




Синхронизация процессов: *семафоры*

Семафоры

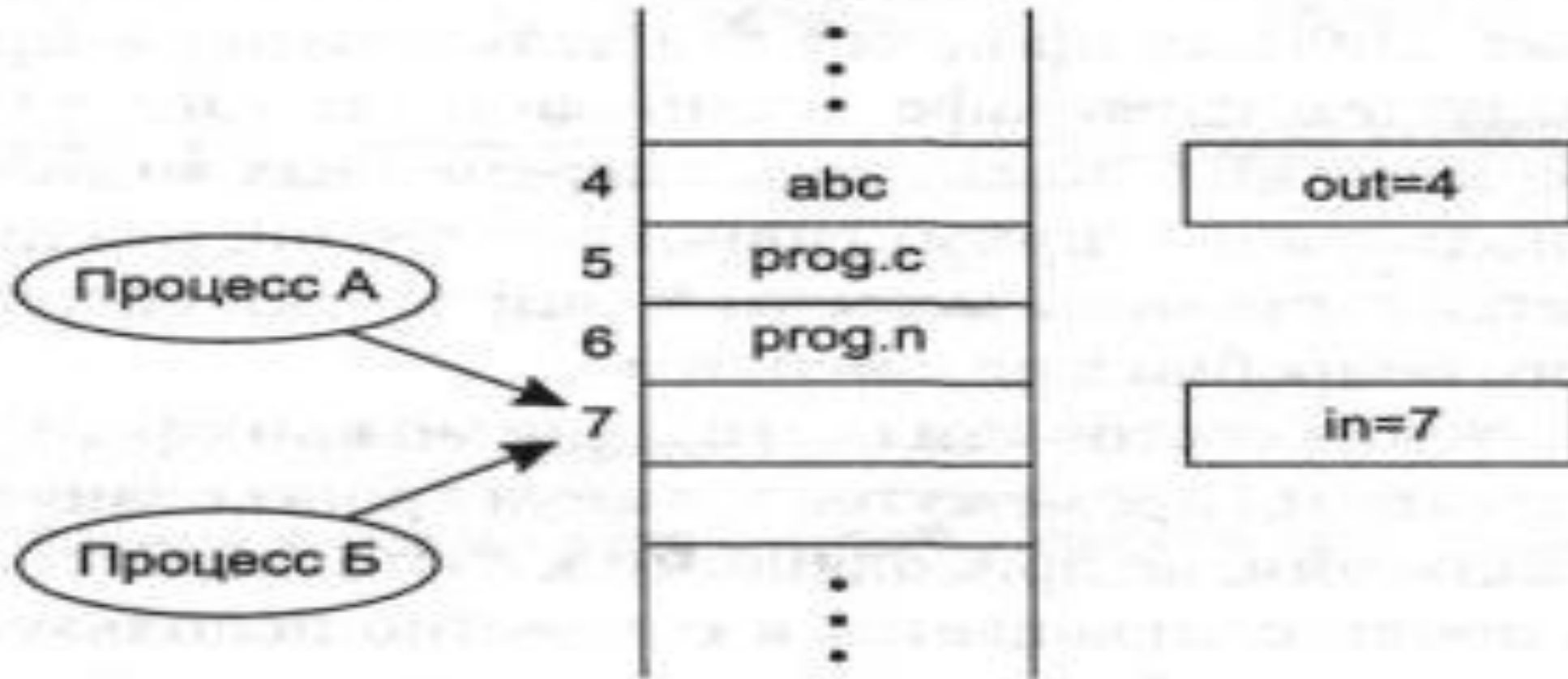
➡ Состояние состязания

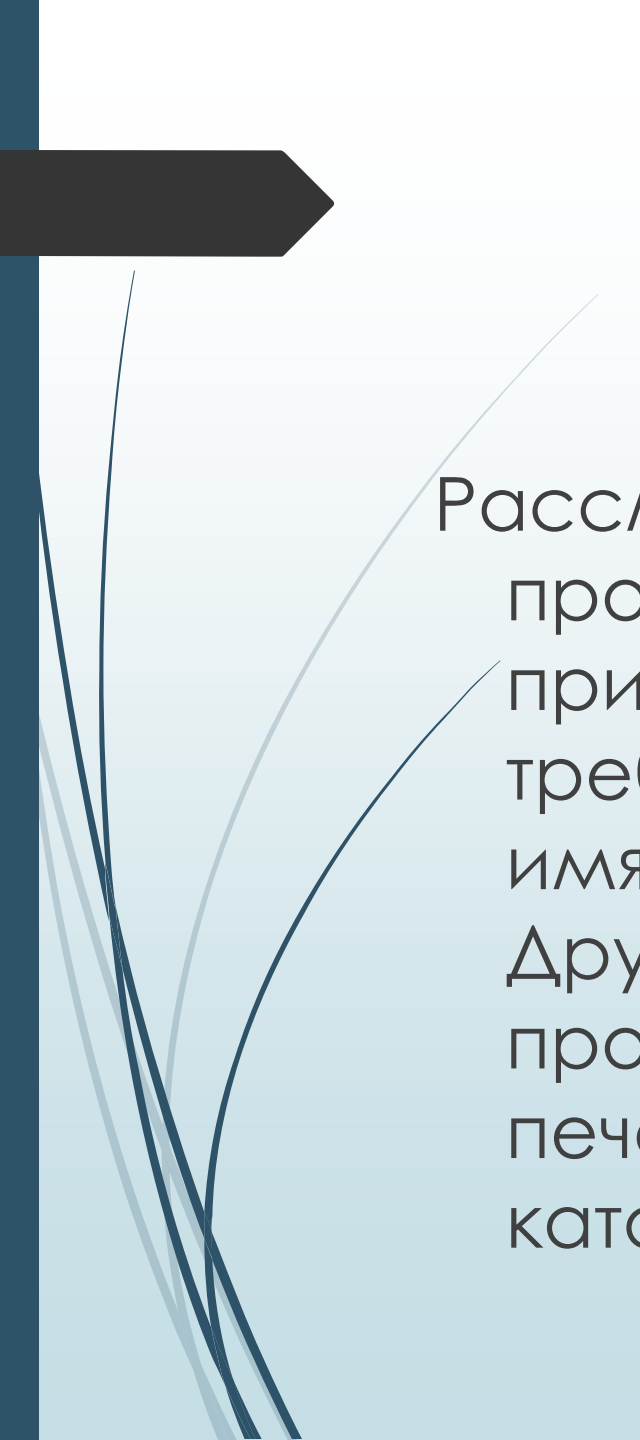
- ➡ В некоторых операционных системах процессы, работающие совместно, могут сообща использовать некое общее хранилище данных. Каждый из процессов может считывать из общего хранилища данных и записывать туда информацию. Это хранилище представляет собой участок в основной памяти (возможно, в структуре данных ядра) или файл общего доступа.




Рассмотрим межпроцессное взаимодействие на простом, но очень распространенном примере: спулер печати. Если процессу требуется вывести на печать файл, он помещает имя файла в специальный каталог спулера. Другой процесс, демон печати, периодически проверяет наличие файлов, которые нужно печатать, печатает файл и удаляет его имя из каталога.

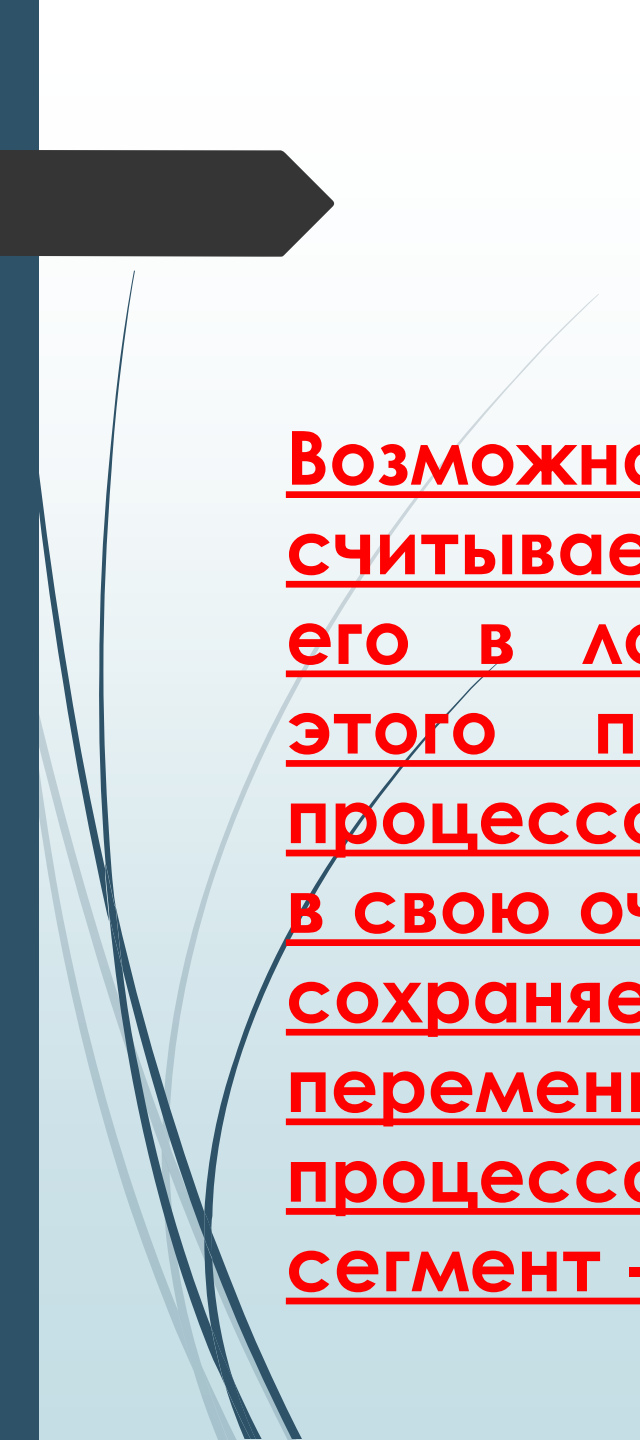
Директория спулера





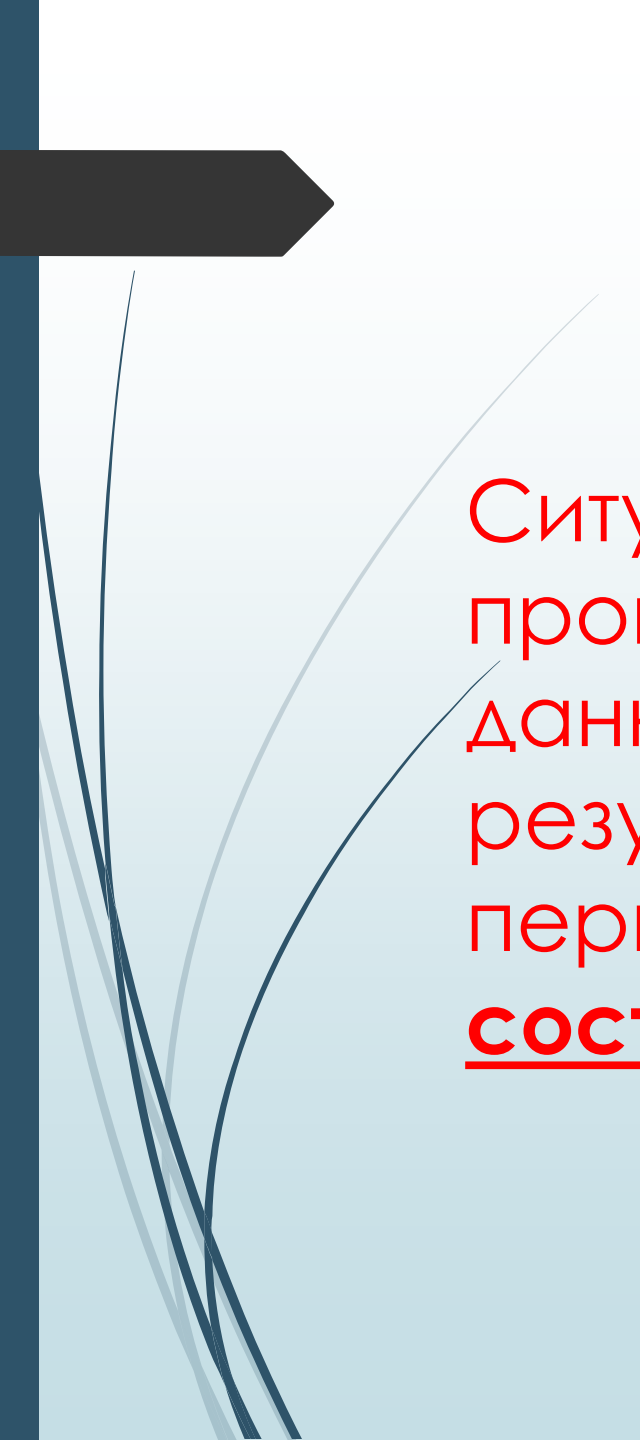
Рассмотрим межпроцессное взаимодействие на простом, но очень распространенном примере: спулер печати. Если процессу требуется вывести на печать файл, он помещает имя файла в специальный каталог спулера. Другой процесс, демон печати, периодически проверяет наличие файлов, которые нужно печатать, печатает файл и удаляет его имя из каталога.

