант записи, в которой обновлено значение поля *Оплата* (но не обновлено поле *Заказ*). Таким образом, в базе данных будут зафиксированы сведения о том, что клиент N произвел оплату, но информация о его заказе окажется потерянной.

Пример возникновения эффекта гонок



Варианты соотношения скоростей потоков и моментов их прерывания

Пример возникновения эффекта гонок

Как видно, сложность выявления эффекта гонок состоит в нерегулярности возникающих ситуаций. В примере:

а) потеряна информация о заказе;

б) потеряна информация об оплате;

в) все изменения были успешно внесены.

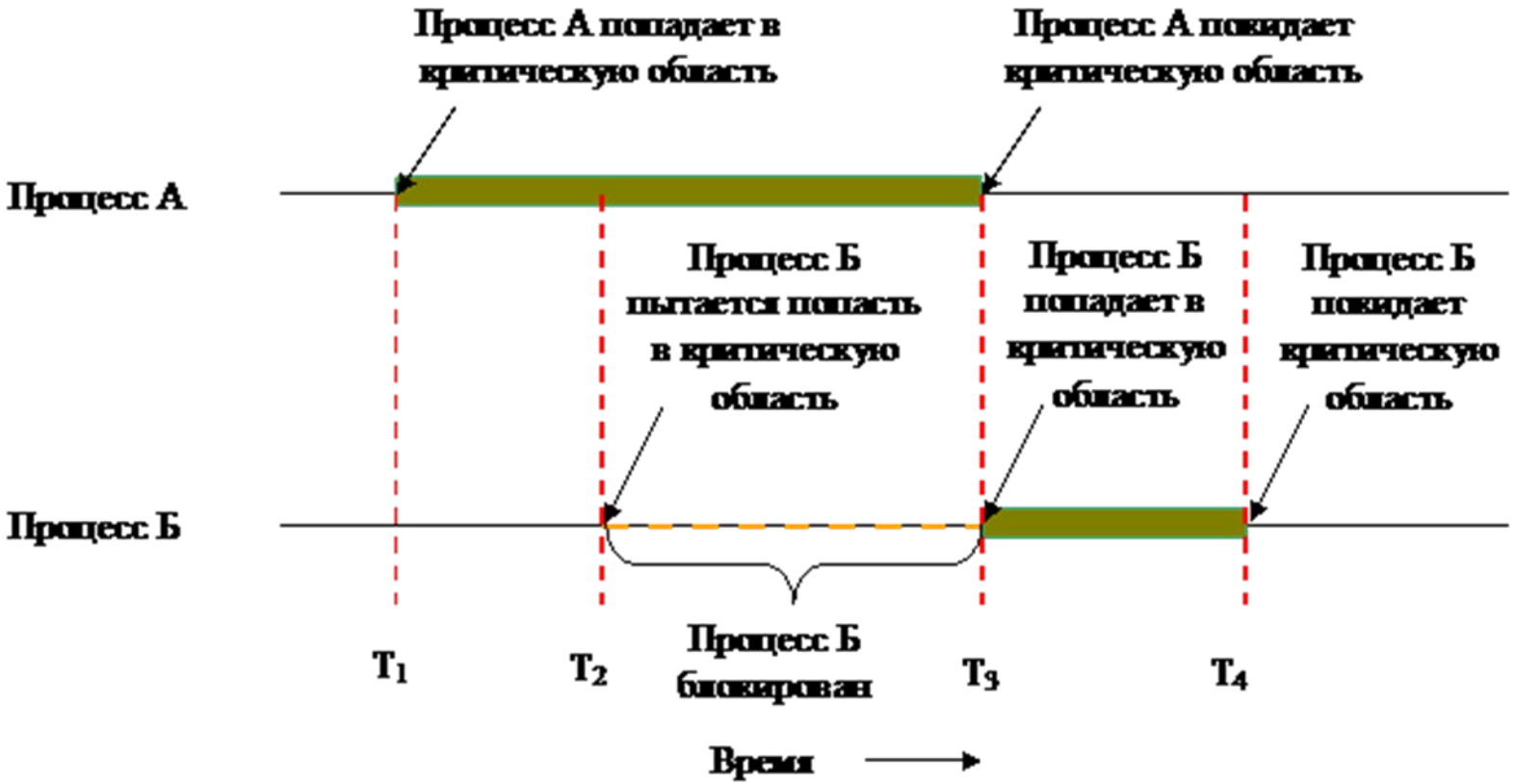
Все определяется взаимными скоростями потоков и моментами их прерывания.

Поэтому отладка взаимодействующих потоков является сложной задачей.

