Алгоритмы планирования процессов

Планирование процессов включает в себя решение следующих задач:

- определение момента времени для смены выполняемого процесса;
- выбор процесса на выполнение из очереди готовых процессов;
- переключение контекстов "старого" и "нового" процессов.

Первые две задачи решаются программными средствами, а последняя в значительной степени аппаратно

Рассмотрим две группы наиболее часто встречающихся алгоритмов:

- алгоритмы, основанные на *квантовании,* и
- алгоритмы, основанные на *приоритетах*.

алгоритмы, основанные на *квантовании*

Алгоритм, основанный на **квантовании**

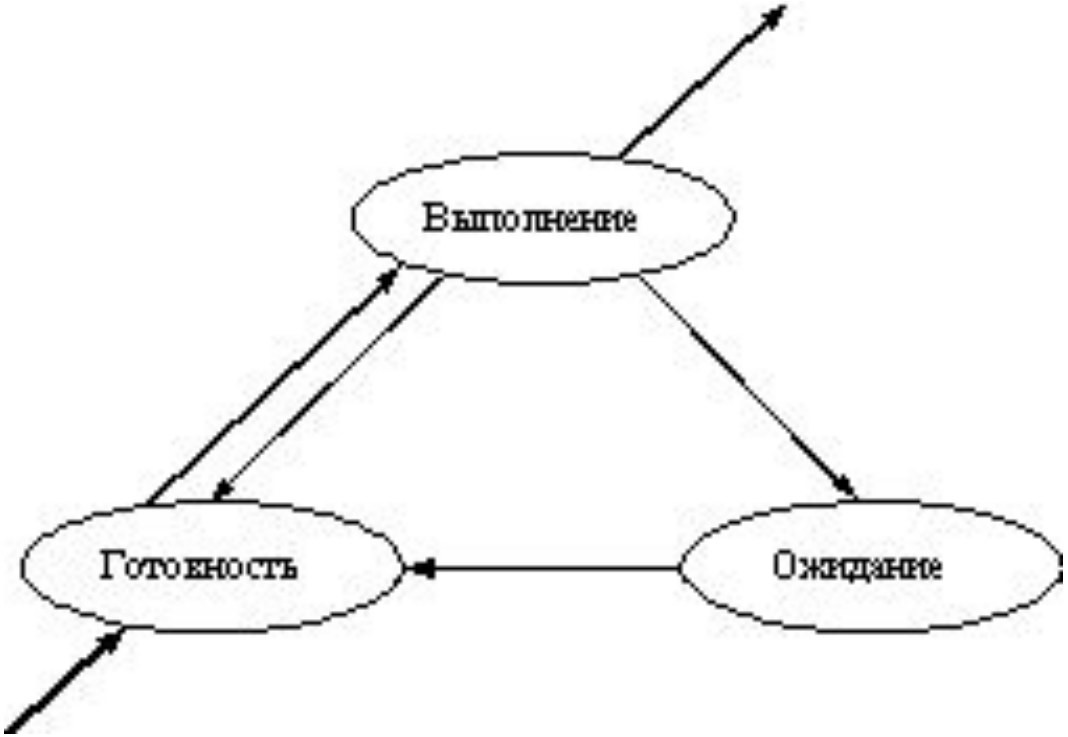
смена активного процесса происходит, если:

- процесс завершился и покинул систему,
- произошла ошибка,
- процесс перешел в состояние ОЖИДАНИЕ,
- исчерпан квант процессорного времени, отведенный данному процессу.

Алгоритм, основанный на **квантовании** (2)

- Процесс, который исчерпал свой квант переводится в состояние ГОТОВНОСТЬ и ожидает, когда ему будет предоставлен новый квант процессорного времени,
- на выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых.
- Таким образом, ни один процесс не занимает процессор надолго, поэтому квантование широко используется в системах разделения времени

.Планирование, основанное на квантовании



- . Кванты, выделяемые процессам, могут быть одинаковыми для всех процессов или различными

- Кванты, выделяемые одному процессу, могут быть фиксированной величины или изменяться в разные периоды жизни процесса.

- Процессы, которые не полностью использовали выделенный им квант (например, из-за ухода на выполнение операций ввода-вывода), могут получить или не получить компенсацию в виде привилегий при последующем обслуживании
- По разному может быть организована очередь готовых процессов: циклически, по правилу "первый пришел - первый обслужился" (FIFO) или по правилу "последний пришел - первый обслужился" (LIFO).

алгоритмы, основанные на приоритетах

Приоритет процесса

- - это число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной машины, в частности, процессорного времени: чем выше приоритет, тем выше привилегии.

Приоритет процесса

- Приоритет может выражаться целыми или дробными, положительным или отрицательным значением.
- Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях.

Приоритет

- может назначаться директивно администратором системы в зависимости от важности работы или внесенной платы,
- либо вычисляться самой ОС по определенным правилам, он может оставаться фиксированным на протяжении всей жизни процесса
- либо изменяться во времени в соответствии с некоторым законом. *В последнем случае приоритеты называются динамическими.*

Существует две разновидности приоритетных алгоритмов:

- алгоритмы, использующие **относительные приоритеты**;
- алгоритмы, использующие **абсолютные приоритеты**.

В обоих случаях

выбор процесса на выполнение из очереди готовых осуществляется одинаково:

- ***выбирается процесс, имеющий наивысший приоритет***.

- По разному решается проблема определения момента смены активного процесса.

