

# Системное программирование

Лекция 7

Многопоточность и синхронизация

# Компиляторы и многопоточность

- Большинство программ являются однопоточными => компиляторы ориентированы на оптимизацию однопоточных программ.
- Многие оптимизации основаны на предположении, что значение переменной не изменяется, если не было записи в нее.
- В многопоточной программе переменная может быть изменена другим потоком => значение переменной изменится, даже если в текущем потоке в нее ничего не записывалось.

Оптимизации, производимые компиляторами, приводят к нарушению работы даже формально корректно составленных многопоточных программ.

Запретить компилятору оптимизации, связанные с доступом к определенной переменной, можно, обозначив переменную как **volatile**.

**Volatile** в общем случае недостаточен [и потому не должен использоваться] для обеспечения корректной работы с общим ресурсом!

# Переупорядочивание операций

Порядок выполнения операций чтения/записи может отличаться от указанного в программе по 2 причинам:

- компилятор может переупорядочить операции во время компиляции в целях оптимизации;
- инструкции могут быть переупорядочены в очереди микроопераций ядра ЦП при использовании спекулятивного выполнения.

Для обеспечения корректного порядка операций используются **барьеры памяти** – точки программы, относительно которых запрещено переупорядочение операций чтения/записи.

Список функций, которые обязаны быть барьерами памяти, приводится в [разделе 4.12 последней версии POSIX](#).

В частности, все функции, работающие с примитивами синхронизации, являются и барьерами времени компиляции, и барьерами времени выполнения.

# Атомарные операции (пример)

Операция называется **атомарной** (греч. atomos, неделимый), если с точки зрения внешнего наблюдателя она либо не выполняется, либо выполняется целиком.

На x86-64 по умолчанию атомарными являются выровненные чтение/запись переменных размером  $\leq 8$  байт. Чтение/запись переменных размером  $\leq 16$  байт в определенных условиях также происходят атомарно.

Все остальные действия без применения дополнительных мер являются неатомарными.

# Неатомарное чтение-изменение-запись

`volatile int x=0;`

Thread1: `++x;`

Thread2: `x += 2;`

Thread1: `int y = x;`

Переменная `y` может принять значения 1, 2 или 3.

Ядро 1		Ядро 2	
<code>mov</code>	<code>eax, [x]</code>		
<code>inc</code>	<code>eax</code>	<code>mov</code>	<code>eax, [x]</code>
<code>mov</code>	<code>[x], eax</code>	<code>add</code>	<code>eax, 2</code>
		<code>mov</code>	<code>[x], eax</code>

Ядро 1		Ядро 2	
<code>mov</code>	<code>eax, [x]</code>	<code>mov</code>	<code>eax, [x]</code>
<code>inc</code>	<code>eax</code>	<code>add</code>	<code>eax, 2</code>
<code>mov</code>	<code>[x], eax</code>	<code>mov</code>	<code>[x], eax</code>

t

# Атомарные типы данных

В C11/C++11 вместе с поддержкой многопоточности были введены атомарные типы данных (заголовочный файл <atomic> в C++, <stdatomic.h> в C ).

Атомарные переменные гарантируют атомарность связанных с ними операций.

**Замечание:** производительность атомарных переменных ниже, чем неатомарных.

Неполный список приведен ниже

C++

```
std::atomic_flag  
std::atomic_int = std::atomic<int>  
std::atomic_long = std::atomic<long>  
std::atomic_llong = std::atomic<long long>
```

C

```
atomic_flag  
atomic_int = _Atomic int  
atomic_long = _Atomic long  
atomic_llong = _Atomic long long
```