Понятие критической секции программы и взаимных исключений

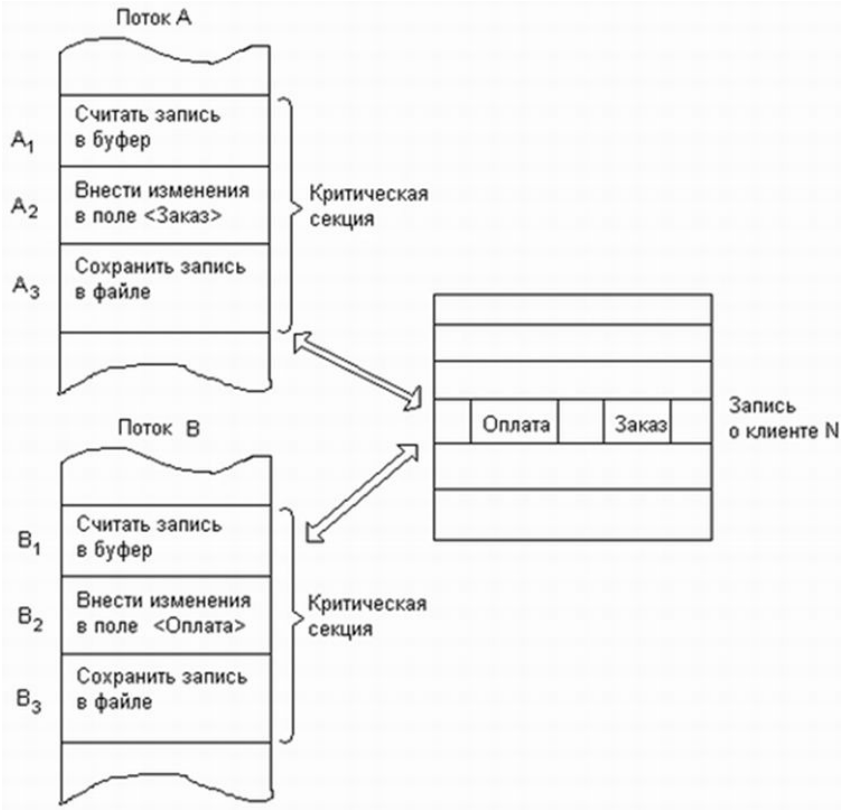
Важным понятием синхронизации потоков является понятие **критической секции** программы.

Критическая секция(критический раздел, критическая область) - это часть программы, в которой осуществляется доступ к разделяемым ресурсам (данным).



Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо обеспечить, чтобы **в каждый момент в критической секции, связанной с этим ресурсом, находился максимум один поток(процесс)**.

Такой прием и называют **взаимным исключением**.



Для синхронизации потоков прикладных программ программист может использовать как собственные средства и приемы синхронизации, так и средства операционной системы.

Например, два потока одного прикладного процесса могут координировать свою работу с помощью доступной для них обоих глобальной логической переменной, которая устанавливается в единицу при осуществлении некоторого события, например выработки одним потоком данных, нужных для продолжения работы другого.

Однако во многих случаях более эффективными или даже единственно возможными являются средства синхронизации, предоставляемые операционной системой в форме объектов синхронизации или системных вызовов.

Операционные системы позволяют использовать различные способы реализации взаимного исключения.

Некоторые способы пригодны для взаимного исключения при вхождении в критическую секцию только потоков одного процесса, в то время как другие могут обеспечить взаимное исключение и для потоков разных процессов.