- 
- Представьте, что каталог спулера состоит из большого числа сегментов, пронумерованных 0, 1, 2,..., в каждом из которых может храниться имя файла. Также есть две совместно используемые переменные: *out*, указывающая на следующий файл для печати, и *in*, указывающая на следующий свободный сегмент. Эти две переменные можно хранить в одном файле (состоящем из двух слов), доступном всем процессам. Пусть в данный момент сегменты с 0 по 3 пусты (эти файлы уже напечатаны), а сегменты с 4 по 6 заняты (эти файлы ждут своей очереди на печать). Более или менее одновременно процессы A и B решают поставить файл в очередь на печать.



Возможна следующая ситуация. Процесс А считывает значение (7) переменной out и сохраняет его в локальной переменной *next free slot*. После этого происходит прерывание по таймеру, и процессор переключается на процесс В. Процесс В, в свою очередь, считывает значение переменной *in* и сохраняет его (опять 7) в своей локальной переменной *next free slot*. В данный момент оба процесса считают, что следующий свободный сегмент - седьмой.

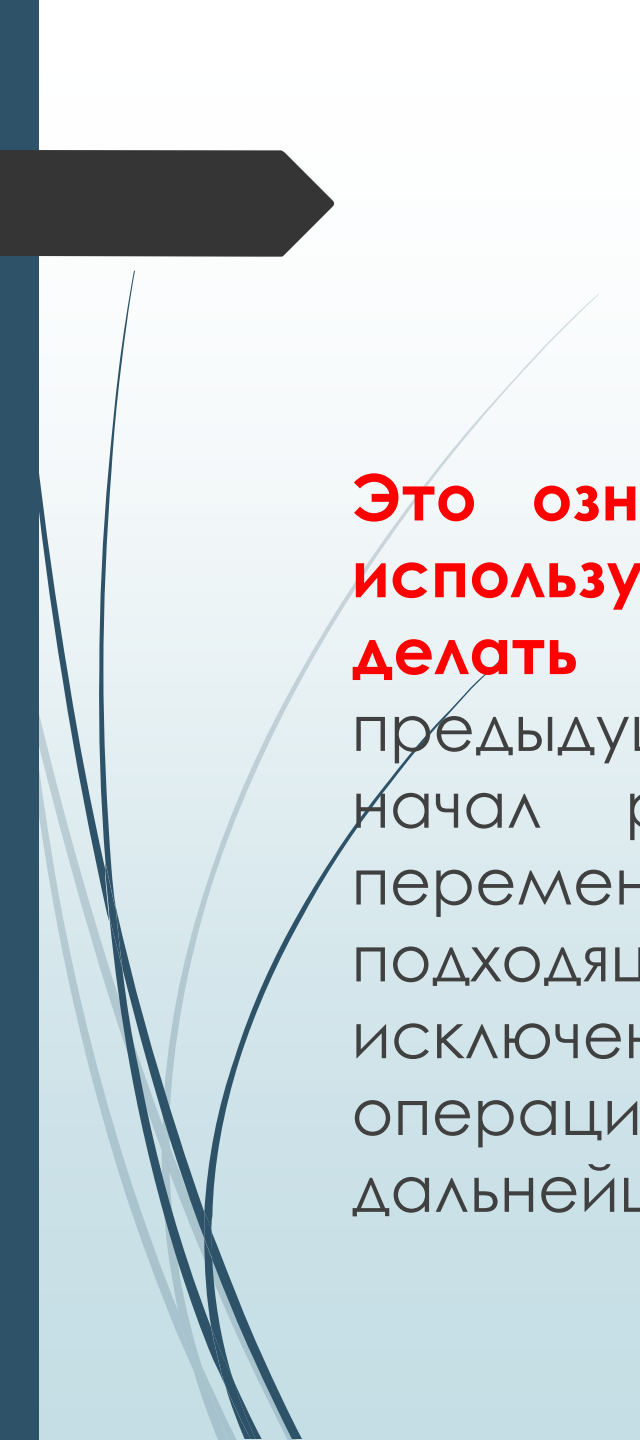
- Процесс В сохраняет в каталоге спулера имя файла и заменяет значение out на 8, затем продолжает заниматься своими задачами, не связанными с печатью.
- Наконец управление переходит к процессу А, и он продолжает с того места, на котором остановился. Он обращается к переменной *next free slot*, считывает ее значение и записывает в седьмой сегмент имя файла (разумеется, удаляя при этом имя файла, записанное туда процессом В). Затем он заменяет значение in на 8 ($next\ free\ slot + 1 = 8$). Структура каталога спулера не нарушена, так что демон печати не заподозрит ничего плохого, но файл процесса В не будет напечатан.



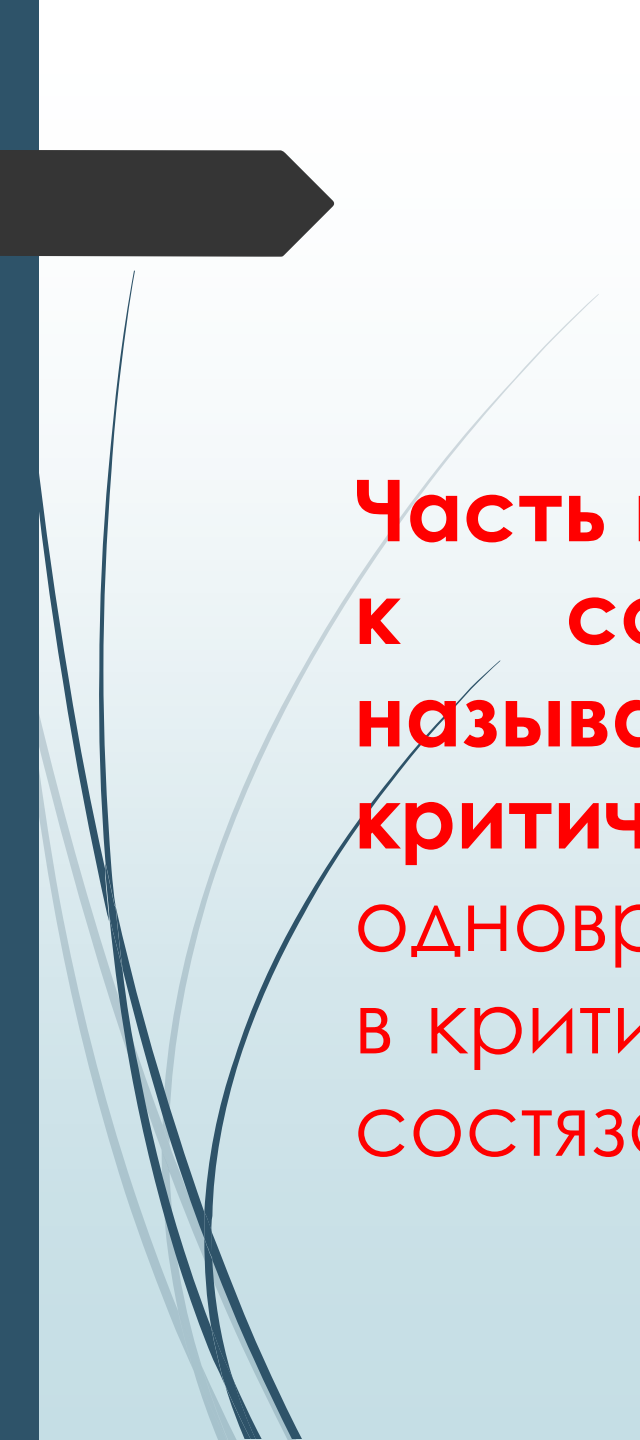
Ситуации, в которых два (и более) процесса считывают или записывают данные одновременно и конечный результат зависит от того, какой из них был первым, называются **СОСТОЯНИЯМИ СОСТЯЗАНИЯ.**

Критические области


Основным способом предотвращения проблем в этой и любой другой ситуации, связанной с совместным использованием памяти, файлов и чего-либо еще, является запрет одновременной записи и чтения разделенных данных более чем одним процессом. Говоря иными словами, необходимо взаимное исключение.



Это означает, что в тот момент, когда один процесс использует разделенные данные, другому процессу это делать будет запрещено. Проблема, описанная в предыдущем параграфе, возникла из-за того, что процесс В начал работу с одной из совместно используемых переменных до того, как процесс А ее закончил. Выбор подходящей примитивной операции, реализующей взаимное исключение, является серьезным моментом разработки операционной системы, и мы рассмотрим его подробно в дальнейшем.




Часть программы, в которой есть обращение к совместно используемым данным, называется критической областью или критической секцией. Если удастся избежать одновременного нахождения двух процессов в критических областях, мы сможем избежать состязаний.

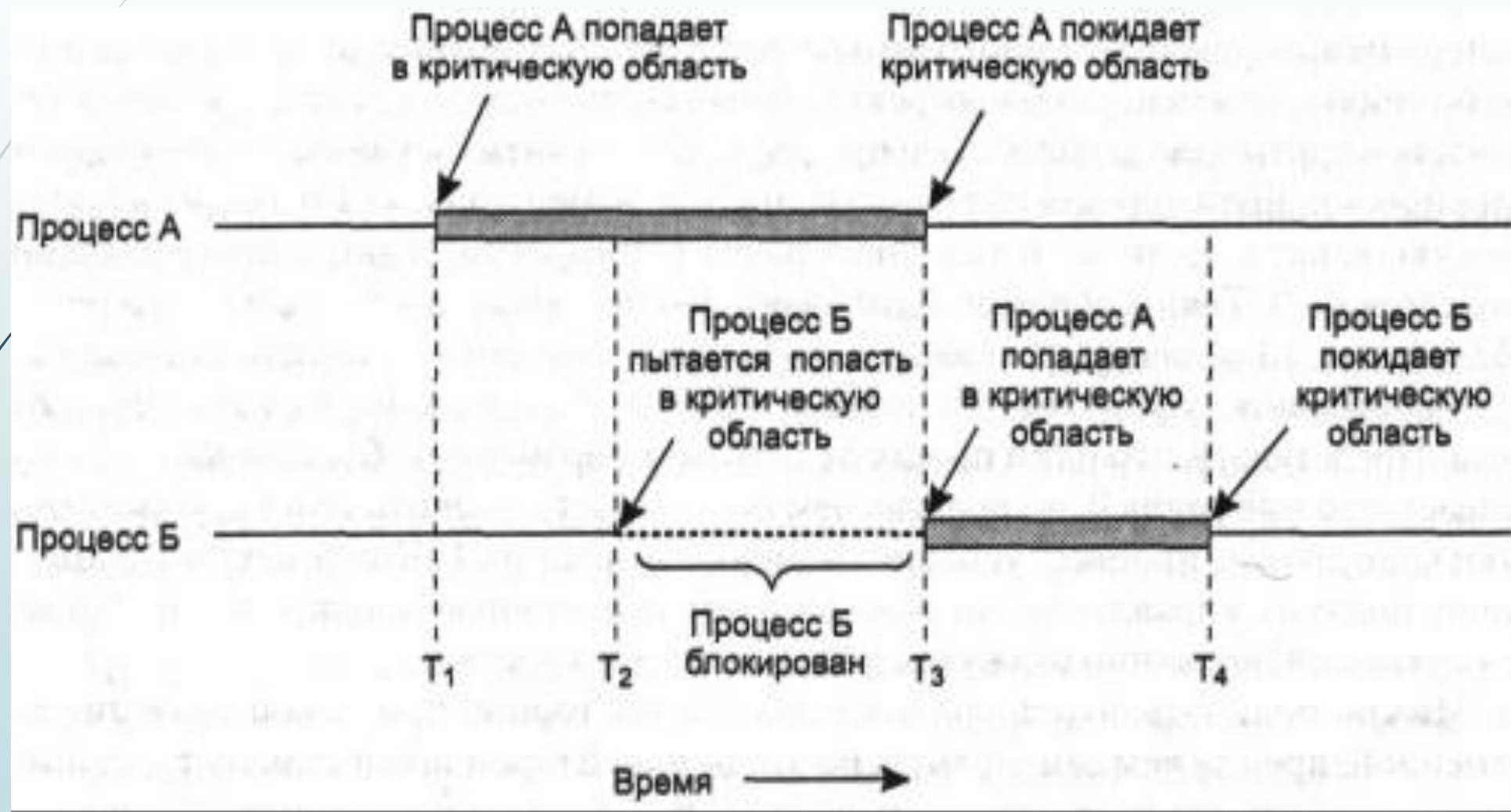


■ Несмотря на то что это требование исключает состязание, его недостаточно для правильной совместной работы параллельных процессов и эффективного использования общих данных. Для этого **необходимо выполнение четырех условий:**

- **1. Два процесса не должны одновременно находиться в критических областях.**
- **2. В программе не должно быть предположений о скорости или количестве процессоров.**
- **3. Процесс, находящийся вне критической области, не может блокировать другие процессы.**
- **4. Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.**



В абстрактном виде требуемое поведение
процессов представлено на рисунке



Процесс A попадает в критическую область в момент времени T_1 .