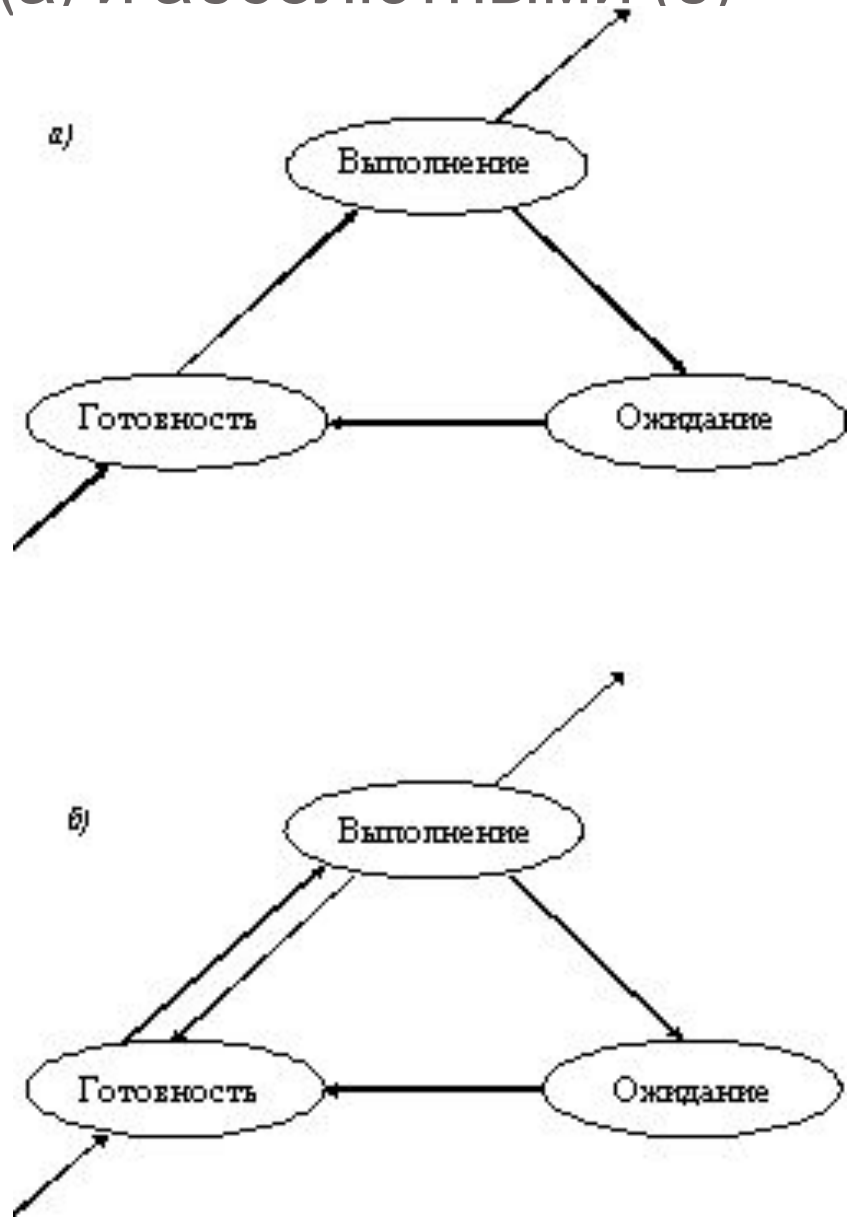
В системах с относительными приоритетами

- активный процесс выполняется до тех пор, пока он ***сам не покинет процессор***, перейдя в состояние ОЖИДАНИЕ (или же произойдет ошибка, или процесс завершится).

В системах с абсолютными приоритетами

- выполнение активного процесса прерывается еще при одном условии: ***если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса***. В этом случае прерванный процесс переходит в состояние готовности.

Графы состояний процесса для алгоритмов с относительными (а) и абсолютными (б) приоритетами.



Во многих операционных системах алгоритмы планирования

- построены с использованием как квантования, так и приоритетов.

- Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора процесса из очереди готовых определяется приоритетами процессов.

Вытесняющие и невытесняющие алгоритмы планирования

вытесняющие (preemptive) и невытесняющие (non-preemptive).

Non-preemptive multitasking

- *невытесняющая многозадачность* - это способ планирования процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

Preemptive multitasking

- *вытесняющая многозадачность* - это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей.

Основным различием

- между preemptive и non-preemptive вариантами многозадачности является **степень централизации механизма планирования задач**.

- При **вытесняющей многозадачности** механизм планирования задач целиком сосредоточен в операционной системе, и программист пишет свое приложение, не заботясь о том, что оно будет выполняться параллельно с другими задачами.

При этом операционная система выполняет следующие функции:

- определяет момент снятия с выполнения активной задачи,
- запоминает ее контекст,
- выбирает из очереди готовых задач следующую и
- запускает ее на выполнение, загружая ее контекст.

- При **невытесняющей многозадачности** механизм планирования распределен между системой и прикладными программами.

Прикладная программа, получив управление от операционной системы, сама

- определяет момент завершения своей очередной итерации и
- передает управление ОС с помощью какого-либо системного вызова,

а ОС

- формирует очереди задач и
- выбирает в соответствии с некоторым алгоритмом (например, с учетом приоритетов) следующую задачу на выполнение.

Такой механизм создает проблемы как для пользователей, так и для разработчиков.

Для пользователей это означает

- , что управление системой теряется на произвольный период времени, который определяется приложением (а не пользователем).