Использование блокирующих переменных

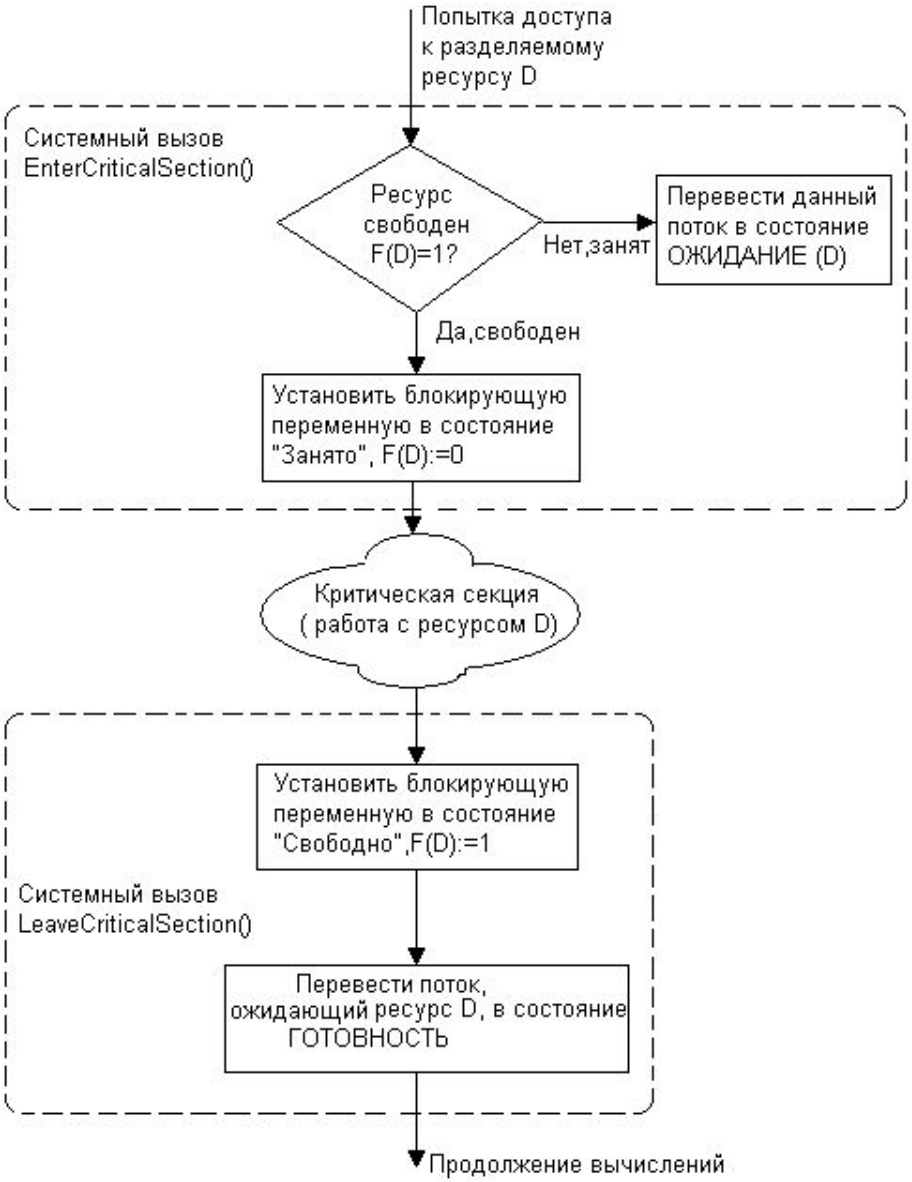
Один из способов обеспечения взаимного исключения - использование блокирующих переменных.

С каждым разделяемым ресурсом связывается двоичная переменная, которая принимает значение 1, если ресурс свободен (то есть ни один процесс не находится в данный момент в критической секции, связанной с данным процессом), и значение 0, если ресурс занят.

Реализация критических секций с использованием блокирующих переменных

Реализация взаимного исключения описанным выше способом имеет существенный **недостаток**: в течение времени, когда один поток находится в критической секции, другой поток, которому требуется тот же ресурс, получив доступ к процессору, будет непрерывно опрашивать блокирующую переменную, **бесполезно тратя выделяемое ему процессорное время**, которое могло бы быть использовано для выполнения какого-нибудь другого потока. Для устранения этого недостатка во многих ОС предусматриваются **специальные системные вызовы** для работы с критическими секциями.

Использование блокирующих переменных



Реализация взаимного исключения с использованием системных функций входа в критическую секцию и выхода из нее

Синхронизация потоков различных процессов

Рассмотренные выше механизмы синхронизации, *основанные на использовании глобальных переменных процесса*, обладают существенным недостатком — они *не подходят для синхронизации потоков разных процессов*.

При необходимости синхронизации потоков различных процессов операционная система должна предоставлять потокам **системные объекты** синхронизации, которые были бы **доступны для всех потоков**, даже если они принадлежат разным процессам и работают в разных адресных пространствах.

Примерами таких синхронизирующих объектов ОС являются:

- мьютексы;
- семафоры;
- события;
- таймеры.

Синхронизация потоков различных процессов

Чтобы процессы могли разделять синхронизирующие объекты, в разных ОС используются разные методы.

Некоторые ОС возвращают указатель на объект. Этот указатель может быть доступен всем родственным процессам, наследующим характеристики общего родительского процесса.

В других ОС процессы в запросах на создание объектов синхронизации указывают имена, которые должны быть им присвоены. Далее эти имена используются разными процессами для манипуляций объектами синхронизации.

Объекты могут находиться в двух состояниях: сигнальном и несигнальном — свободном.

> Для каждого объекта смысл, вкладываемый в понятие «сигнальное состояние», зависит от типа объекта. Так, например, поток переходит в сигнальное состояние тогда, когда он завершается. Процесс переходит в сигнальное состояние тогда, когда завершаются все его потоки. Файл переходит в сигнальное состояние в том случае, когда завершается операция ввода-вывода для этого файла. Для остальных объектов сигнальное состояние устанавливается в результате выполнения специальных системных вызовов.