В момент времени T_2 , процесс B пытается попасть в критическую область, но ему это не удается, поскольку в критической области уже находится процесс A ,

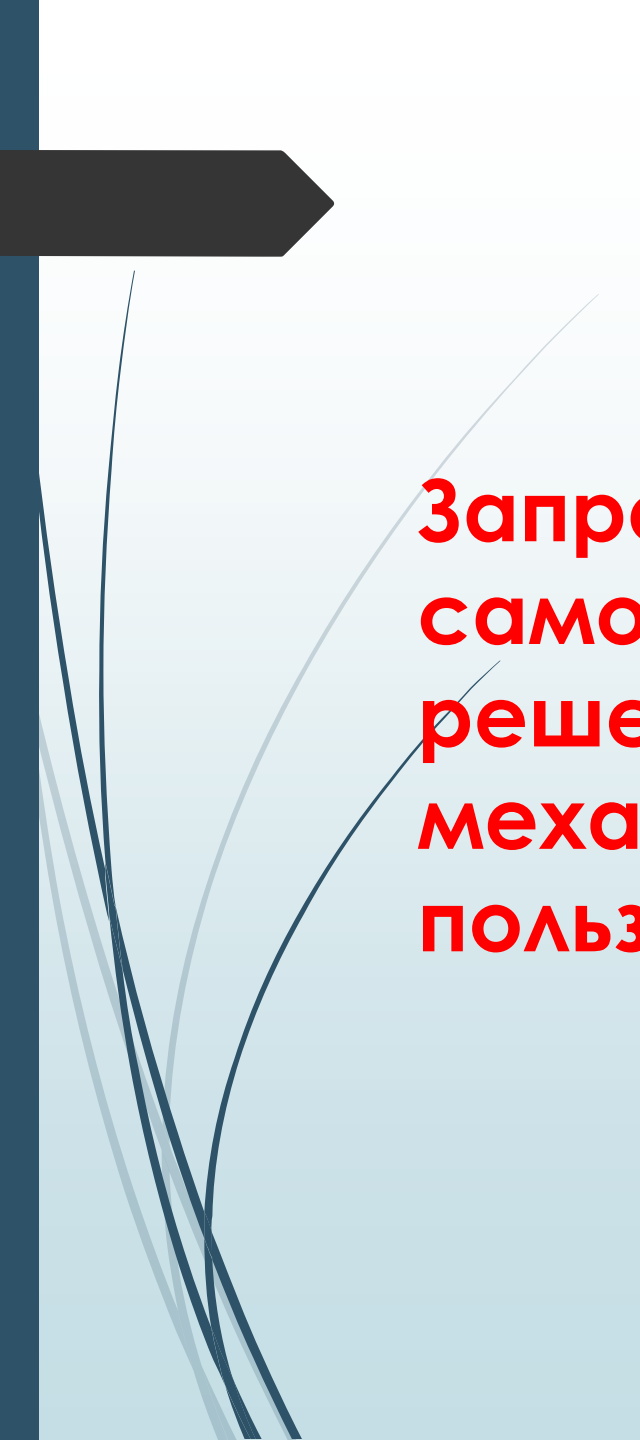
Процесс B временно приостанавливается, до наступления момента времени T_3 , когда процесс A выходит из критической области.

В момент времени T_4 процесс B также покидает критическую область, и мы возвращаемся в исходное состояние, когда ни одного процесса в критической области не было.

Взаимное исключение с активным ожиданием

Запрещение прерываний

Самое простое решение состоит в запрещении всех прерываний при входе процесса в критическую область и разрешение прерываний по выходе из области. Если прерывания запрещены, невозможно прерывание по таймеру. Поскольку процессор переключается с одного процесса на другой только по прерыванию, отключение прерываний исключает передачу процессора другому процессу. Таким образом, запретив прерывания, процесс может спокойно считывать и сохранять совместно используемые данные, не опасаясь вмешательства другого процесса.



Запрет прерываний бывает полезным в самой операционной системе, но это решение неприемлемо в качестве механизма взаимного исключения для пользовательских процессов.

Переменные блокировки

Рассмотрим одну совместно используемую переменную блокировки, изначально равную 0. Если процесс хочет попасть в критическую область, он предварительно считывает значение переменной блокировки. Если переменная равна 0, процесс изменяет ее на 1 и входит в критическую область. Если же переменная равна 1, то процесс ждет, пока ее значение сменится на 0. Таким образом, 0 означает, что ни одного процесса в критической области нет, а 1 означает, что какой-либо процесс находится в критической области.



Строгое чередование

Метод требует, чтобы два процесса попадали в критические области строго по очереди. Ни один из них не сможет попасть в критическую область (например, послать файл на печать) **два раза подряд**. Хотя этот алгоритм и исключает состояния состязания, его нельзя рассматривать всерьез, поскольку он нарушает третье условие успешной работы двух параллельных процессов с совместно используемыми данными. (рисунок ниже)

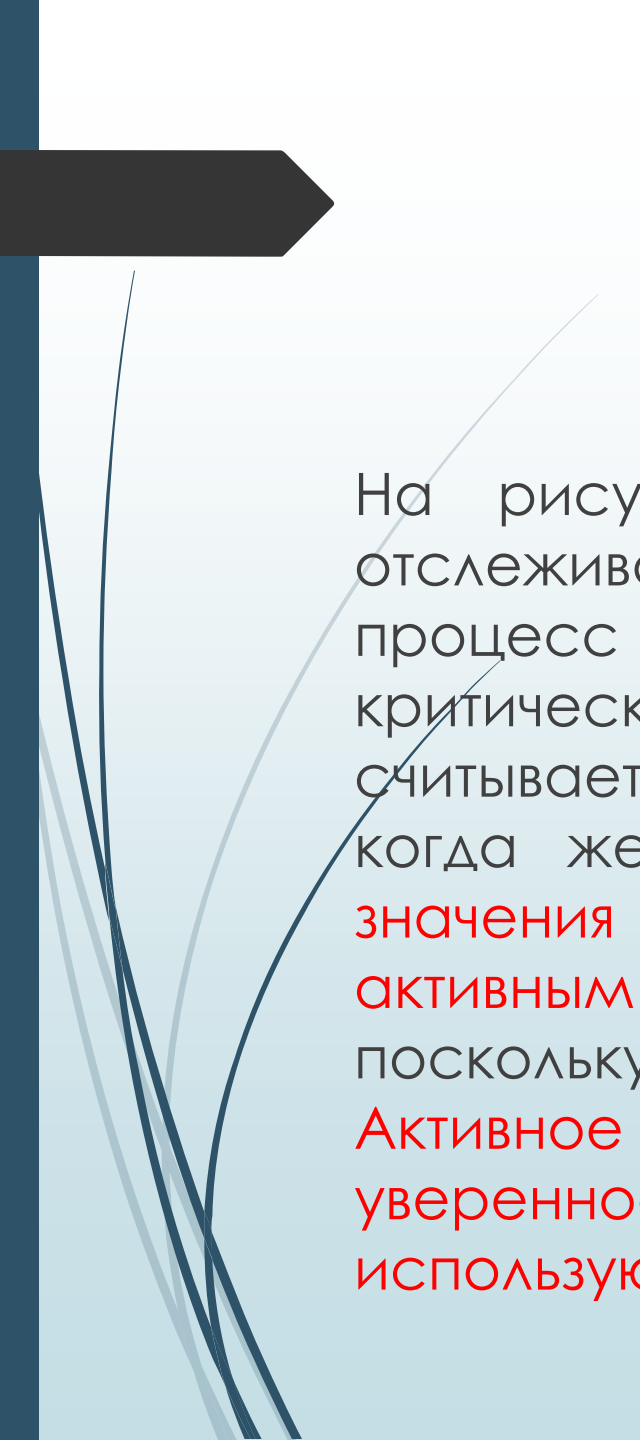
```
while(TRUE)
{while(turn!=0)      /*loop*/;
 critical_region();
 turn=1;
 noncritical_region
}
```

a

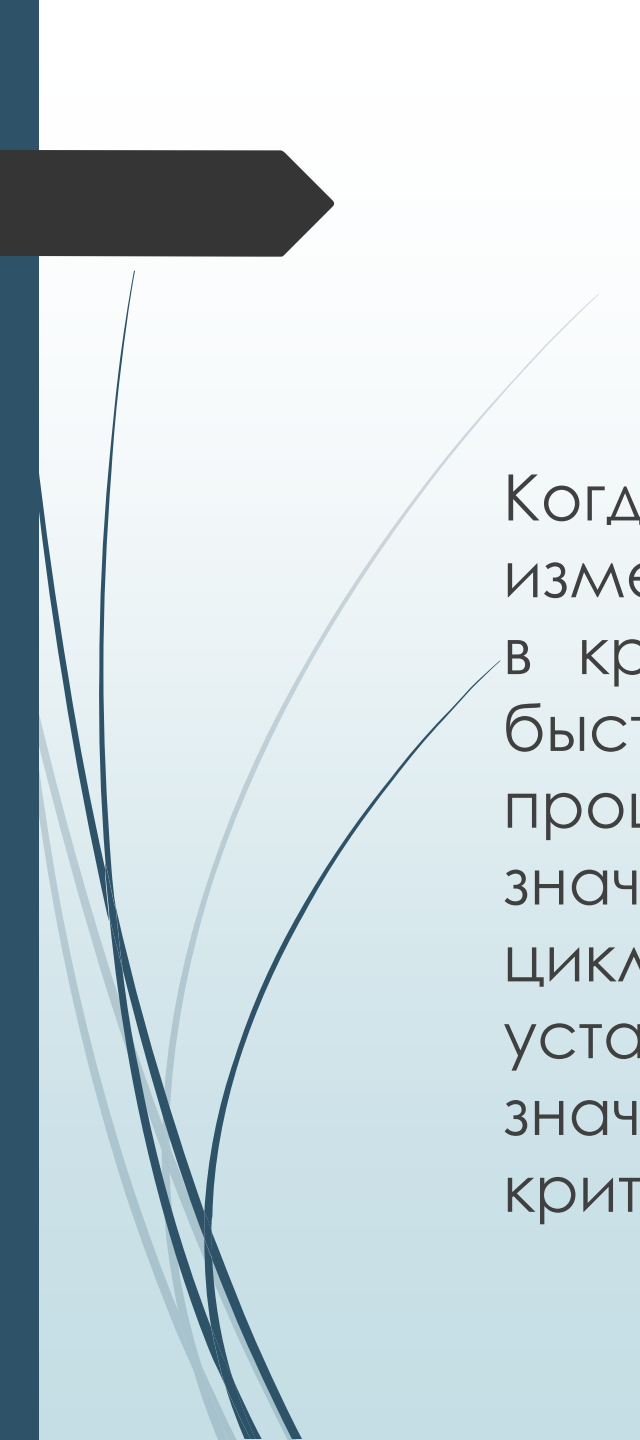
```
while(TRUE)
{while(turn!=0)      /*loop*/;
 critical_region ();
 if(page_not_in_cache(&page))
 turn=0;
 return_page(&page);
 noncritical_region ();
}
```

б

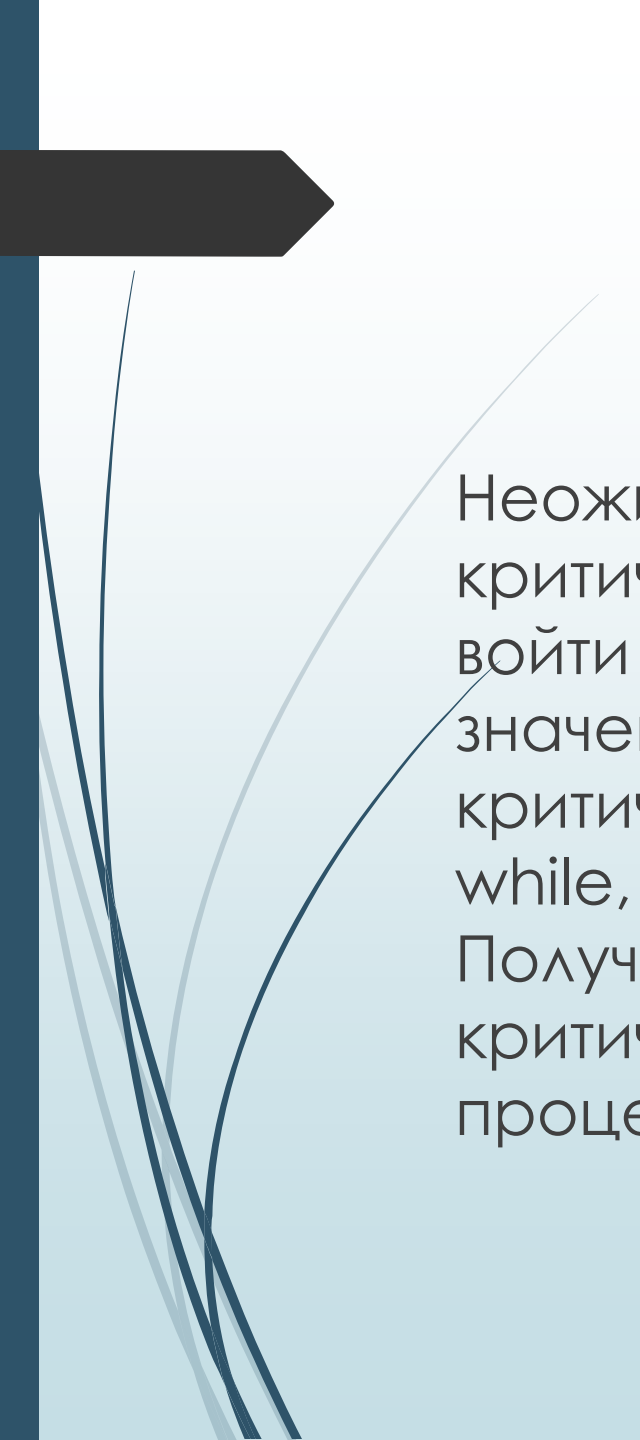
Рис. 2.16. Предлагаемое решение проблемы критической области: процесс 0 (а); процесс 1 (б). В обоих случаях необходимо удостовериться в наличии точки с запятой, ограничивающей цикл while



На рисунке целая переменная *turn*, изначально равная 0, отслеживает, чья очередь входить в критическую область. Вначале процесс 0 проверяет значение *turn*, считывает 0 и входит в критическую область. Процесс 1 также проверяет значение *turn*, считывает 0 и после этого входит в цикл, непрерывно проверяя, когда же значение *turn* будет равно 1. **Постоянная проверка значения переменной в ожидании некоторого значения называется активным ожиданием.** Подобного способа следует избегать, поскольку он является бесцельной тратой времени процессора. **Активное ожидание используется только в случае, когда есть уверенность в небольшом времени ожидания. Блокировка, использующая активное ожидание, называется спин-блокировкой.**



Когда процесс 0 покидает критическую область, он изменяет значение *turn* на 1, позволяя процессу 1 попасть в критическую область. Предположим, что процесс 1 быстро покидает свою критическую область, так что оба процесса теперь находятся вне критической области, и значение *turn* равно 0. Теперь процесс 0 выполняет весь цикл быстро, выходит из критической области и устанавливает значение *turn* равным 1. В этот момент значение *turn* равно 1, и оба процесса находятся вне критической области.