# std::atomic

```
std::atomic_int x=0;
```

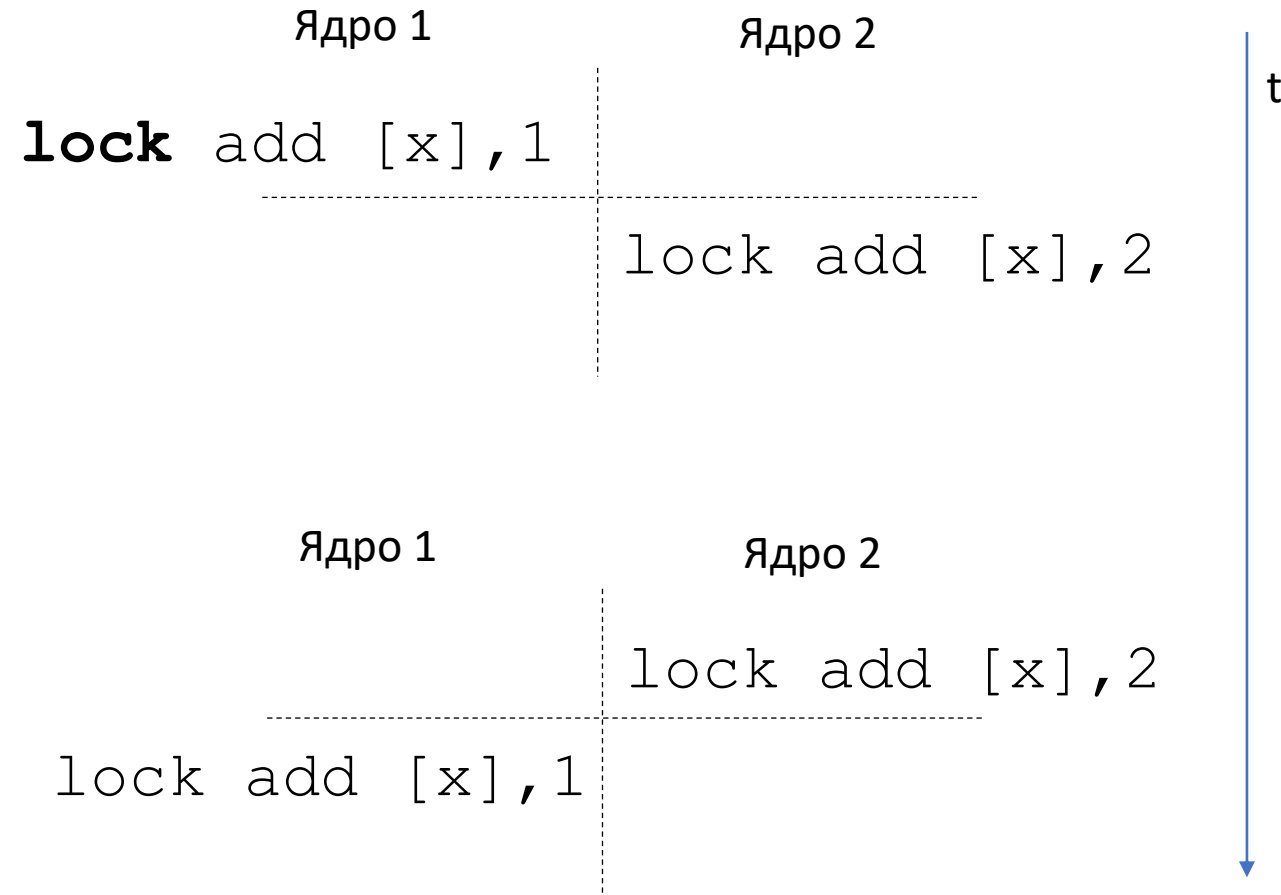
```
Thread1: ++x;
```

```
Thread2: x += 2;
```

```
Thread1: int y = x;
```

Переменная `y` может принять только значения 1 и 3.

Префикс **lock** накладывает блокировку на кэш-линию с переменной до завершения инструкции. Другие ядра получить доступ к кэш-линии не могут.