- ПРИМЕР: Приложение тратит слишком много времени на форматирование диска, пользователь не может переключиться с этой задачи на другую задачу, например, на текстовый редактор, в то время как форматирование продолжалось бы в фоновом режиме. Эта ситуация нежелательна, так как пользователи обычно не хотят долго ждать, когда машина завершит свою задачу.

- Поэтому разработчики приложений для **non-preemptive** операционной среды, возлагая на себя функции планировщика, должны создавать приложения так, чтобы они выполняли свои задачи небольшими частями.

- Например, *программа форматирования может отформатировать одну дорожку дискеты и вернуть управление системе*. После выполнения других задач система возвратит управление программе форматирования, чтобы та отформатировала следующую дорожку.

- Крайним проявлением "недружественности" приложения является его зависание, которое приводит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм снимет зависшую задачу с выполнения.

С другой стороны, Существенным преимуществом non-preemptive систем является более высокая скорость переключения с задачи на задачу.

- разработчик сам определяет в программе момент времени отдачи управления, то при этом исключаются нерациональные прерывания программ в "неудобные" для них моменты времени.
- Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит эти данные.

- Почти во всех современных операционных системах, ориентированных на высокопроизводительное выполнение приложений, реализована вытесняющая многозадачность.

Алгоритмы планирования процессов

Краткосрочное планирование

Долгосрочное, среднесрочное планирование

- Планирование **заданий** используется в качестве **долгосрочного планирования** процессов. Оно отвечает за порождение новых процессов в системе, определяя ее степень мультипрограммирования, т. е. количество процессов, одновременно находящихся в ней.
- Когда и какой из процессов нужно **перекачать на диск** и **вернуть обратно**, решается дополнительным промежуточным уровнем планирования процессов (Свопинг)– среднесрочным .

Краткосрочное планирование

- Планирование использования процессора применяется в качестве **краткосрочного планирования процессов.**
- Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода или просто по завершении определенного интервала времени.
- Поэтому краткосрочное планирование осуществляется, как правило, не реже одного раза в **100 миллисекунд.**

- Существует достаточно большой набор разнообразных алгоритмов планирования, которые предназначены для достижения различных целей и эффективны для разных классов задач.
- Многие из них могут использоваться на нескольких уровнях планирования.
- Рассмотрим некоторые наиболее употребительные алгоритмы применительно к процессу кратковременного планирования.

Простые алгоритмы планирования

- **FCFS (First-Come, First-Served)**
- **RR (Round Robin)**
- **SJF (Shortest-Job-First)**
- **Гарантированное планирование**
- **Приоритетное планирование**
- **Многоуровневые очереди**
- **Многоуровневые очереди с обратной связью**

Алгоритм планирования FCFS

First-Come, First-Served (FCFS)

- (первым пришел, первым обслужен).
- Процессы, находящиеся в состоянии готовность, выстроены в очередь.
- Когда процесс переходит в состояние готовность, он, а точнее, ссылка на его PCB помещается в конец этой очереди.
- Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB.
- Очередь подобного типа имеет в программировании специальное наименование – FIFO, сокращение от First In, First Out (первым вошел, первым вышел).

First-Come, First-Served (FCFS)

- алгоритм выбора процесса осуществляет невытесняющее планирование.
- Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего CPU burst .
- После этого для выполнения выбирается новый процесс из начала очереди.

Процесс	p_0	p_1	p_2
Продолжительность очередного *CPU burst*	13	4	1

Преимущества и недостатки

- Расскажите на ☺️ коллоквиуме

Алгоритм RR

Round Robin (вид детской карусели в США)

- Модификация алгоритма FCFS или сокращенно RR.
- Тот же самый алгоритм, реализованный в режиме **вытесняющего планирования**.
- Представьте все множество готовых процессов организованным циклически – процессы сидят на карусели.
- Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10 – 100 миллисекунд (см. слайд 40)
- Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

Round Robin


готовность
Процесс 1
готовность
готовность
Процесс 3
Процесс 2
исполнение
Процесс 4
Процессор
Начальный момент времени
готовность
Процесс 3
готовность
готовность
Процесс 4
Процесс 1
исполнение
Процесс 2
Процессор
По прошествии кванта

RR

- Подробный алгоритм на коллоквиуме

Алгоритм Shortest-Job-First (SJF)

Shortest-Job-First

- Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает.
- Если бы мы знали время следующих CPU burst для процессов, находящихся в состоянии готовность, то могли бы выбрать для исполнения не процесс из начала очереди, а **процесс с минимальной длительностью** CPU burst.
- Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется.

Описанный алгоритм получил название "кратчайшая работа первой" или **Shortest Job First (SJF)**.

Shortest-Job-First

- Подробный алгоритм на коллоквиуме

Гарантированное планирование

Гарантированное планирование

- При интерактивной работе N пользователей можно применить алгоритм, который гарантирует, что каждый из пользователей будет иметь в своем распоряжении ~1/N часть процессорного времени.
- Пронумеруем всех пользователей от 1 до N.

Для каждого пользователя с номером i введем две величины:

- T_i – время нахождения пользователя в системе (длительность сеанса его общения с машиной) и
- T_i - суммарное процессорное время уже выделенное всем его процессам в течение сеанса. Справедливым для пользователя было бы получение T_i/N процессорного времени.

Гарантированное планирование

- Если $\tau_i << T_i/N$

то i -й пользователь несправедливо обделен процессорным временем.

- Если же $\tau_i >> T_i/N$
- то система явно благоволит к пользователю с номером i.

- Вычислим для процессов каждого пользователя значение коэффициента справедливости

$$\tau_i N/T_i$$

- и будем предоставлять очередной квант времени готовому процессу с наименьшей величиной этого отношения.