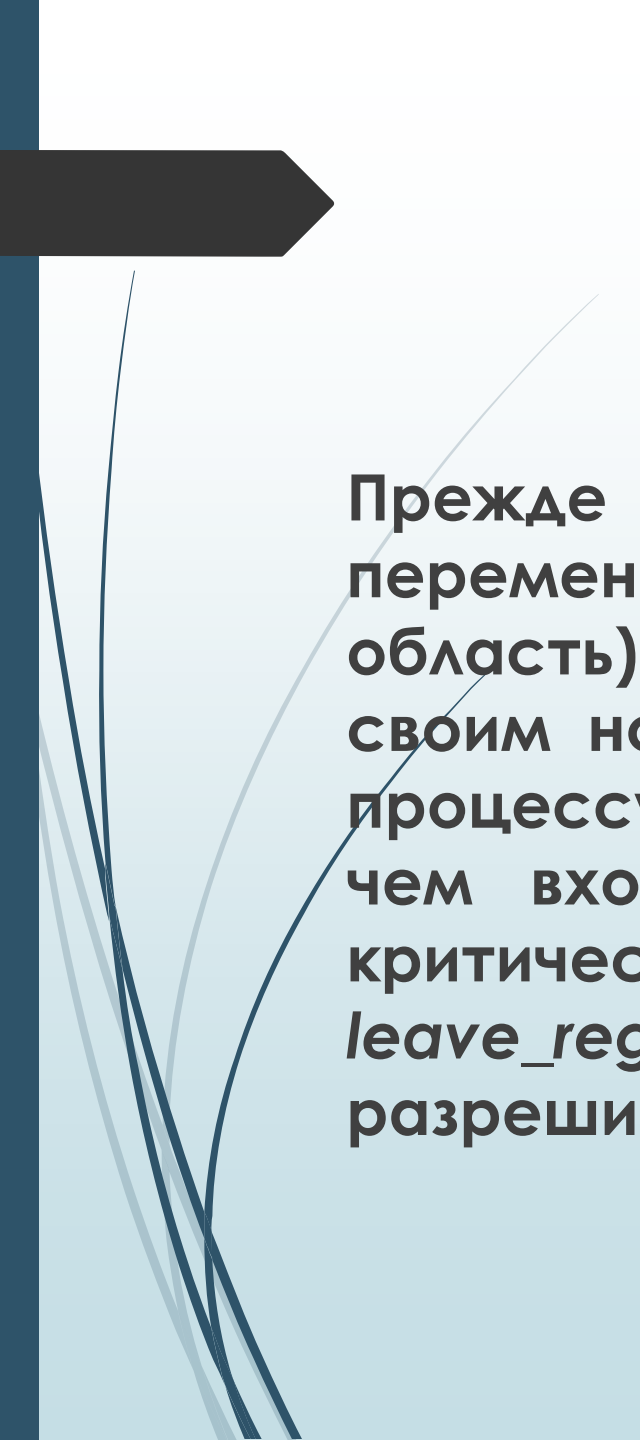


Неожиданно процесс 0 завершает работу вне критической области и возвращается к началу цикла. Но войти в критическую область он не может, поскольку значение *turn* равно 1 и процесс 1 находится вне критической области. Процесс 0 зависнет в своем цикле *while*, ожидая, пока процесс 1 изменит значение *turn* на 0. Получается, что метод поочередного доступа к критической области не слишком удачен, если один процесс существенно медленнее другого.


Алгоритм Петерсона

Листинг 2.1. Решение Петерсона для взаимного исключения


```
#define FALSE 0
#define TRUE 1
#define N 2    /* Количество процессов */
int turn;      /* Чья сейчас очередь? */
int interested[N]; /* Все переменные изначально равны 0 (FALSE) */
void enter region(int process) /* Процесс 0 или 1 */
{
    int other; /* Номер второго процесса */
    other = 1 - process; /* Противоположный процесс */
    interested[process] = TRUE; /* Индикатор интереса */
    turn = process; /* Установка флага */
    while (turn == process && interested[other] — TRUE) /* Пустой оператор */;
}
void leave region(int process) /* process; процесс, покидающий критическую область */
{
    interested[process] = FALSE; /* Индикатор выхода из критической области */
}
```




Прежде чем обратиться к совместно используемым переменным (то есть перед тем, как войти в критическую область), процесс вызывает процедуру *enter_region* со своим номером (0 или 1) в качестве параметра. Поэтому процессу при необходимости придется подождать, прежде чем входить в критическую область. После выхода из критической области процесс вызывает процедуру *leave_region*, чтобы обозначить свой выход и тем самым разрешить другому процессу вход в критическую область.



Исходно оба процесса находятся вне критических областей. Процесс 0 вызывает *enter_region*, задает элементы массива и устанавливает переменную *turn* равной 0. Поскольку процесс 1 не заинтересован в попадании в критическую область, процедура возвращается. Теперь, если процесс 1 вызовет *enter_region*, ему придется подождать, пока *interested* [0] примет значение *FALSE*, а это произойдет только в тот момент, когда процесс 0 вызовет процедуру *leave_region*, чтобы покинуть критическую область.




Представьте, что оба процесса вызвали *enter_region* практически одновременно. Оба сохраняют свои номера в *turn*. Сохранится номер того процесса, который был вторым, а предыдущий номер будет утерян. Предположим, что вторым был процесс 1, так что значение *turn* равно 1. Когда оба процесса дойдут до оператора *while*, процесс 0 войдет в критическую область, а процесс 1 останется в цикле и будет ждать, пока процесс 0 выйдет из критической области.

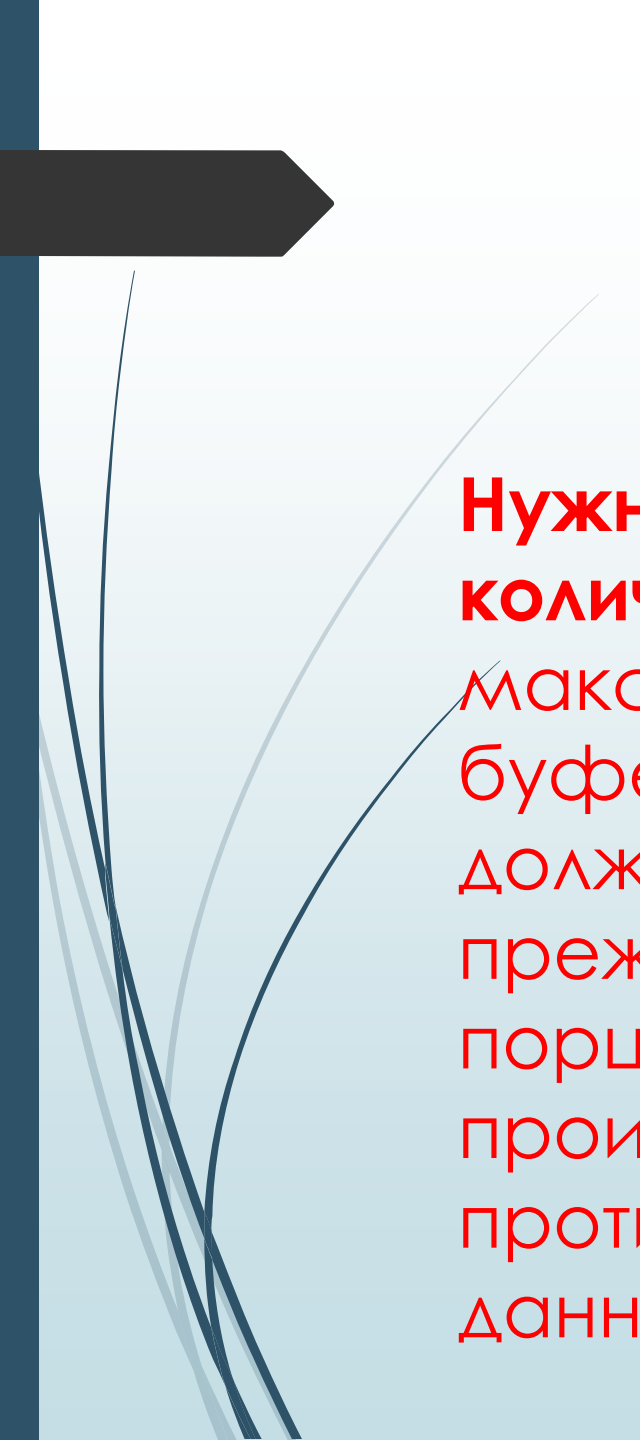


Проблема производителя и потребителя

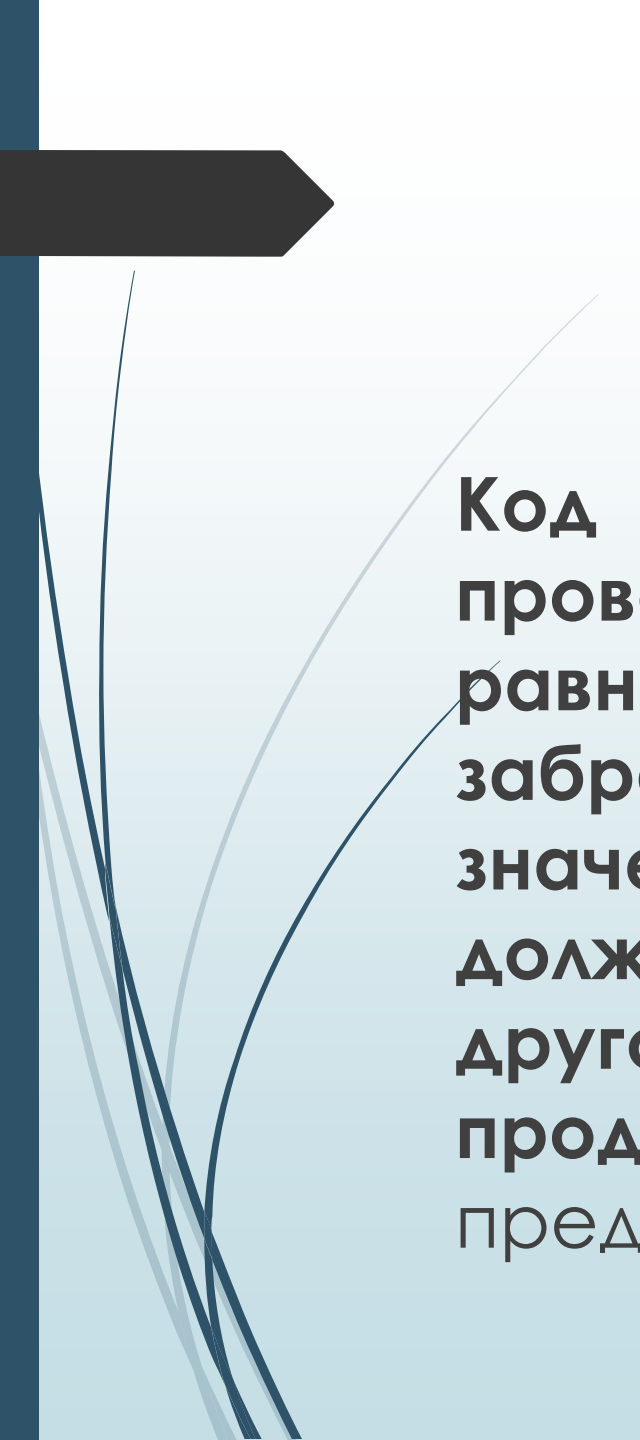
Рассмотрим проблему производителя и потребителя, также известную как проблема ограниченного буфера. Два процесса совместно используют буфер ограниченного размера. Один из них, производитель, помещает данные в этот буфер, а другой, потребитель, считывает их оттуда.