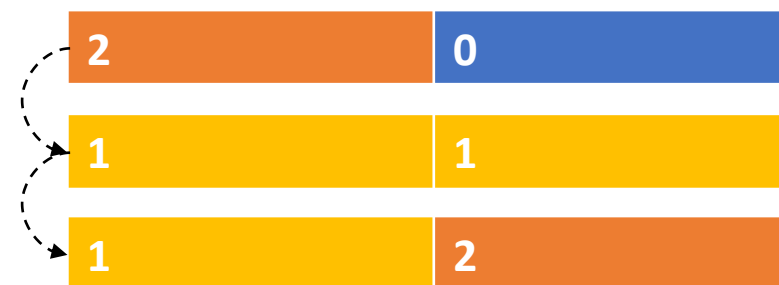
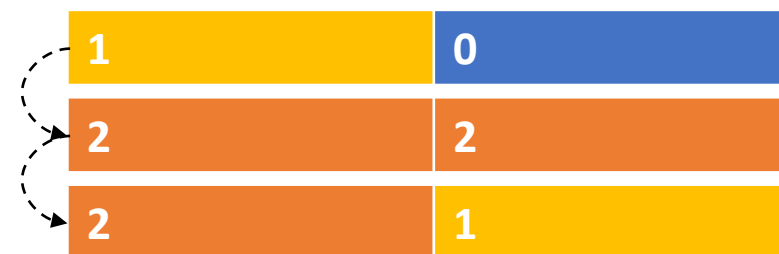


# Неатомарное чтение/запись

```
struct S { size_t x, y; };  
volatile S global_var = {0,0};
```

```
Thread1: global_var = { 1,1 };  
Thread2: global_var = { 2,2 };  
Thread1: S local_var = global_var;
```

Переменная `local_var` с некоторой вероятностью может быть {1,1}, {2,2}, {1,2} и {2,1};



*Проблема неатомарного чтения/записи может быть решена только с помощью примитивов синхронизации!*



# Состояние гонки и синхронизация

Ситуация, при которой корректность работы программы зависит от порядка выполнения потоками определенных действий называется **состоянием гонки** (race condition).

Состояние гонки является источником трудноуловимых ошибок, поскольку проявление таких ошибок случайно (например, программа падает в 1% случаев, при которых сложились специальные условия).

Состояние гонки – всегда ошибка программиста.

В целом, действия по обеспечению корректности результатов работы взаимодействующих потоков называют **синхронизацией**.

Для устранения состояния гонки применяются атомарные переменные и примитивы синхронизации.

# Общий ресурс

Ресурс (переменная, файл и пр.) является **общим** (shared), если:

- Доступ к нему производится из одного или нескольких потоков (возможно разных процессов);
- Доступ к нему производится из обработчика сигнала и из обычного кода.

Общие ресурсы являются источником состояний гонки (точнее их разновидности – **data race**). В частности, любые неатомарные операции над общим ресурсом автоматически порождают состояние гонки.