Гарантированное планирование

- К недостаткам этого алгоритма можно отнести невозможность предугадать поведение пользователей.
- Если некоторый пользователь отправится на пару часов пообедать и поспать, не прерывая сеанса работы, то по возвращении его процессы будут получать неоправданно много процессорного времени.

Приоритетное планирование

Приоритетное планирование

- Алгоритмы SJF и гарантированного планирования представляют собой частные случаи приоритетного планирования.

При приоритетном планировании

- каждому процессу присваивается определенное числовое значение – приоритет, в соответствии с которым ему выделяется процессор.
- Процессы с одинаковыми приоритетами планируются в порядке FCFS.
- Для алгоритма SJF в качестве такого приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс.
- Для алгоритма гарантированного планирования приоритетом служит вычисленный коэффициент справедливости. Чем он меньше, тем больше у процесса приоритет .

Приоритетное планирование

- Особенности алгоритма на коллоквиуме.

Многоуровневые очереди (Multilevel Queue)

Многоуровневые очереди

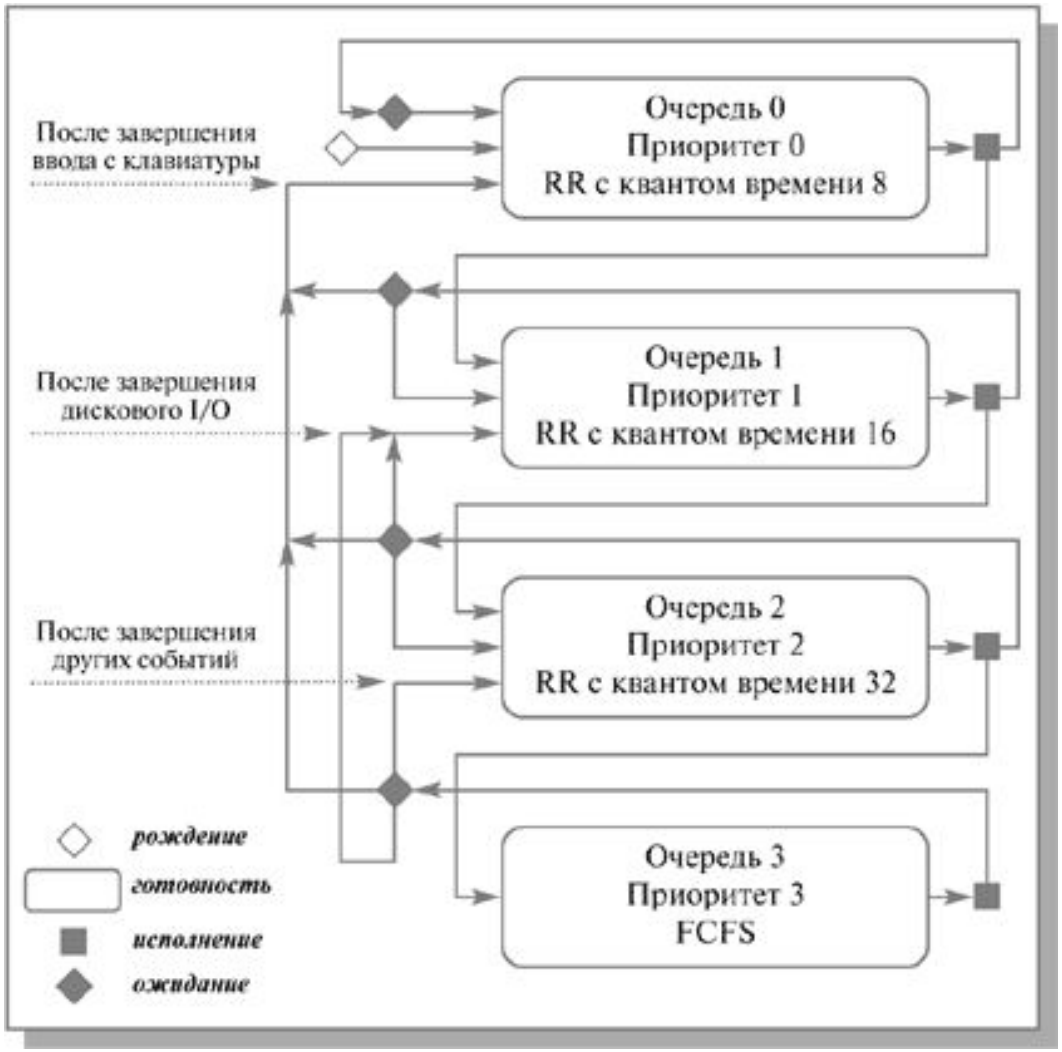
Многоуровневые очереди

- Для систем, в которых процессы могут быть легко рассортированы по разным группам, был разработан другой класс алгоритмов планирования.

- Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии готовность
- Этим очередям приписываются фиксированные приоритеты.

Многоуровневые очереди с обратной связью

Многоуровневые очереди с обратной связью



Многоуровневые очереди с обратной связью

- процесс не постоянно приписан к определенной очереди, а **может мигрировать** из одной очереди в другую **в зависимости от своего поведения.**

- На коллоквиуме приводите по примеру каждого алгоритма с таблицами и оценками качества
- Регистрируйтесь и см. http://www.intuit.ru/studies/courses/2192/31/lecture/972?page=3 .