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



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




Код программы потребителя прост: сначала проверить, не равно ли значение *count* нулю. Если равно, то уйти в состояние ожидания; иначе забрать порцию данных из буфера и уменьшить значение *count*. Каждый из процессов также должен проверять, не следует ли активизировать другой процесс, и в случае необходимости проделывать это. Программы обоих процессов представлены в листинге ниже

Листинг 2.3. Проблема производителя и потребителя с неустранимым состоянием соревнования

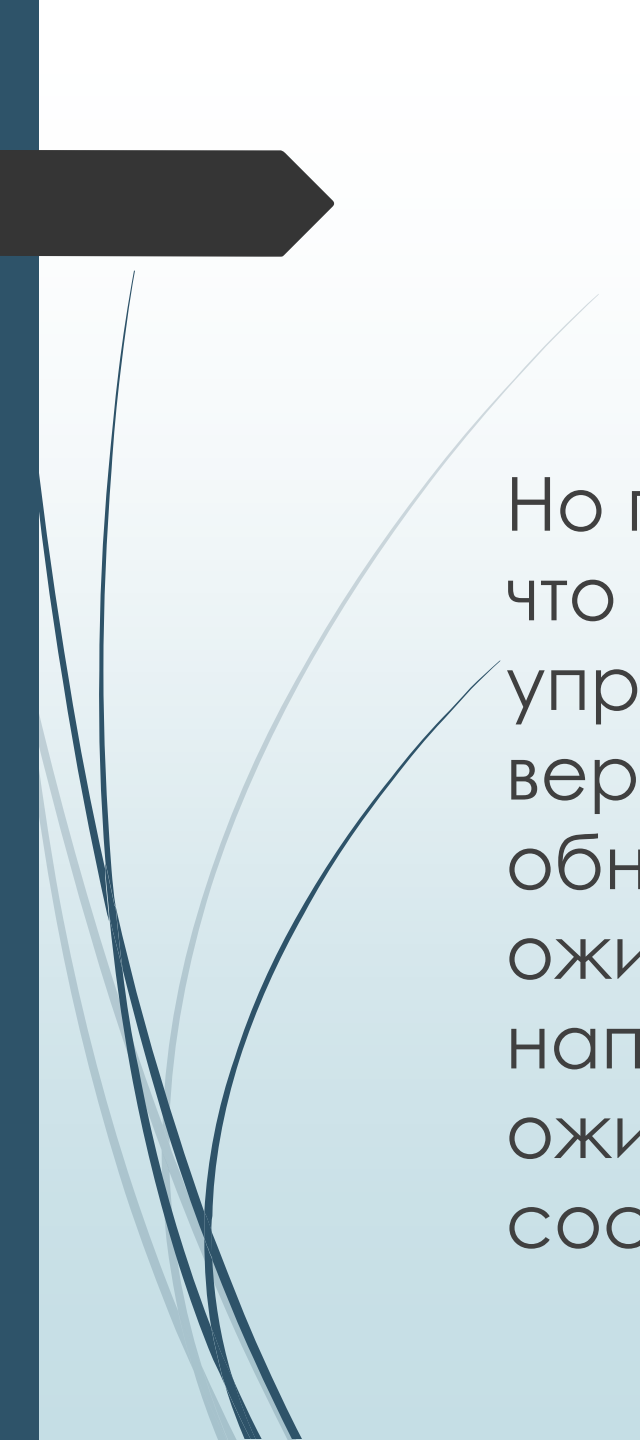
```
#define N 100 /* Максимальное количество элементов в буфере */
int count = 0; /* Текущее количество элементов в буфере */
void producer(void)
{
    int item;
    while (TRUE) /* Повторять вечно */
    {
        item = produce_item(); /* Сформировать следующий элемент */
        if (count == N) sleep(); /* Если буфер полон, уйти в состояние ожидания */
        insert_item(item); /* Поместить элемент в буфер */
        count = count + 1; /* Увеличить количество элементов в буфере */
        if (count == 1) wakeup(consumer); /* Был ли буфер пуст? */
    }
}

void consumer(void)
{
    int item;
    while (TRUE) /* Повторять вечно */
    {
        if (count == 0) sleep(); /* Если буфер пуст, уйти в состояние ожидания */
        item = remove_item(); /* Забрать элемент из буфера */
        count = count - 1; /* Уменьшить счетчик элементов в буфере */
        if (count == N - 1) wakeup(producer); /* Был ли буфер полон? */
        consume_item(item); /* Отправить элемент на печать */
    }
}
```




Состояние состязания возможно, поскольку доступ к переменной *count* не ограничен. Может возникнуть следующая ситуация: буфер пуст, и потребитель только что считал значение переменной *count*, чтобы проверить, не равно ли оно нулю.

В этот момент планировщик передал управление производителю, производитель поместил элемент в буфер и увеличил значение *count*, проверив, что теперь оно стало равно 1. Зная, что перед этим оно было равно 0 и потребитель находился в состоянии ожидания, производитель активизирует его с помощью вызова *wakeup*.



Но потребитель не был в состоянии ожидания, так что сигнал активизации пропал впустую. Когда управление перейдет к потребителю, он вернется к считанному когда-то значению *count*, обнаружит, что оно равно 0, и уйдет в состояние ожидания. Рано или поздно производитель наполнит буфер и также уйдет в состояние ожидания. Оба процесса так и останутся в этом состоянии.

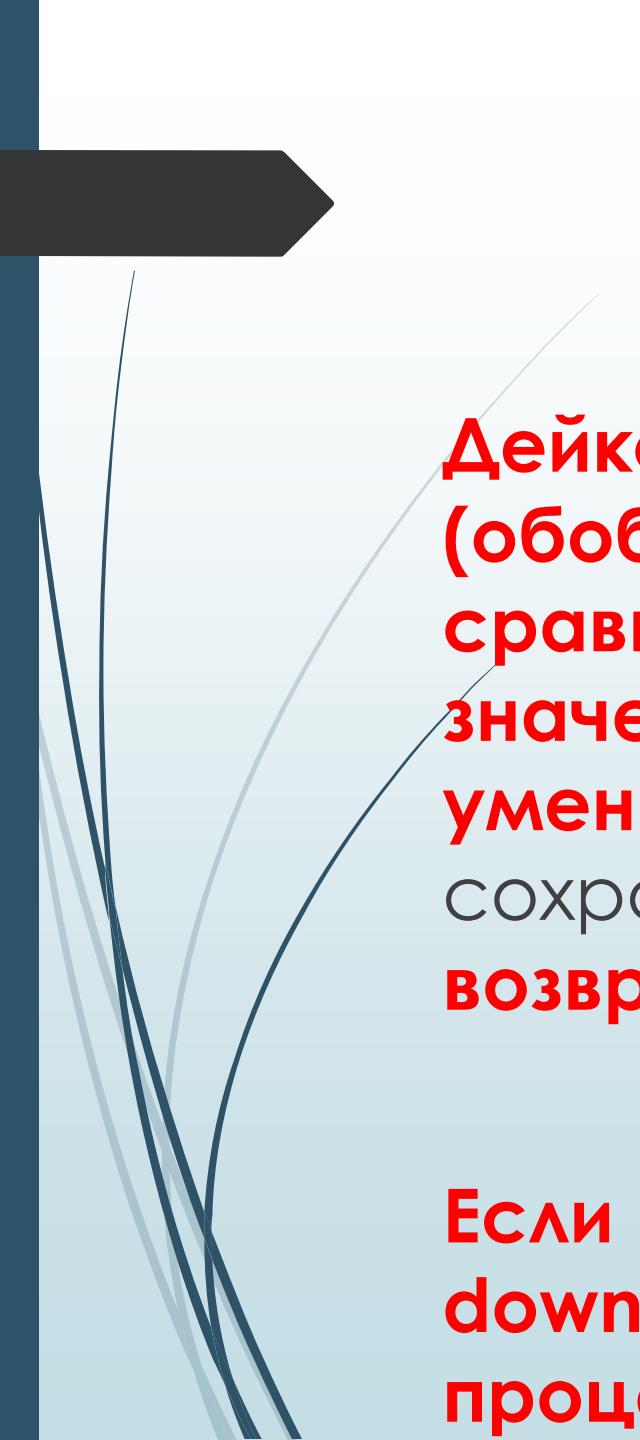


Суть проблемы в данном случае состоит в том, что сигнал активизации, пришедший к процессу, не находящемуся в состоянии ожидания, пропадает. Если бы не это, проблемы бы не было. **Быстрым решением может быть добавление бита ожидания активизации.** Если сигнал активизации послан процессу, не находящемуся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

Семафоры

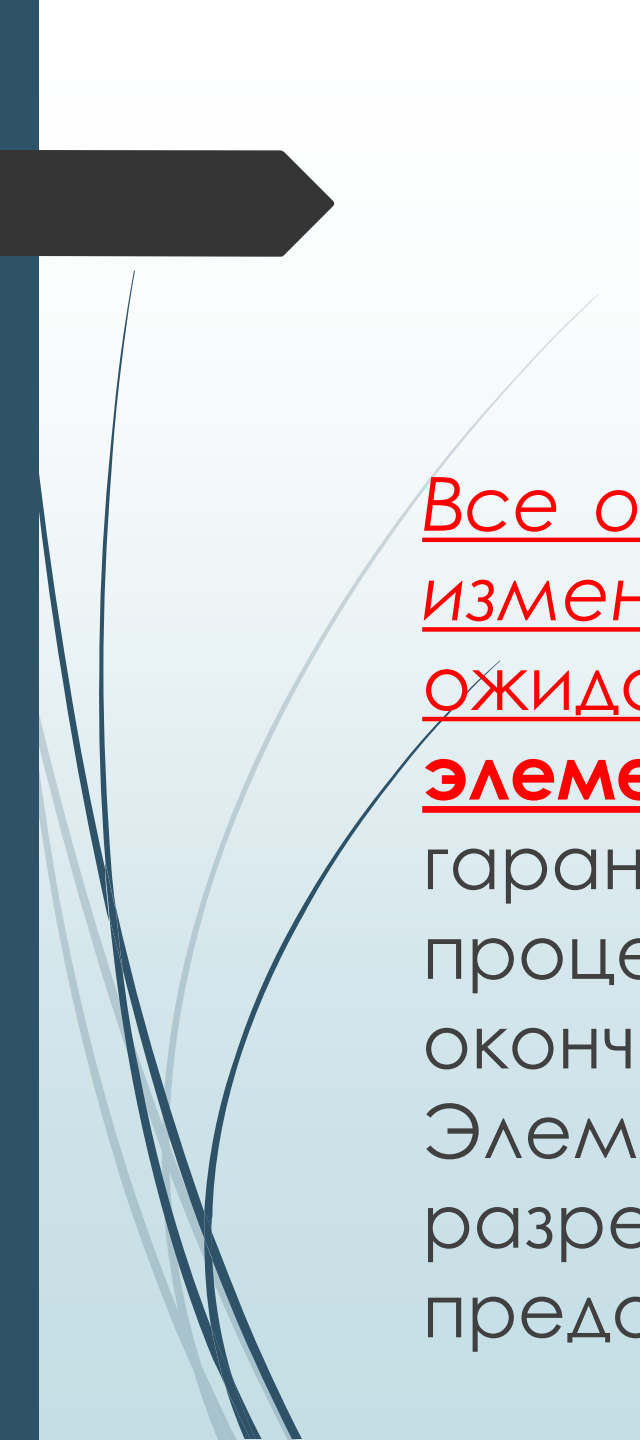
В 1965 году Дейкстра (E. W. Dijkstra) предложил использовать целую переменную для подсчета сигналов запуска, сохраненных на будущее.

Им был предложен новый тип переменных, так называемые семафоры, значение которых может быть нулем (в случае отсутствия сохраненных сигналов активизации) или некоторым положительным числом, соответствующим количеству отложенных активизирующих сигналов.




Дейкстра предложил две операции, down и up (обобщения sleep и wakeup). Операция down сравнивает значение семафора с нулем. Если значение семафора больше нуля, операция down **уменьшает его** (то есть расходует один из сохраненных сигналов активации) **и просто возвращает управление.**

Если значение семафора равно нулю, процедура down не возвращает управление процессу, а процесс переводится в состояние ожидания.

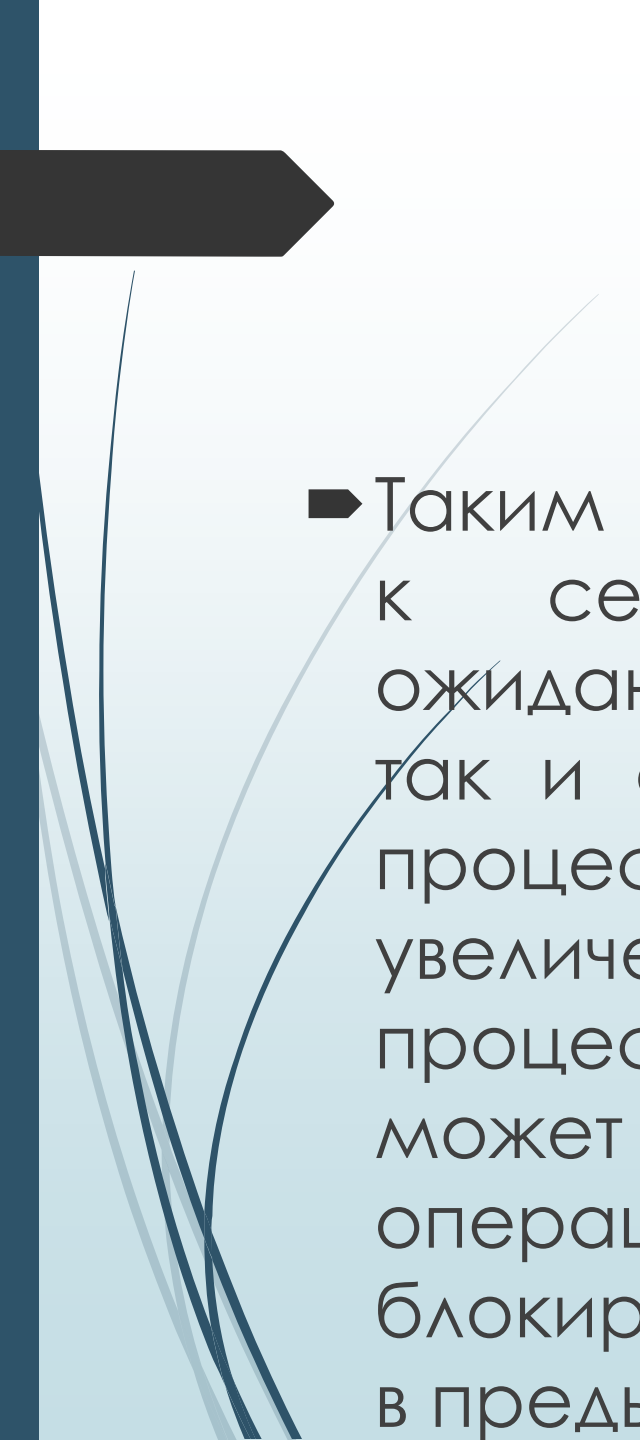



Все операции проверки значения семафора, его изменения и перевода процесса в состояние ожидания выполняются как единое и неделимое **элементарное действие.** Тем самым

гарантируется, что после начала операции ни один процесс не получит доступа к семафору до окончания или блокирования операции. Элементарность операции чрезвычайно важна для разрешения проблемы синхронизации и предотвращения состояния состязания.