Приостановка и активизация потоков осуществляются в зависимости от состояния синхронизирующих объектов ОС.

Семафоры

Обобщающее средство синхронизации процессов предложил Дейкстра, который **ввел два новых примитива**. В абстрактной форме эти примитивы, обозначаемые P и V, оперируют над целыми неотрицательными защищенными переменными, называемыми **семафорами**.

Пусть S такой семафор. Операции определяются следующим образом:

V(S) : переменная S увеличивается на 1 одним неделимым действием; выборка, **инкремент и запоминание не могут быть прерваны**, и к S нет доступа другим процессам во время выполнения этой операции.

P(S) : уменьшение S на 1, **если это возможно**. Если S=0, то невозможно уменьшить S и остаться в области целых неотрицательных значений, в этом случае процесс, вызывающий **P**-операцию, ждет, пока это уменьшение станет возможным. **Успешная проверка и уменьшение также является неделимой операцией.**

В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную. Такой семафор называется бинарным**.** В противном случае семафор называется **обобщенным**(или **считающим**).

Операция P заключает в себе потенциальную **возможность перехода** процесса, который ее выполняет, **в состояние ожидания**, в то время как **V-операция может** при некоторых обстоятельствах **активизировать другой процесс**, приостановленный при выполнении для него операции **P**

Семафоры

Рассмотрим использование семафоров на классическом примере взаимодействия двух процессов «Поставщик» - «Потребитель», выполняющихся в режиме мультипрограммирования, один из которых пишет данные в буферный пул, а другой считывает их из буферного пула.

Пусть буферный пул состоит из N буферов, каждый из которых может содержать одну запись. Процесс "писатель" должен приостанавливаться, когда все буфера оказываются занятыми, и активизироваться при освобождении хотя бы одного буфера. Напротив, процесс "читатель" приостанавливается, когда все буферы пусты, и активизируется при появлении хотя бы одной записи.

Введем два семафора:
e - число пустых буферов
f - число заполненных буферов.
Введем также двоичный семафор b, используемый для обеспечения взаимного исключения. Тогда процессы могут быть описаны следующим образом:

Семафоры

```
// Глобальные переменные
#define N 256
int e = N, f = 0, b = 1;
void Writer ()
{
    while(1)
    {
        PrepareNextRecord(); /* подготовка новой записи   */
        P(e);          /* Уменьшить число свободных буферов, если они
        есть */
                    /* в противном случае - ждать, пока они освободятся */
        P(b);           /* Вход в критическую секцию  */
        AddToBuffer();  /* Добавить новую запись в буфер */
        V(b);           /* Выход из критической секции   */
        V(f);          /* Увеличить число занятых буферов */
    }
}
```

Семафоры

```
void Reader ()
{
    while(1)
    {
        P(f); /* Уменьшить число занятых буферов, если они есть */
                    /* в противном случае ждать, пока они появятся    */
        P(b);            /* Вход в критическую секцию                  */
        GetFromBuffer();  /* Взять запись из буфера                     */
        V(b);            /* Выход из критической секции                */
        V(e);            /* Увеличить число свободных буферов          */
        ProcessRecord();  /* Обработать запись                          */
    }
}
```

Программная реализация взаимоисключений

Программно взаимоисключение может быть реализовано для параллельных процессов, которые выполняются как в однопроцессорной, так и в многопроцессорной системе с разделяемой основной памятью.

Дейкстра приводит **алгоритм для организации взаимных исключений двух процессов**, предложенный голландским математиком Деккером (Dekker).

Алгоритм Деккера

```
boolean flag[2]; int turn;
void P0(){
  while (true){
      flag[0]   =  true;
      while(flag[l] )
       if (turn == 1) {
            flag[0] = false;
            while(turn == 1)  /* Ничего не делать */;
            flag[0] = true;
             }
      /* Критический раздел */,
      turn = 1;
      flag[0] = false;
      /* Остальной код */;
  }
}
```

Алгоритм Деккера

```
void P1(){
  while(true){
      flag[1] = true;
      while(flag[0] )
         if (turn == 0) {
            flag[1] = false;
            while (turn == 0)/* Ничего не делать */;
            flag[1] = true;
         }
      /* Критический раздел */;
      turn = 0;
      flag[1] = false;
      /* Остальной код */;
  }
}
void main ()
{
 flag[0] = false;
 flag[1] = false;
 turn = 1;
 parbegin(P0, P1) ;
}
```

Алгоритм Деккера

Для того чтобы доказать, что алгоритм Деккера обеспечивает взаимоисключение:
1. Возможно показать, что когда процесс **Pi** входит в критический раздел, истинно следующее выражение:

flag[i] and(not flag[1-i])