Средства синхронизации и взаимодействия процессов

Критическая секция

Семафоры

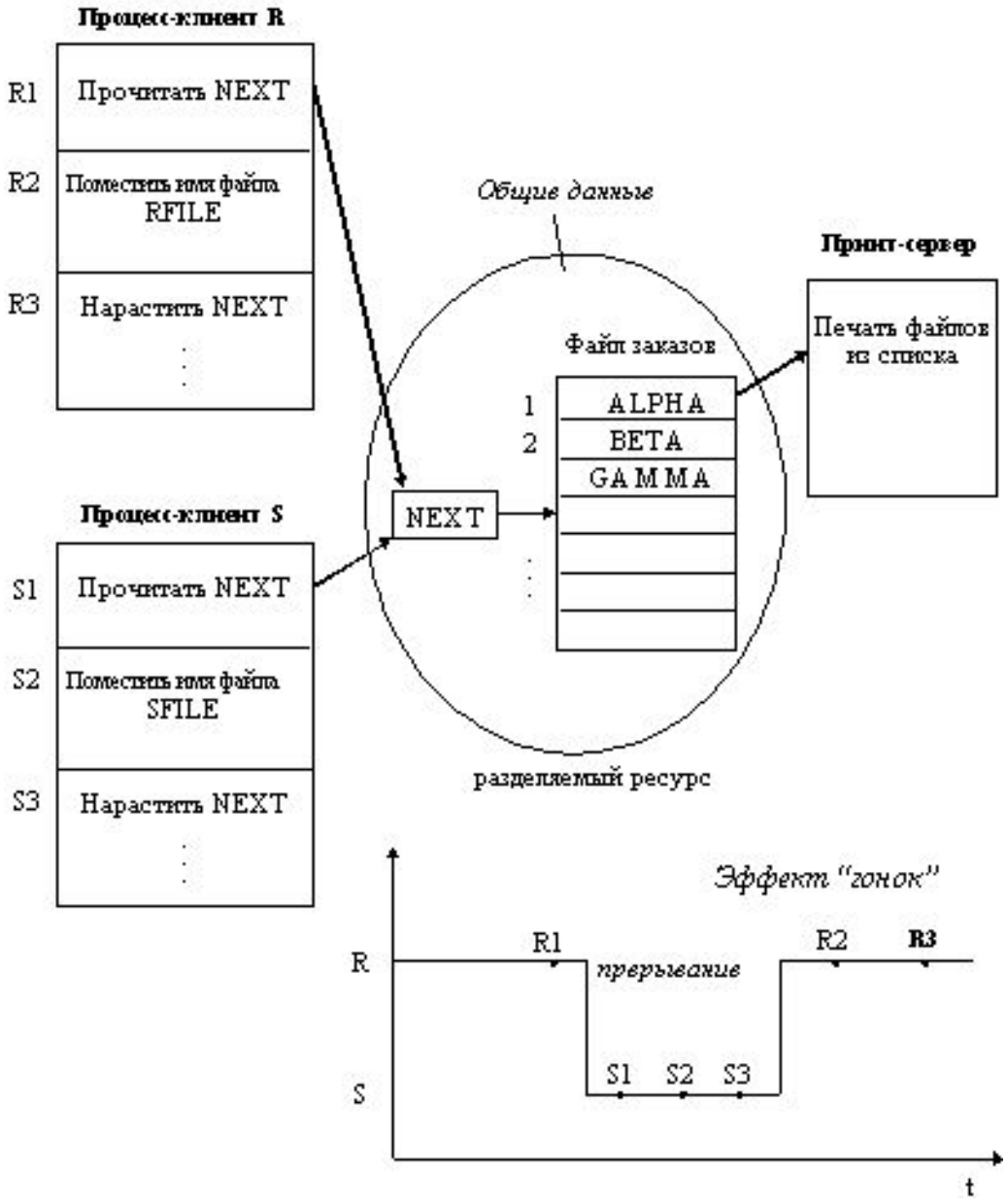
Проблема синхронизации

Процессам часто нужно взаимодействовать друг с другом, например, *один процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла*.

Во всех этих случаях возникает проблема синхронизации процессов, которая может решаться

- приостановкой и активизацией процессов,
- организацией очередей,
- блокированием и освобождением ресурсов.

программа печати файлов (принт-сервер).



- Эта программа печатает по очереди все файлы, имена которых последовательно в порядке поступления записывают в специальный общедоступный файл "заказов" другие программы.
- Особая переменная NEXT, также доступная всем процессам-клиентам, содержит номер первой свободной для записи имени файла позиции файла "заказов".
- Процессы-клиенты читают эту переменную, записывают в соответствующую позицию файла "заказов" имя своего файла и наращивают значение NEXT на единицу.

- Предположим, что в некоторый момент процесс R решил распечатать свой файл,
- для этого он прочитал значение переменной NEXT, значение которой для определенности предположим равным 4.
- Процесс запомнил это значение, но поместить имя файла не успел, так как его выполнение было прервано (например, в следствие исчерпания кванта).
- Очередной процесс S, желающий распечатать файл, прочитал то же самое значение переменной NEXT, поместил в четвертую позицию имя своего файла и нарастил значение переменной на единицу.

- Когда в очередной раз управление будет передано процессу R, то он, продолжая свое выполнение, в полном соответствии со значением текущей свободной позиции, полученным во время предыдущей итерации, запишет имя файла также в позицию 4, поверх имени файла процесса S.
- Таким образом, процесс S никогда не увидит свой файл распечатанным.

- Сложность проблемы синхронизации состоит в нерегулярности возникающих ситуаций:
- в предыдущем примере можно представить и другое развитие событий: *были потеряны файлы нескольких процессов или, напротив, не был потерян ни один файл.*

- В данном случае все определяется взаимными скоростями процессов и моментами их прерывания. Поэтому отладка взаимодействующих процессов является сложной задачей. Ситуации подобные той, когда два или более процессов обрабатывают разделяемые данные, и конечный результат зависит от соотношения скоростей процессов, называются *гонками*.