Операция up увеличивает значение семафора. Если с этим семафором связаны один или несколько ожидающих процессов, которые не могут завершить более раннюю операцию down, один из них выбирается системой (например, случайным образом) и ему разрешается завершить свою операцию down.

- 
- Таким образом, после операции `up`, примененной к семафору, связанному с несколькими ожидающими процессами, значение семафора так и останется равным 0, но число ожидающих процессов уменьшится на единицу. Операция увеличения значения семафора и активизации процесса тоже неделима. Ни один процесс не может быть блокирован во время выполнения операции `up`, как ни один процесс не мог быть блокирован во время выполнения операции `wakeup` в предыдущей модели.

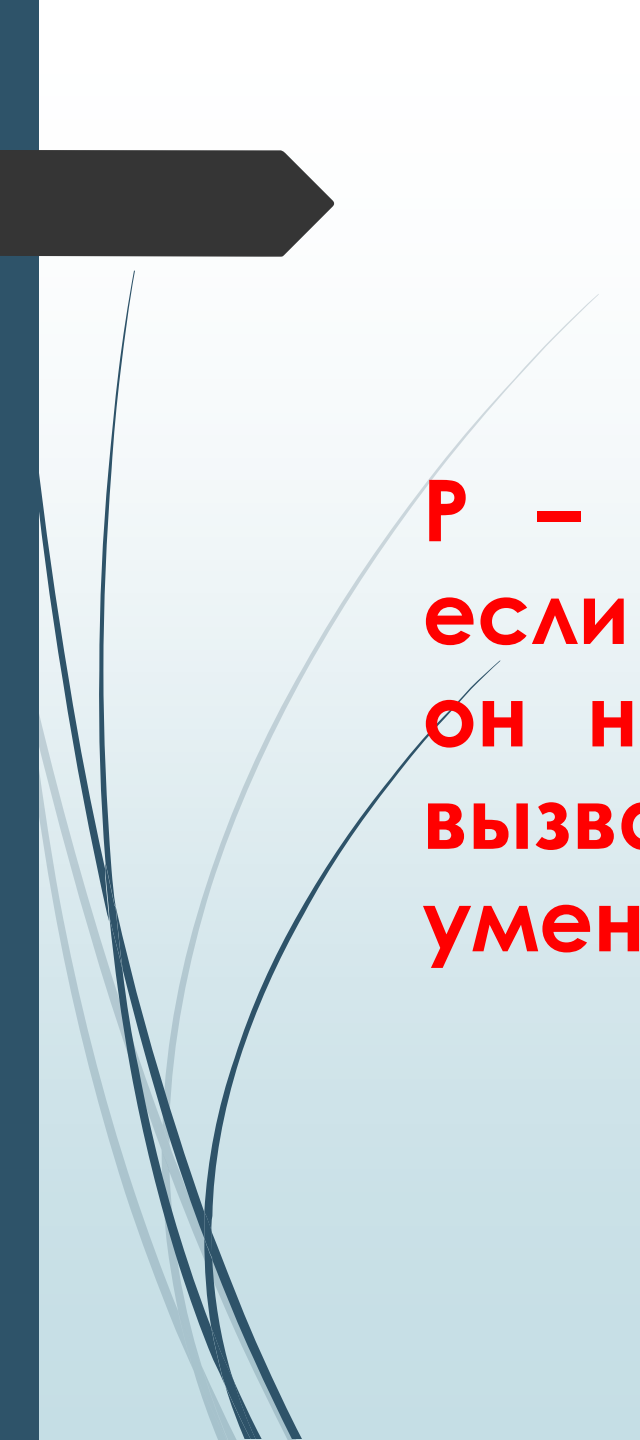


В оригинале Дейкстра использовал вместо
down и up обозначения P и V
соответственно.

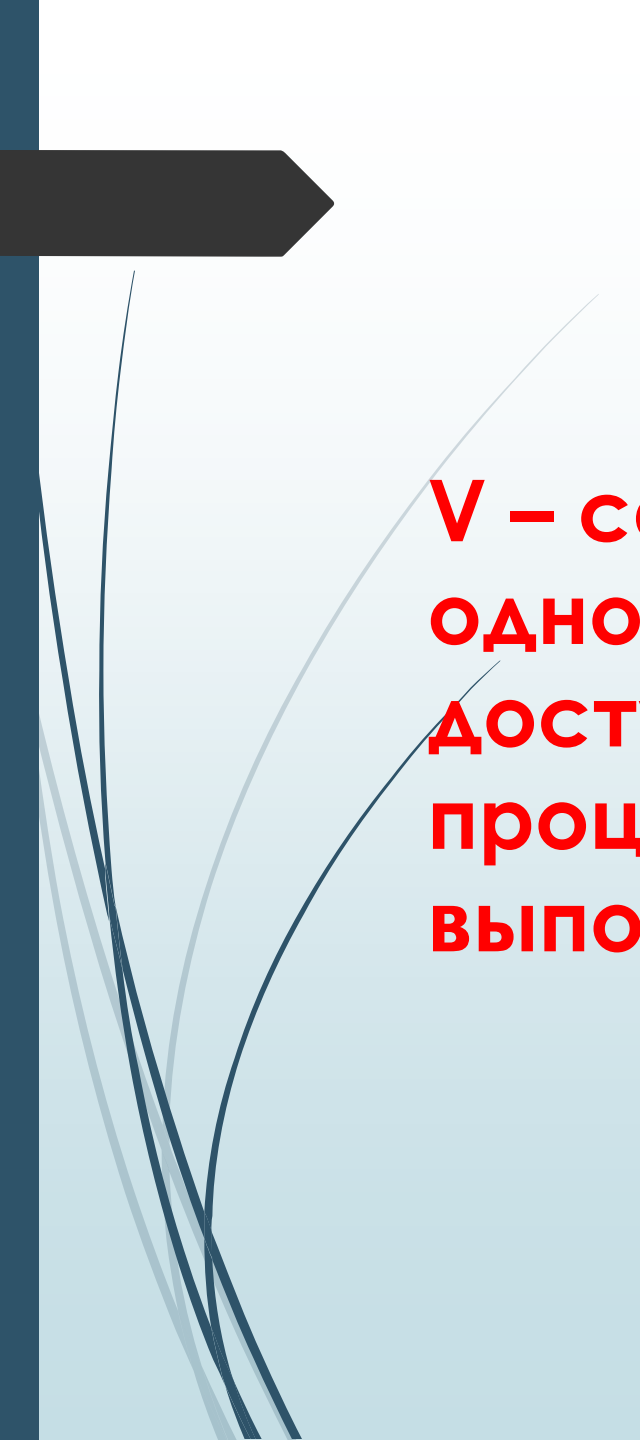


Дейкстра описывает семафоры как неотрицательные целые переменные.

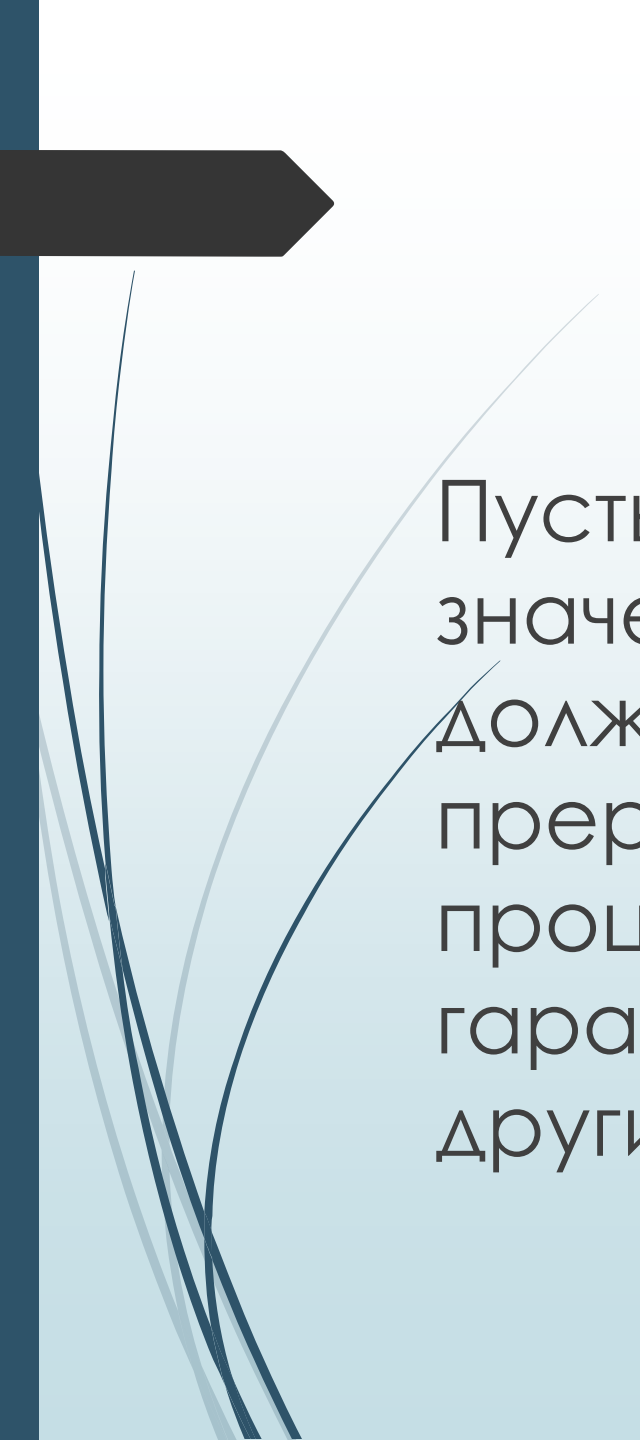
Для семафоров определены 2 операции:



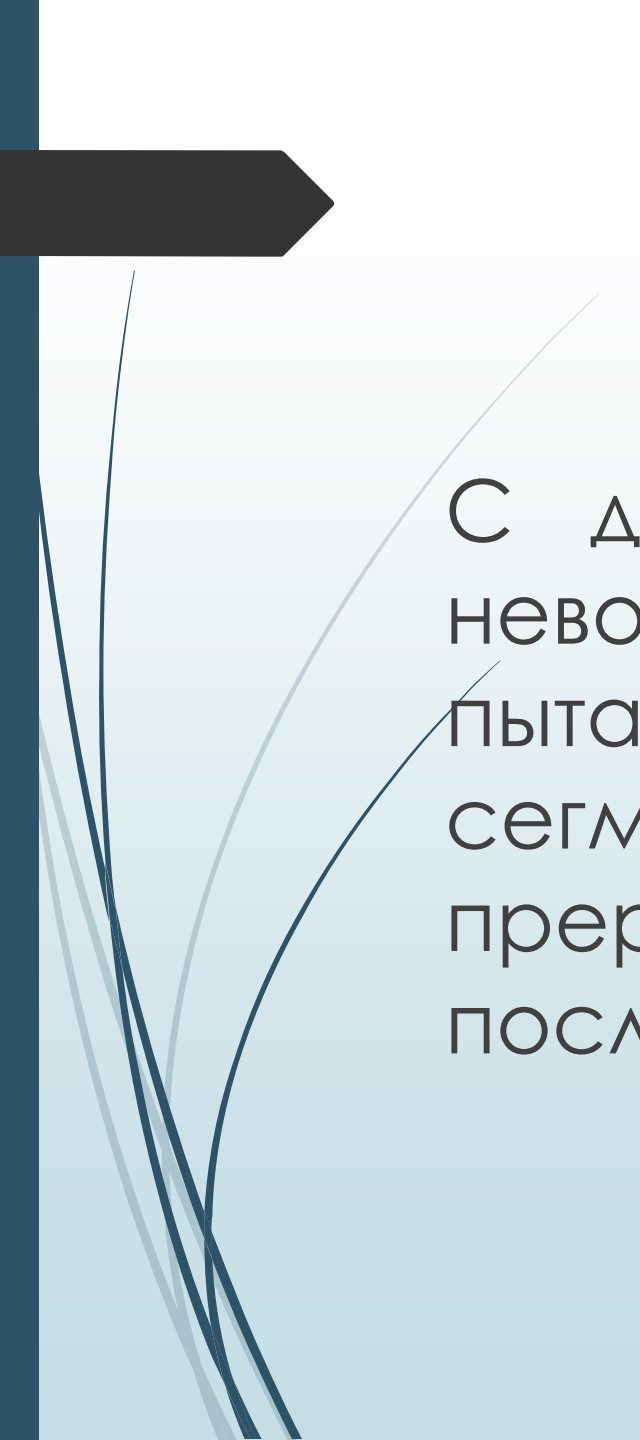
P – уменьшение семафора на единицу, если возможно. Если семафор равен нулю, он не может быть уменьшен, и процесс вызвавший операцию, ждет пока уменьшение станет возможным.