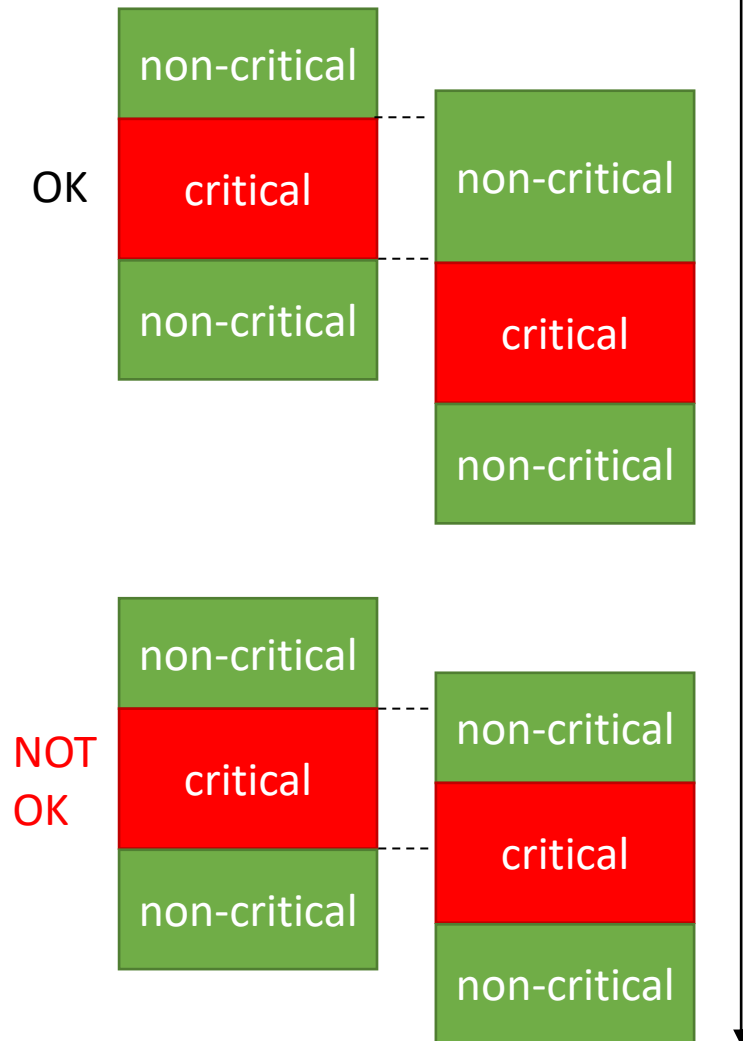
# Критические секции

Участок кода, в котором происходит доступ к общему ресурсу, называется **критической секцией** (critical section).

В пределах критической секции необходимо обеспечить:

- Актуальность видимого состояния общего ресурса в момент входа и выхода из секции;
- Возможность изменения состояния ресурса только из текущей секции;
- Невозможность чтения промежуточного состояния общего ресурса.

Все задачи решаются с использованием примитивов синхронизации.



# Примитивы синхронизации

**Примитивы синхронизации** – специальные объекты, используемые для обеспечения синхронизации.

- Семафоры (`sem_t`);
- Мьютексы (`pthread_mutex_t`);
- Циклические блокировки (`pthread_spinlock_t`);
- Блокировки чтения-записи (`pthread_rwlock_t`);
- Условные переменные (`pthread_conditional_t`);
- Барьеры (`pthread_barrier_t`).

*Примечание: примитивы синхронизации POSIX гарантируют, что поток «видит» актуальное состояние памяти => помечать общие переменные `volatile` необязательно.*

# Примитивы синхронизации в Pthreads

Аналогами конструктора и деструктора для примитивов выступают функции вида\*

```
int pthread_###_init(pthread_###_t* p,  
                     const pthread_###attr_t* attr);  
int pthread_###_destroy(pthread_###_t* p);
```

Обе функции возвращают 0 в случае успеха или код ошибки .

\*состав аргументом может отличаться

# Блокировки

**Блокировка (lock)** – примитив синхронизации используемый для ограничения доступа в общему ресурсу.

- Мьютексы [эксклюзивная блокировка];
- Циклические блокировки [эксклюзивная блокировка];
- Блокировки чтения-записи;

Перед входом в критическую секцию блокировка «захватывается», после выхода из критической секции – «освобождается» или «снимается».

Эксклюзивные блокировки могут быть захвачены только одним потоком, неэксклюзивные - несколькими потоками одновременно.