Для того чтобы доказать что алгоритм Деккера не приводит к тупику(процесс запрашивающий доступ к критическому разделу не может быть **навсегда** задержан). Достаточно рассмотреть случаи, когда:

1. Только один процесс пытается войти в критический раздел;
2. Оба процесса пытаются войти в критический раздел, при этом:
    1. **turn=0 и flag[0]=false**
    2. **turn=0 и flag[0]=true**

Алгоритм Деккера

Для того чтобы доказать, что алгоритм Деккера обеспечивает взаимоисключение:

1. Возможно показать, что когда процесс **Pi** входит в критический раздел, истинно следующее выражение:

flag[i] and(not flag[1-i])

Для того чтобы доказать что алгоритм Деккера не приводит к тупику(процесс запрашивающий доступ к критическому разделу не может быть **навсегда** задержан). Достаточно рассмотреть случаи, когда:

1. Только один процесс пытается войти в критический раздел;
2. Оба процесса пытаются войти в критический раздел, при этом:
    1. **turn=0 и flag[0]=false**
    2. **turn=0 и flag[0]=true**

Алгоритм Петерсона

Алгоритм Деккера решает задачу взаимных исключений, но достаточно сложным путем, корректность которого не так легко доказать. Петерсон (Peterson) предложил более простое и элегантное решение. Как и ранее, глобальная переменная **flag** указывает положение каждого процесса по отношению к взаимоисключению, а глобальная переменная **turn** разрешает конфликты одновременности.

Алгоритм Петерсона

```
boolean flag[2] int turn; void P0() {
while (true){
      flag[0]   = true;
      turn = 1;
      while(flag[1] && turn == 1) /* Ничего не делать */;
         /* Критический раздел */;
      flag[0] = false;
      /* Остальной код */;
}
void P1() {
while(true){
      flag[1] = true; turn = 0;
      while(flag[0] && turn == 0) /* Ничего не делать */;
      /* Критический раздел */;
      flag[1] = false;
      /* Остальной код */;
    }
}
void main()
{
flag[0] = false;flag[1] = false;parbegin(PO,Pl);
}
```

Алгоритм Петерсона

Выполнение условий взаимоисключения легко показать.

Рассмотрим процесс P0. После того, как flag[0] установлен им равным true, P1 войти в критический раздел не может.

Если же P1 уже находится в критическом разделе, то flag[1] = true и для P0 вход в критический раздел заблокирован.

Взаимная блокировка в данном алгоритме предотвращена:

Предположим, что P0 заблокирован в своем цикле while. Это означает, что flag[1] равен true, а turn = 1. P0 может войти в критический раздел, когда либо flag[1] становится равным false, либо turn становится равным 0. Рассмотрим три исчерпывающих случая.

1. P1 не намерен входить в критический раздел. Такой случай невозможен, поскольку при этом выполнялось бы условие flag[1] = false.

2. P2 ожидает вход в критический раздел. Такой случай также невозможен, поскольку если turn = 1, то P1 способен войти в критический раздел.

3. P1 циклически использует критический раздел, монополизировав доступ к нему. Этого не может произойти, поскольку P1 вынужден перед каждой попыткой входа в критический раздел дать такую возможность процессу P0, устанавливая значение turn равным 0.

Следовательно, у нас имеется простое решение проблемы взаимных исключений для двух процессов.