V – семафор увеличивается на единицу одной инструкцией, что предотвращает доступ к семафору любых других процессов, за исключением процесса выполняющего операцию.




Пусть S – двоичный семафор, имеющий значения 0 и 1. Когда $S=0$, какой-то процесс должен выполняться и не может быть прерванным. Поскольку только один процесс может уменьшить S до нуля, гарантируется взаимное исключение всех других процессов.



С другой стороны взаимная блокировка невозможна, потому что процессы, пытающиеся исполнить критический сегмент (не могут быть в это время прерваны), будут выполняться последовательно, когда $S=1$.



Пример решения показан ниже: Процесс должен исполнить V-операцию после выхода из критического сегмента.



```
*  
семафор S  
*  
установить S в 1  
/  
/ Процесс P1:  
/  
P1:P(S)  
    Выполнить критический сегмент  
    V(S)  
    .  
    GOTO P1  
    ...  
/  
/ Процесс Pn:  
/  
Pn:P(S)  
    Выполнить критический сегмент  
    V(S)  
    .  
    GOTO Pn  
    ...
```

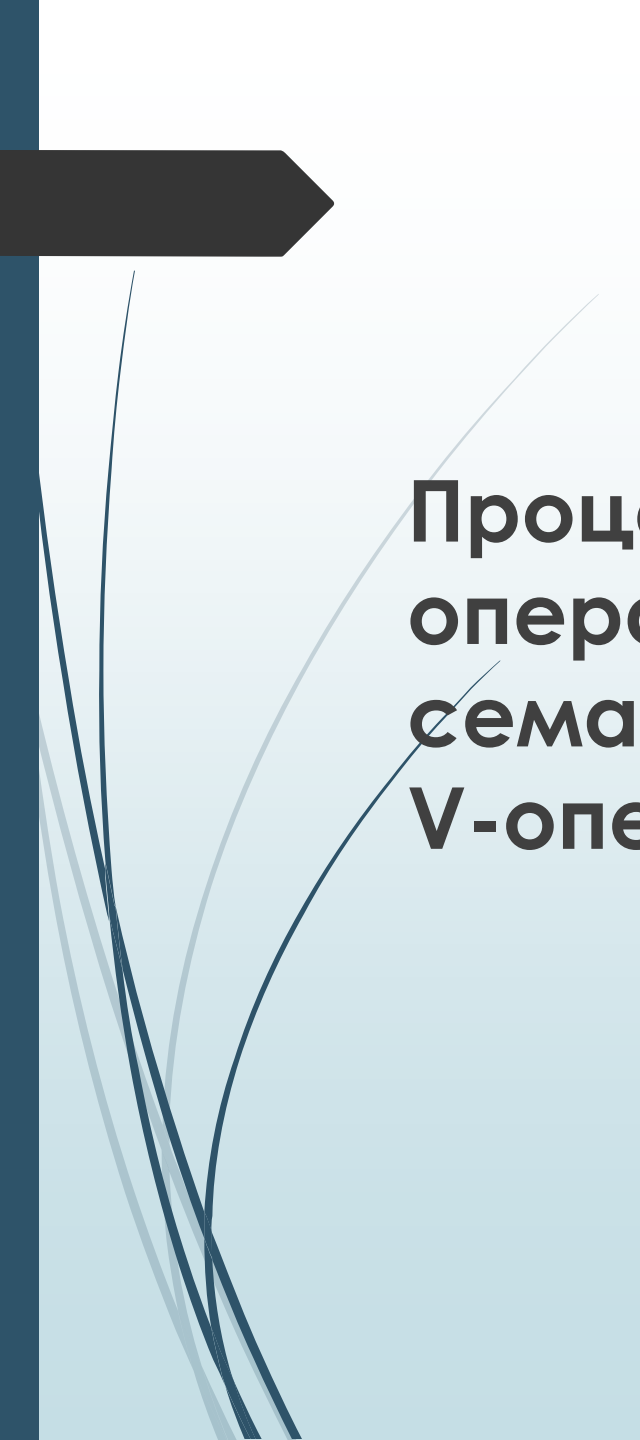


Взаимное исключение через семафоры


Семафоры как счетчики ресурсов и синхронизаторы.

Семафор может использоваться как счетчик ресурса, следящим за потреблением и производством ресурса.

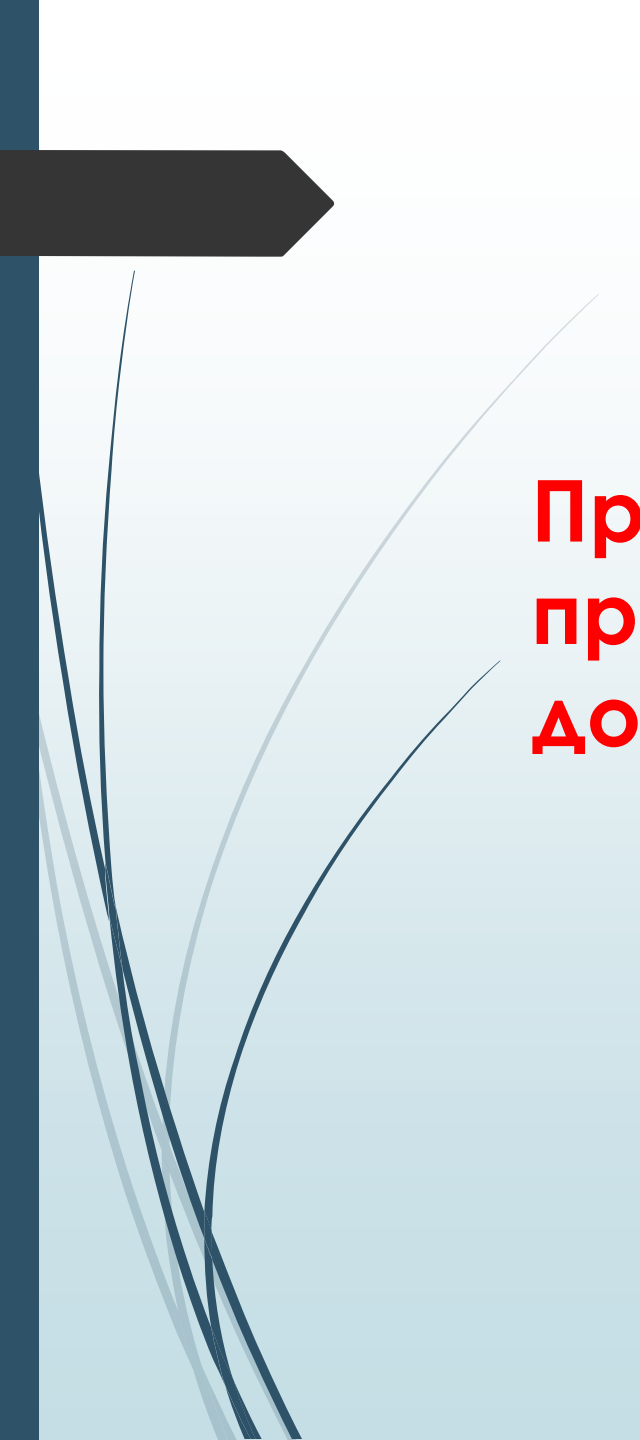
Семафор может также использоваться как синхронизатор, координирующим производство и потребление ресурсов.



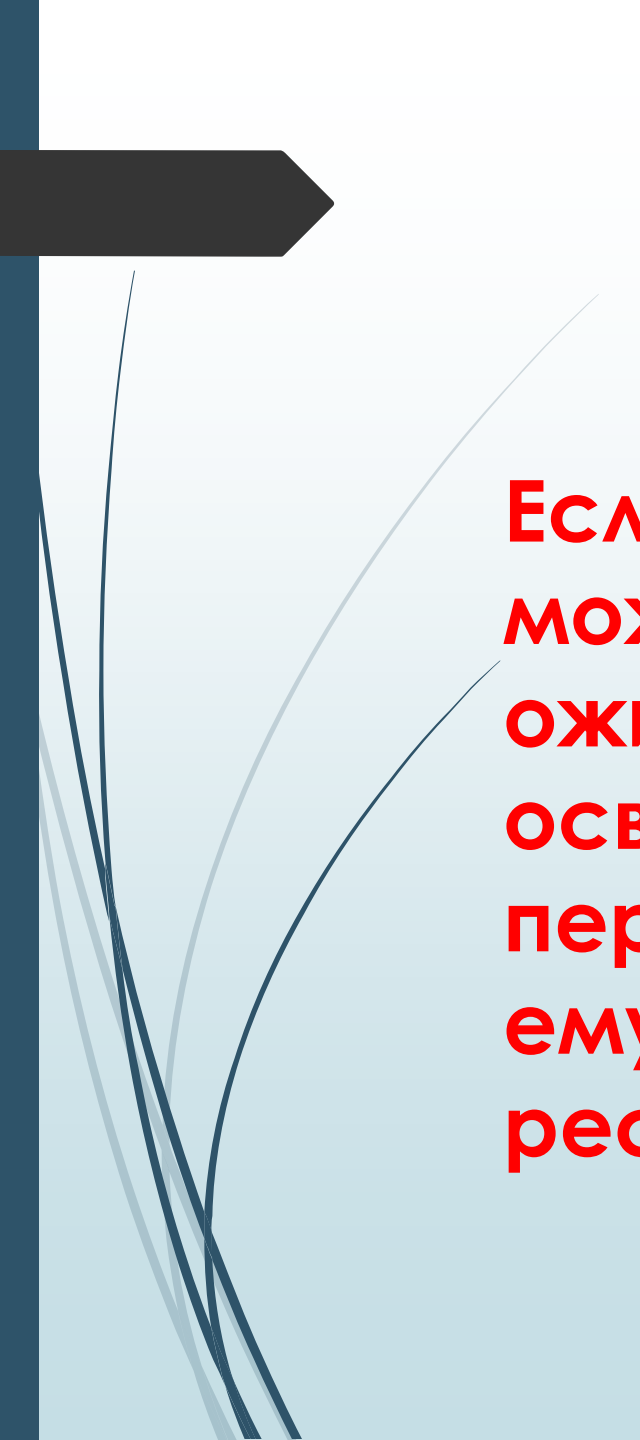
Процесс потребляет ресурс, исполняя Р-операцию над связанным с ресурсом семафором, и производит ресурс, выполняя V-операцию над тем же семафором.




В реализации семафора счетчик уменьшается на единицу при каждой Р-операции и увеличивается на единицу при каждой V-операции.




При инициализации счетчика ему присваивается значение, равное числу доступных ресурсов.



Если ресурс недоступен, то процесс может быть помещен в очередь ожидающих ресурса процессов. При освобождении ресурса активизируется первый процесс в очереди ожидающих и ему предоставляется управление ресурсом.



Пример : Использование двумя процессами одного ресурса (буферная память). Один процесс помещает в память информацию, другой выбирает ее оттуда и что-то с ней делает (например, печатает).



Буферная память состоит из Nбуферов
одинакового размера, каждый из
которых может содержать одну запись.

Эти 2 процесса можно легко
синхронизировать, используя три
семафора:



1) Пуст-буфер – число пустых буферов;

2) Полн-буфер – число полных буферов;

3) Семафор –буфер – семафор взаимного исключения для буферных операций.

Алгоритм синхронизации двух процессов на примере обращения к буферной памяти:

Семафоры: Пуст-буфер, Полн-буфер, Семафор-буфера

Инициализация Пуст-буфер = N (число буферов одинакового размера)

Инициализация Полн-буфер = 0

Инициализация Семафор-буфер = 1

Procedure P(S)

EnDProc

Procedure V(S)

EnDProc

Procedure главный-процесс

Создать выводную запись

P(пуст-буфер)

P(семафор-буфер)

Добавить-в-буфер ()

V(семафор-буфер)

V(полн-буфер)

EnDProc

Procedure процесс-ввода-вывода

P(полн-буфер)

P(семафор-буфер)


Взять из буфера ()

V(семафор-буфер)

V(пуст-буфер)

Печатать запись

EnDProc




Каждый семафор ресурса состоит из блока управления ресурсом и двух очередей: очереди ожидания и очереди свободных элементов.



Блок управления ресурсом имеет следующую структуру:

- адрес блока управления очередью ожидания,
- адрес блока управления очередью свободных элементов,
- адрес распределителя.



Адреса БУО ожидания и БУО свободных элементов указывают на блоки управления очередью.

Очереди содержат соответственно список процессов, ожидающих ресурсы и список свободных ресурсов.

Распределитель – это программа, которая назначает ресурсы ожидающим процессам в некотором предписанном формате.