Критическая секция

Критическая секция

- *Критическая секция* - это часть программы, в которой осуществляется доступ к разделяемым 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 процесс.

- Этот прием называют взаимным исключением.

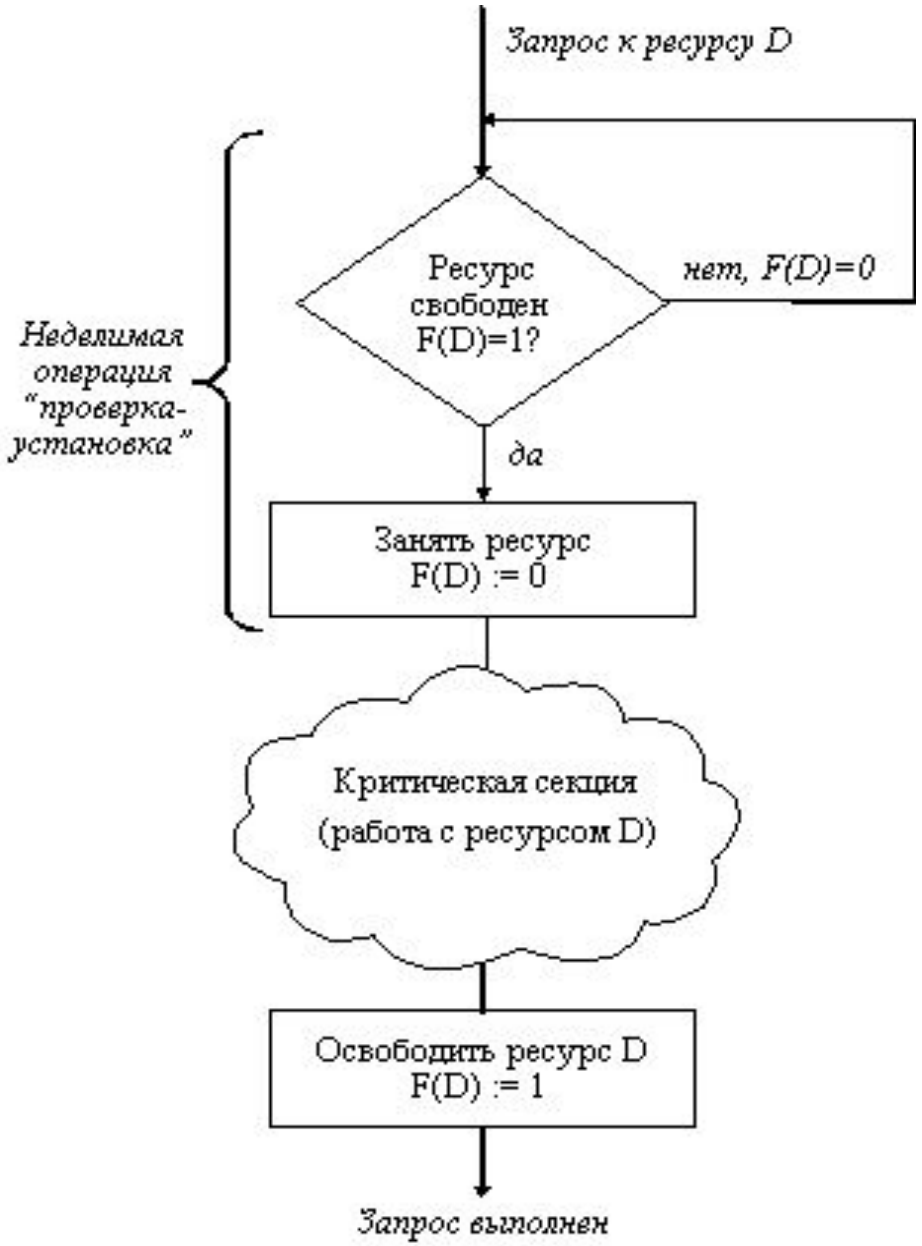
Простейший способ обеспечить взаимное исключение

- позволить процессу, находящемуся в критической секции, запрещать все прерывания.
- Однако этот способ непригоден, так как опасно доверять управление системой пользовательскому процессу; *он может надолго занять процессор, а при крахе процесса в критической области крах потерпит вся система, потому что прерывания никогда не будут разрешены*.

Другим способом

- является 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



На сл. 42 показан фрагмент алгоритма процесса,

использующего для реализации взаимного исключения доступа к разделяемому ресурсу D блокирующую переменную F(D).

- Перед входом в критическую секцию процесс проверяет, свободен ли ресурс D.
- Если он занят, то проверка циклически повторяется,
- если свободен, то значение переменной F(D) устанавливается в 0, и процесс входит в критическую секцию.
- После того, как процесс выполнит все действия с разделяемым ресурсом D, значение переменной F(D) снова устанавливается равным 1.

Если все процессы написаны с использованием вышеописанных соглашений, то взаимное исключение гарантируется.

недостатки

Реализация критических секций с использованием блокирующих переменных имеет существенный недостаток:

- в течение времени, когда один процесс находится в критической секции, другой процесс, которому требуется тот же ресурс, будет выполнять рутинные действия по опросу блокирующей переменной, **бесполезно тратя процессорное время**.

Для устранения таких ситуаций

- может быть использован так называемый **аппарат событий.**
- С помощью этого средства могут решаться не только проблемы взаимного исключения, но и более общие задачи синхронизации процессов.