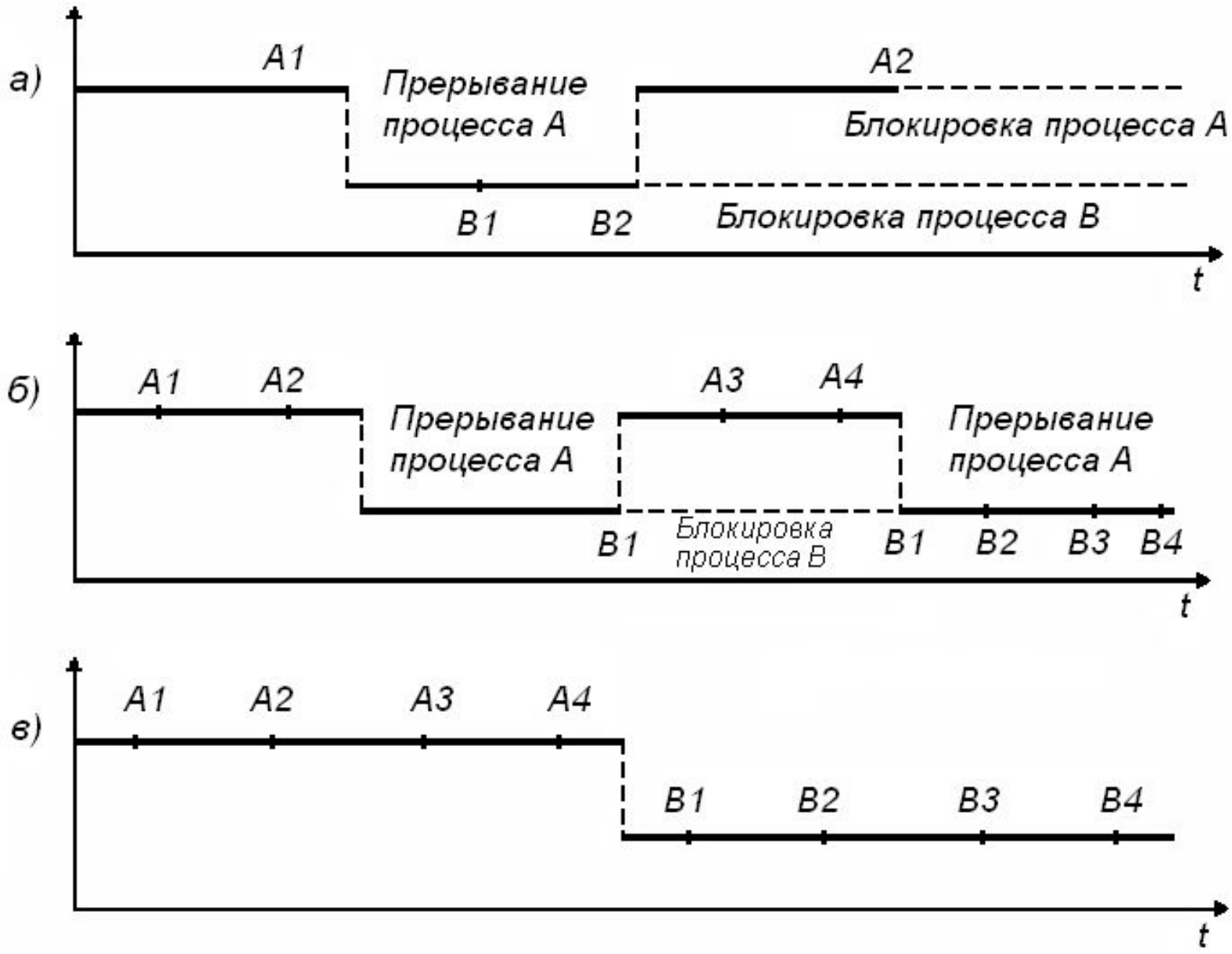
Взаимные блокировки (Тупики)

Рассмотрим пример возникновения взаимной блокировки (тупика).

Пусть двум процессам, выполняющимся в режиме мультипрограммирования, для выполнения их работы нужно два ресурса, например, принтер и диск.

На рисунке показаны фрагменты соответствующих программ.

Взаимные блокировки (Тупики)



Алгоритм предотвращения тупиков – алгоритм банкира

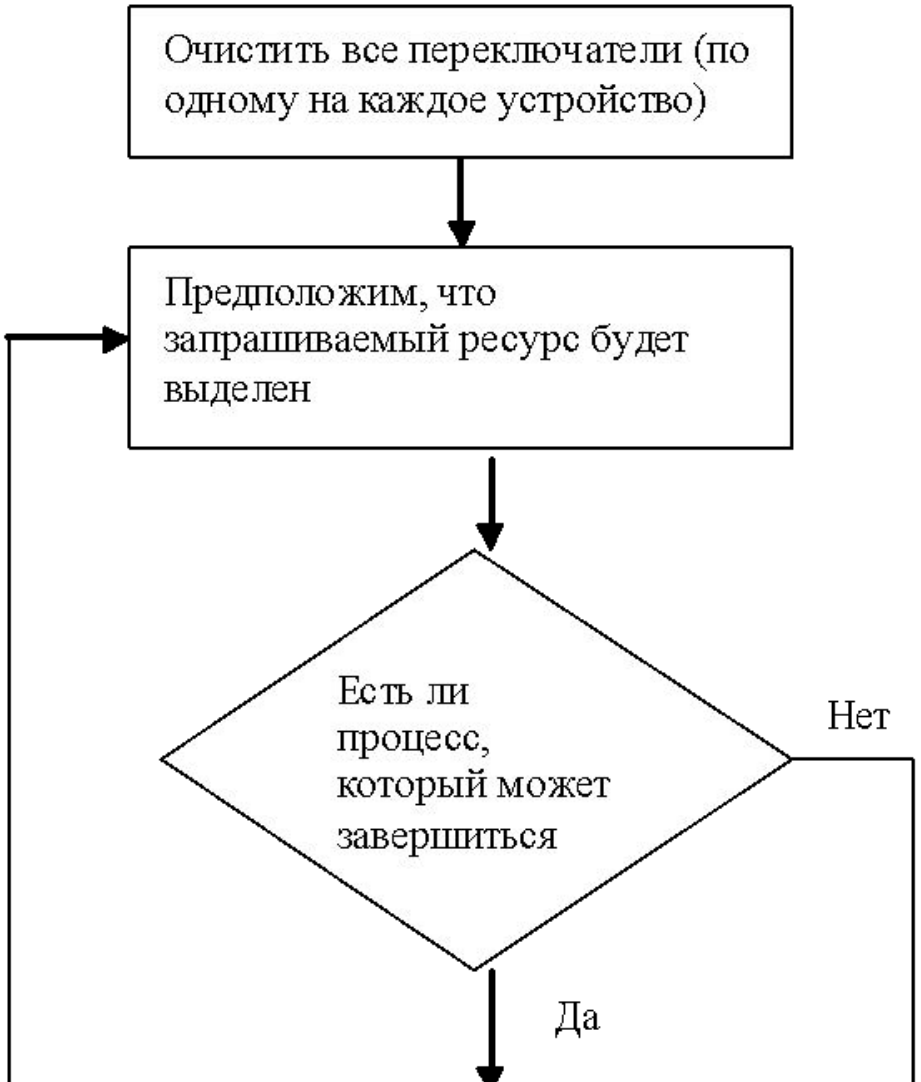
"Алгоритм банкира" (предложил Дейкстра) напоминает процедуру принятия решения, может ли банк безопасно выдать заем.