Решение проблемы производителя и потребителя с помощью семафоров

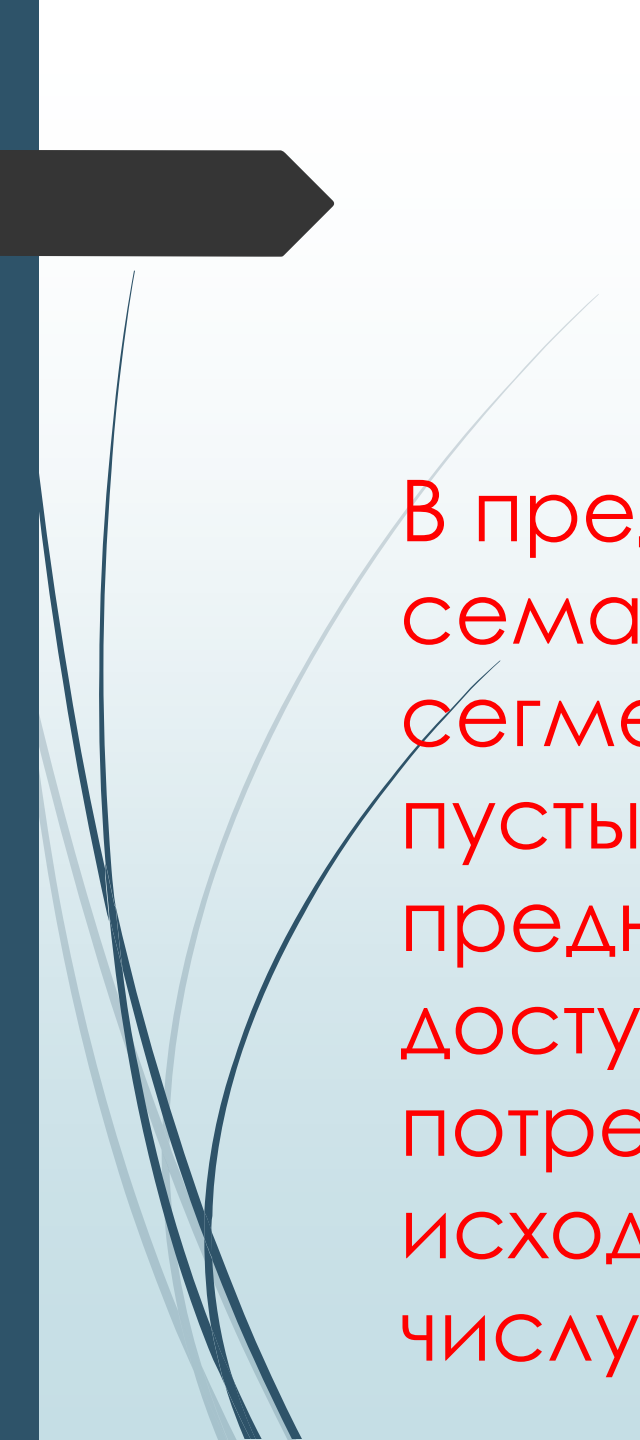
```

#define N 100          /* количество сегментов в буфере */
typedef int semaphore; /* семафоры - особый вид целочисленных переменных */
semaphore mutex = 1;   /* контроль доступа в критическую область */
semaphore empty = N;   /* число пустых сегментов буфера */
semaphore full = 0;     /* число полных сегментов буфера */

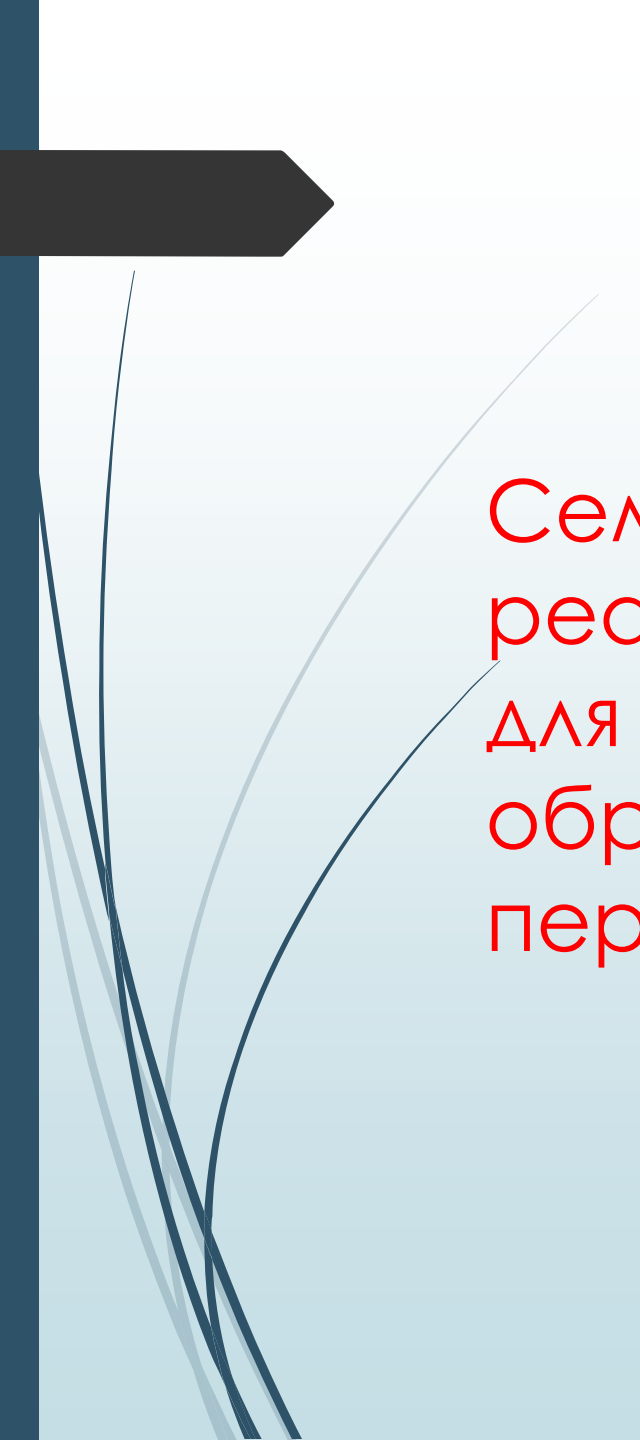
void producer(void)
{
    int item;
    while (TRUE)
    {
        /* TRUE - константа, равная 1 */
        item = produce_item(); /* создать данные, помещаемые в буфер */
        down(& empty); /* уменьшить счетчик пустых сегментов буфера */
        down(& mutex); /* вход в критическую область */
        insert_item(item); /* поместить в буфер новый элемент */
        up(&mutex); /* выход из критической области */
        up(&full); /* увеличить счетчик полных сегментов буфера */
    }
}

void consumer(void)
{
    int item;
    while (TRUE)
    {
        /* бесконечный цикл */
        down(& full); /* уменьшить числа полных сегментов буфера */
        down(& mutex); /* вход в критическую область */
        item = remove_item(); /* удалить элемент из буфера */
        up(& mutex); /* выход из критической области */
        up(& empty); /* увеличить счетчик пустых сегментов буфера */
        consume_item(item); /* обработка элемента */
    }
}


```



В представленном решении используются три семафора: один для подсчета заполненных сегментов буфера (*full*), другой для подсчета пустых сегментов (*empty*), а третий предназначен для исключения одновременного доступа к буферу производителя и потребителя (*mutex*). Значение счетчика *full* исходно равно нулю, счетчик *empty* равен числу сегментов в буфере, а *mutex* равен 1.



Семафор *mutex* используется для реализации взаимного исключения, то есть для исключения одновременного обращения к буферу и связанным переменным двух процессов.



Остальные семафоры использовались для синхронизации. Семафоры *full* и *empty* необходимы, чтобы гарантировать, что определенные последовательности событий происходят или не происходят. В нашем случае они гарантируют, что производитель прекращает работу, когда буфер полон, а потребитель прекращает работу, когда буфер пуст.

Мьютексы

Мьютекс — переменная, которая может находиться в одном из двух состояний: блокированном или неблокированном.


Для описания мьютекса требуется всего один бит, хотя чаще используется целая переменная, у которой 0 означает неблокированное состояние, а все остальные значения соответствуют блокированному состоянию.




Значение мьютекса устанавливается двумя процедурами.

Если поток (или процесс) собирается войти в критическую область, он вызывает процедуру *mutex_lock*.

Если мьютекс не заблокирован (то есть вход в критическую область разрешен), запрос выполняется и вызывающий поток может попасть в критическую область.



Напротив, если мьютекс заблокирован, вызывающий поток блокируется до тех пор, пока другой поток, находящийся к критической области, не выйдет из нее, вызвав процедуру *mutex_unlock*. Если мьютекс блокирует несколько потоков, то из них случайным образом выбирается один.



Мьютексы легко реализовать в пользовательском пространстве, если доступна команда TSL. Код программы для процедур *mutex_lock* и *mutex_unlock* в случае потоков на уровне пользователя представлен в листинге ниже.

Листинг 2.5. Реализация mutex_lock и mutex_unlock

mutex_lock:

TSL REGISTER,MUTEX	Старое значение мьютекса копируется в регистр;
	устанавливается новое значение 1
CMP REGISTER,#0	Сравнение старого значения с нулем
JZE ok	Если старое = 0, мьютекс не был блокирован. Возврат
CALL thread_yield	Мьютекс занят, управление передается другому потоку
JMP mutex_lock	Повторить попытку позже

ok: RET	Возврат, вход в критическую область
---------	-------------------------------------

mutex_unlock:

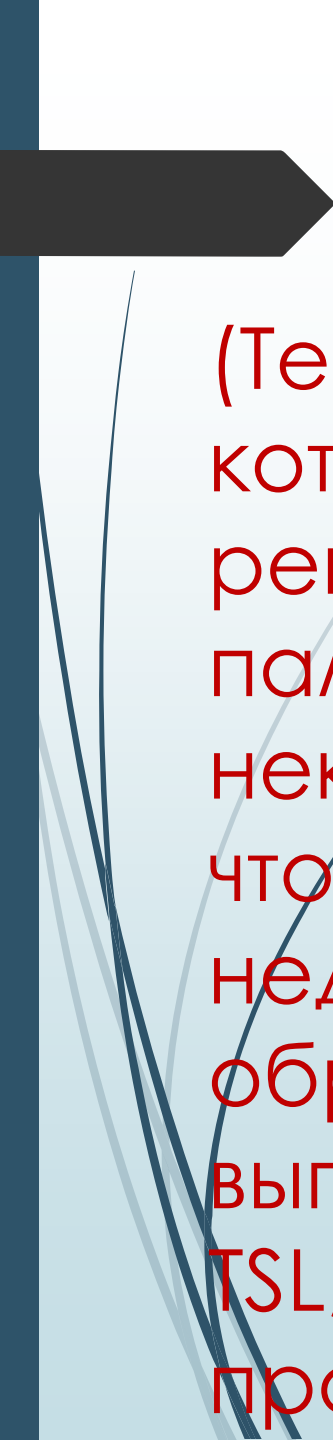
MOVE MUTEX,#0	Устанавливается значение мьютекса 0
RET	Возврат




Команда TSL

Рассмотрим решение, требующее участия аппаратного обеспечения. Многие компьютеры, особенно разработанные с расчетом на несколько процессоров, имеют команду

TSL RX.LOCK



(Test and Set Lock — проверить и заблокировать), которая действует следующим образом. В регистр *RX* считывается содержимое слова памяти *lock*, а в ячейке памяти *lock* сохраняется некоторое ненулевое значение. Гарантируется, что операция считывания слова и сохранения неделима — другой процесс не может обратиться к слову в памяти, пока команда не выполнена. Процессор, выполняющий команду *TSL*, блокирует шину памяти, чтобы остальные процессоры не могли обратиться к памяти.




Воспользуемся командой TSL. Пусть совместно используемая переменная *lock* управляет доступом к разделенной памяти. Если значение переменной *lock* равно 0, любой процесс может изменить его на 1 и обратиться к разделенной памяти, и затем изменить его обратно на 0, пользуясь обычной командой *move*.

Как использовать эту команду для взаимного исключения? Решение приведено в листинге 2.2. Здесь представлена подпрограмма из четырех команд, написанная на фиктивном (но типичном) ассемблере. Первая команда копирует старое значение *lock* в регистр и затем устанавливает значение переменной равное 1. Потом старое значение сравнивается с нулем. Если оно ненулевое, значит, блокировка уже была установлена и проверка начинается сначала. Рано или поздно значение окажется нулевым (это означает, что процесс, находившийся в критической области, вышел из нее), и подпрограмма возвращается, установив блокировку. Программа просто помещает 0 в переменную *lock*. Специальной команды процессора не требуется.

Листинг 2.2. Вход и выход из критической области с помощью команды TSL

```
enter_region:
TSL REGISTER,LOCK | значение lock копируется в регистр, значение переменной
                   | устанавливается равным 1
CMP REGISTER,#0   | Старое значение lock сравнивается с нулем JNE enter_region
                   | Если оно ненулевое, значит, блокировка уже была установлена,
                   | поэтому цикл завершается RET
                   | Возврат к вызывающей программе, процесс в критической области

leave_region:
MOVE LOCK,#0      | Сохранение 0 в переменной lock
RET
```



Прежде чем попасть в критическую область, процесс вызывает процедуру *enter_region*, которая выполняет активное ожидание вплоть до снятия блокировки, затем она устанавливает блокировку и возвращается. По выходе из критической области процесс вызывает процедуру *leave_region*, помещающую 0 в переменную *lock*. Как и во всех остальных решениях проблемы критической области, для корректной работы процесс должен вызывать эти процедуры своевременно, в противном случае взаимное исключение не удастся.