В разных операционных системах аппарат событий

- реализуется по своему, но в любом случае используются системные функции аналогичного назначения, которые условно назовем WAIT(x) и POST(x), где x - идентификатор некоторого события. На рисунке 2.5 показан фрагмент алгоритма процесса, использующего эти функции. Если ресурс занят, то процесс не выполняет циклический опрос, а вызывает системную функцию WAIT(D), здесь D обозначает событие, заключающееся в освобождении ресурса D. Функция WAIT(D) переводит активный процесс в состояние ОЖИДАНИЕ и делает отметку в его дескрипторе о том, что процесс ожидает события D. Процесс, который в это время использует ресурс D, после выхода из критической секции выполняет системную функцию POST(D), в результате чего операционная система просматривает очередь ожидающих процессов и переводит процесс, ожидающий события D, в состояние ГОТОВНОСТЬ.

Семафоры

Семафоры, кто предложил

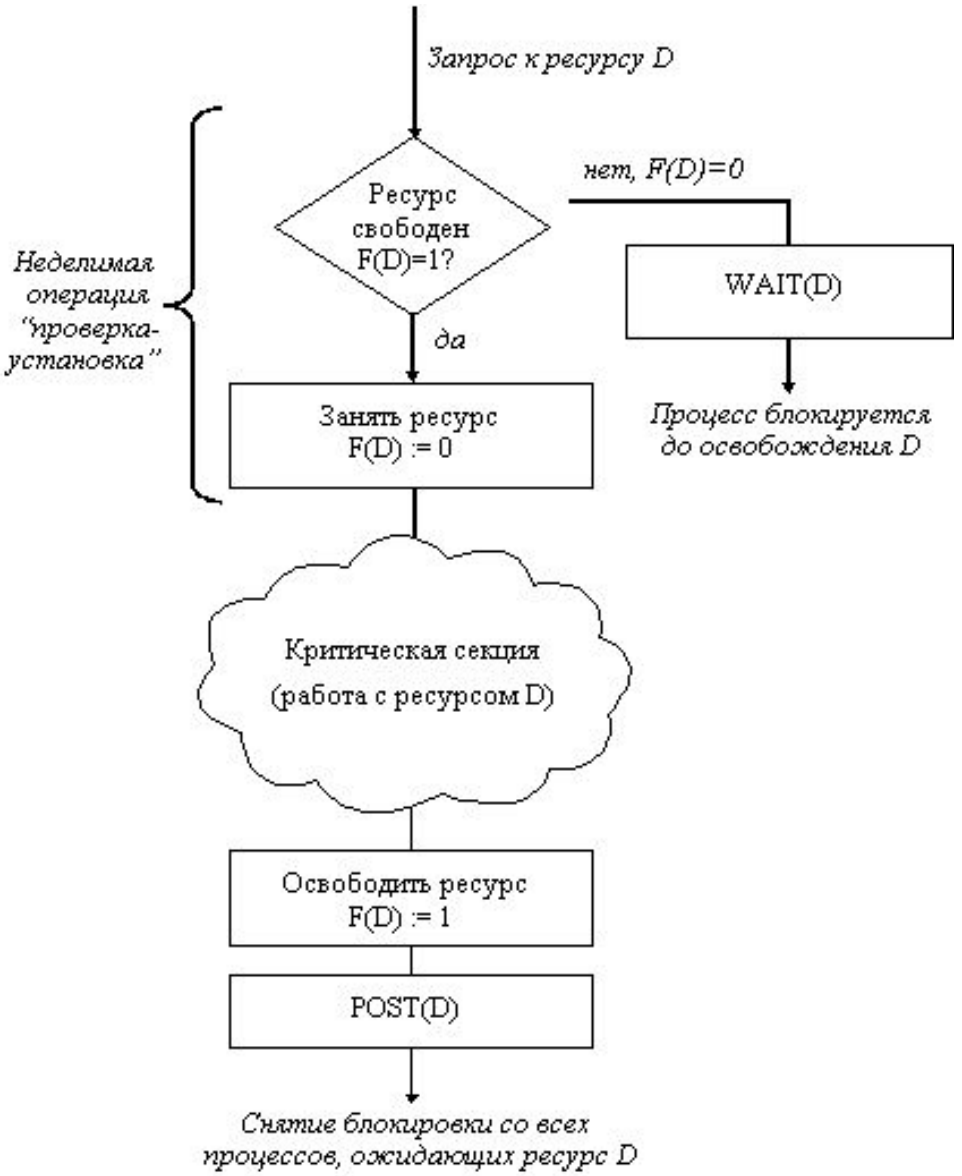
- Обобщающее средство синхронизации процессов предложил **Дейкстра,** который ввел два новых примитива.
- В абстрактной форме эти примитивы, обозначаемые **P** и **V**, оперируют над целыми **неотрицательными переменными**, называемыми ***семафорами***.

- 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.

Реализация критической секции с использованием системных функций WAIT(D) и POST(D)



В частном случае

- когда **семафор S** может принимать только значения **0 и 1**, он превращается **в блокирующую переменную.**
- 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 операцией P.