Для реализации этого метода необходимо выполнить следующие правила:

Заранее объявлять все необходимые заданию ресурсы.

Перед назначением ресурса осуществлять проверку на возможность возникновения клинчей. Если эта возможность исключена, то ресурс можно назначить.

Алгоритм предотвращения тупиков – алгоритм банкира



Алгоритм предотвращения тупиков – алгоритм банкира

Каждому процессу поставлено в соответствие целое число i(1<=i<=N).

Процессу i соответствует его максимальная потребность в устройствах **МАКС[i]**, количество устройств, выделенных ему в данный момент (**ВЫДЕЛУСТР[i]**), полагающийся ему остаток (**ОСТАТОК[i])** и признак (**МОЖЕТ_НЕ_ОКОНЧИТЬСЯ[i]**).

Система заводит глобальную переменную **ОБЩ**, обозначающую общее число имеющихся в системе устройств.

В начале работы неизвестно, может ли какой-либо процесс окончиться (**МОЖЕТ_НЕ_ОКОНЧИТЬСЯ[i]=true** для всех i).

Каждый раз, когда какой-то **ОСТАТОК** может быть выделен из числа остающихся незанятыми устройств, предполагается, что соответствующий процесс работает, пока не окончится, а затем его устройства освобождаются.

Если в конце концов все устройства освободятся, значит, все процессы могут окончиться и система находится в безопасном состоянии. Если состояние системы не безопасное, то она не удовлетворяет рассматриваемый запрос.

Алгоритм предотвращения тупиков – алгоритм банкира

```
Begin
СВОБУСТР:=ОБЩУСТР;
For i:= 1 to N do Begin
       СВОБУСТР:= СВОБУСТР – ВЫДЕЛУСТР[i];
       МОЖЕТ_НЕ_ОКОНЧИТЬСЯ[i]:=true;
           ОСТАТОК[i]:= МАКС[i]-ВЫДЕЛУСТР[i];
 End;
ПРИЗНАК:=true;
WHILE (ПРИЗНАК) DO
    ПРИЗНАК:=false;
     For i:=1 to N do
     Begin
          If МОЖЕТ_НЕ_ОКОНЧИТЬСЯ[i] and (ОСТАТОК[i] <= СВОБУСТР)
            Then begin
              МОЖЕТ_НЕ_ОКОНЧИТЬСЯ[i]:=false;
              СВОБУСТР:= СВОБУСТР+ ВЫДЕЛУСТР[i];
              ПРИЗНАК:=true
              End;
      End;
    End;
End;
If СВОБУСТР= ОБЩУСТР then состояние системы БЕЗОПАСНОЕ
                                  Else состояние системы НЕБЕЗОПАСНОЕ
```