Взаимные блокировки

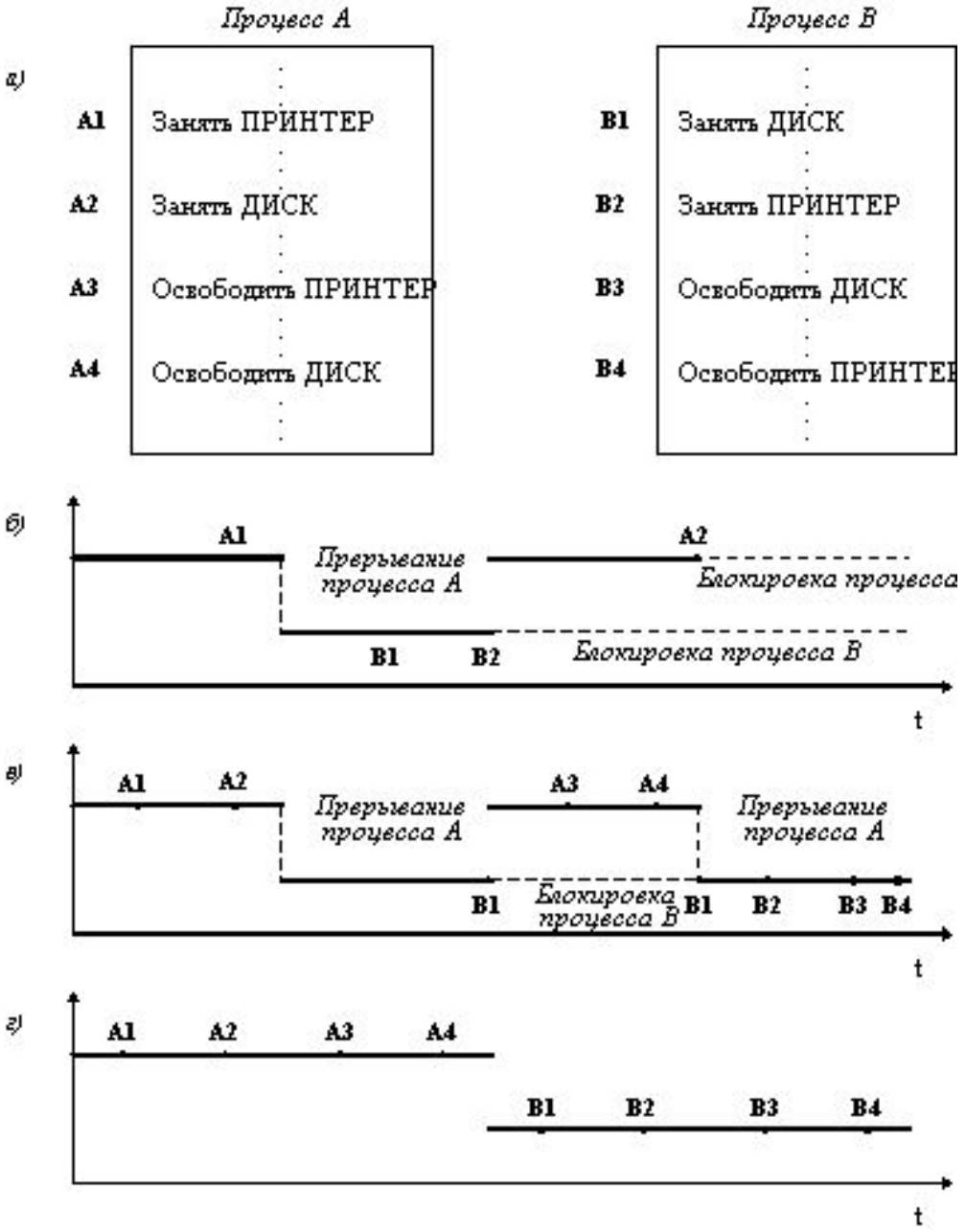
Рассмотрим примеры

```
// 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 */
}
}
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                            */
}
}
```

- Если переставить местами операции **P(e) и P(b)** в программе "писателе", то при некотором стечении обстоятельств эти два процесса могут *взаимно заблокировать друг друга*.

- Действительно, пусть "писатель" первым войдет в критическую секцию и обнаружит отсутствие свободных буферов; он начнет ждать, когда "читатель" возьмет очередную запись из буфера, но "читатель" не сможет этого сделать, так как для этого необходимо войти в критическую секцию, вход в которую заблокирован процессом "писателем".

Пример 2


а)
Процесс А
A1 Занять ПРИНТЕР
A2 Занять ДИСК
A3 Освободить ПРИНТЕР
A4 Освободить ДИСК
Процесс В
B1 Занять ДИСК
B2 Занять ПРИНТЕР
B3 Освободить ДИСК
B4 Освободить ПРИНТЕР
б)
A1
Прерывание процесса А
A2
Блокировка процесса
B1
B2
Блокировка процесса В
t
в)
A1
A2
Прерывание процесса А
A3
A4
Прерывание процесса А
B1
Блокировка процесса Б
B1
B2
B3
B4
t
г)
A1
A2
A3
A4
B1
B2
B3
B4
t

Пример 2 (см слайд 56)

- Пусть **двум процессам**, выполняющимся в режиме **мультипрограммирования**, для выполнения их работы нужно **два ресурса**, например, *принтер и диск*.
- И пусть после того, как процесс А занял принтер (*установил блокирующую переменную*), он был **прерван**.
- *Управление получил процесс **В***, который сначала занял диск, но при выполнении следующей команды был **заблокирован**, так как принтер оказался уже занятым процессом А.
- *Управление снова получил процесс* **А**, который в соответствии со своей программой сделал попытку занять диск и был **заблокирован**: диск уже распределен процессу В.
- В таком положении процессы А и В могут находиться **сколь угодно долго**.

Материал см.
http://v-ps.ru/it/studying/os-net/contents.htm

Продолжение следует...