Алгоритм предотвращения тупиков – алгоритм банкира

Имя процесса	Максимальная потребность	Выделено	Остаток
A	4	2	2
B	6	3	3
C	8	2	6

Имя процесса	Максимальная потребность	Выделено	Остаток
A	4	2	2
B	6	3	3
C	8	4	4

Алгоритм предотвращения тупиков – алгоритм банкира

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Матрица требований

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Матрица распределения

R1	R2	R3
9	3	6

Вектор ресурсов

R1	R2	R3
0	1	1

Вектор доступности

а) Начальное состояние

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Матрица требований

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Матрица распределения

R1	R2	R3
6	2	3

Вектор доступности

б) P2 выполнен до завершения

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

R1	R2	R3
7	2	3

Вектор доступности

Алгоритм предотвращения тупиков – алгоритм банкира

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Матрица требований

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Матрица распределения

R1	R2	R3
9	3	6

Вектор ресурсов

R1	R2	R3
1	1	2

Вектор доступности

а) Начальное состояние

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Матрица требований

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Матрица распределения

R1	R2	R3
0	1	1

Вектор доступности

Недостатки алгоритма банкира

1. Алгоритм исходит из фиксированного количества ресурсов. Ресурсы выходят из строя, находятся на ремонте. Поэтому резерв практически может и не оказаться. Количество единиц ресурсов определяется при генерации.

2. Требуется более или менее ПОСТОЯННЫЕ числа одновременно работающих процессов (некоторого статического режима). Однако в современных системах их число постоянно меняется весьма динамически. Поэтому становится проблематично уследить за заявками и текущими потребностями.

3. При существующей системе распределения ресурсов возможны длительные периоды нахождения процессов в состоянии "приостановки".

4. Для пользователя проблематично указать, какие ресурсы ему потребны. По мере повышения "дружественности" все больше пользователей, которых эти проблемы не волнуют.

Обнаружение тупиков – алгоритм Медника

Предыдущий метод гарантирует отсутствие тупиковых ситуаций за счет весьма осторожной политики. Метод обнаружения **позволяет обнаружить ситуацию тупика, когда она уже произошла**.

Не препятствуя возникновению тупика, метод предполагает его обнаружение и восстановление прерванной им нормальной работы.

Рассмотрим один из алгоритмов **обнаружения** тупиков (алгоритм Медника).

Пусть в какой-то период времени 3 процесса разделяют 5 устройств одного типа. Факты запроса устройств процессами представим в виде таблицы:

T	У1	У2	У3	У4	У5
1	Р3		Р1	Р2	
2		Р2			Р2
3	Р1				
4		Р3			
5			Р2		

Для функционирования алгоритма необходимо использование таблиц, в которых собиралась бы информация:

- о **назначении ресурсов процессам** (**РАСПРЕД**)
- о **процессах, блокированных при попытке обращения к ресурсу** (**БЛК**).

Обнаружение тупиков – алгоритм Медника

НАЧАЛО «МЕДНИК»
(*Процесс Pj запрашивает занятый ресурс Уi*)

К = РАСПРЕД (i) (*К – номер процесса владеющего ресурсом Уi*)
ЦИКЛ-ПОКА БЛК (К) и J≠К
J = К
N = БЛК (К) (*N – номер ресурса, которого ожидает*)
К = РАСПРЕД (N) (* рассматриваемый процесс*)
ВСЕ-ЦИКЛ

ЕСЛИ J = К ТУПИК
ИНАЧЕ БЛК (J) = i (*Перевести процесс Pi в состояние ожидания*)
ВСЕ-ЕСЛИ
КОНЕЦ «МЕДНИК»

Обнаружение тупиков – алгоритм Медника

T	У1	У2	У3	У4	У5
1	Р3		Р1	Р2	
2		Р2			Р2
3	Р1				
4		Р3			
5			Р2		

ТАБЛИЦА РАСПРЕДЕЛЕНИЯ РЕСУРСОВ /РАСПРЕД/

Номер ресурса	Номер процесса, которому распределен ресурс
1	3 (1)
2	2 (2)
3	1 (1)
4	2 (1)
5	2 (2)

ТАБЛИЦА БЛОКИРОВАННЫХ ПРОЦЕССОВ /БЛК/

Номер процесса	Номер ресурса, которого ожидает данный процесс
1	1 (3)
2	
3	2 (4)

Выход из тупика

Существуют два общих подхода для восстановления из тупиковых состояний:

Первый основан на **прекращении процессов**. Процессы в тупике последовательно прекращаются (уничтожаются) в некотором систематическом порядке до тех пор, пока не станет доступным достаточное количество ресурсов для устранения тупика; в худшем случае уничтожаются все процессы, первоначально находившиеся в тупике, кроме одного.

Второй выход основан на **перехвате ресурсов**. У процессов отнимается достаточное количество ресурсов и отдается процессам, находящимся в тупике, чтобы ликвидировать тупик; процессы в первом множестве остаются с выданными запросами на перехваченные у них ресурсы.

Возможно, самым практичным и простым методом является стратегия завершения, по которой первым и уничтожаются процессы с наименьшей ценой прекращения. «Ценой» прекращения процесса может быть, например:

- приоритет процесса;
- цена повторного запуска процесса и выполнение до текущей точки согласно обычным нормальным системным учетным процедурам.