# Взаимоблокировки

**Взаимоблокировка** (deadlock)— ситуация, при которой потоки захватывают блокировки таким образом, что никто не может ни разблокировать свою блокировку, ни получить чужую.

```
lock(a);  
lock(b);  
/*USE RESOURCE*/  
unlock(b);  
unlock(a);
```

```
lock(b);  
lock(a);  
/*USE RESOURCE*/  
unlock(a);  
unlock(b);
```

t

# Взаимоблокировки

Для избегания взаимоблокировок, делайте критические секции как можно короче и проще и не вызывайте в них функции, которые также могут блокироваться. Если нужно захватить сразу несколько блокировок – используйте `try_lock`.

```
if (try_lock(a)){  
    if(try_lock(b)){  
        /*USE RESOURCE*/  
        unlock(b);  
    }  
    unlock(a);  
}  
  
if(try_lock(b)){  
    if (try_lock(a)){  
        /*USE RESOURCE*/  
        unlock(a);  
    }  
    unlock(b);  
}
```



# Самоблокировки

**Самоблокировка** - ситуация, при которой поток из-за ошибки блокирует сам себя. Для избегания самоблокировок всегда убеждайтесь, что каждому `lock()` всегда соответствует `unlock()`.

```
lock_t m;  
int y;  
  
void foo(int x) {  
    lock(m);  
    if (!x)  
        return; //missing unlock  
    y = x;  
    unlock(m);  
}
```

```
lock_t m;  
int y;  
  
void foo(int x) {  
    lock(m);  
    if (x)  
        y = x;  
    unlock(m);  
}
```

# Блокировки в Pthreads

Для мьютексов, циклических блокировок и блокировок чтения записи определены операции блокировки и разблокировки

//Заблокировать примитив. Если уже заблокирован – ждать

```
int pthread_###_lock(pthread_###_t* p);
```

//Заблокировать примитив. Если уже заблокирован, результат==EBUSY

```
int pthread_###_trylock(pthread_###_t* p);
```

//Разблокировать примитив

```
int pthread_###_unlock(pthread_###_t* p);
```

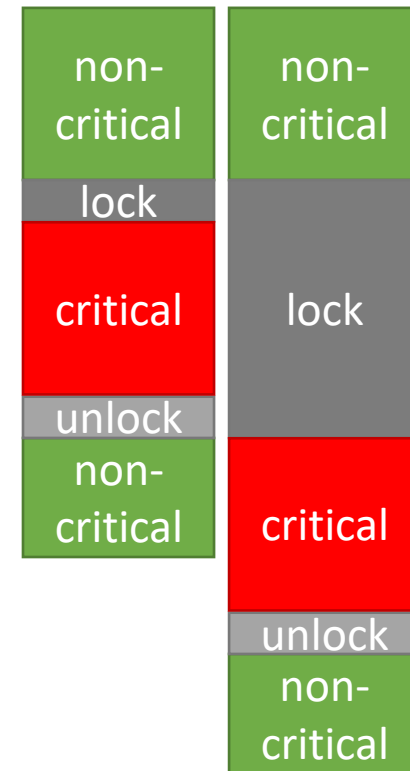
**Замечание:** блокировки не отслеживают, кто их захватил! Блокировка может быть случайно снята не тем потоком, который ее захватил, в случае ошибки.

# Мьютексы и циклические блокировки (пример)

**Мьютекс** (mutex, от **mutual exclusion**) – примитив синхронизации, реализующий исключительную блокировку ресурса.

**Циклическая блокировка** – аналог мьютекса, реализуемый через `while(f){};`

- Попытка захватить уже захваченный мьютекс приведет к тому, что поток приостановится.
- Попытка захватить уже захваченную циклическую блокировку приведет к попаданию потока в цикл `while(f){}`, т.е. поток будет активно проверять состояние блокировки.
- Циклическую блокировку следует выбирать, если критическая секция является небольшой – т.е. время ожидания разблокировки мало.



# Задача читателя-писателя

Задача читателя писателя является типовой задачей синхронизации.

Дано:

- общий ресурс;
- потоки, которые производят обновление состояния ресурса (писатели);
- потоки, которые только читают состояние ресурса (читатели).

Задача: обеспечить корректный доступ к ресурсу, т.е.

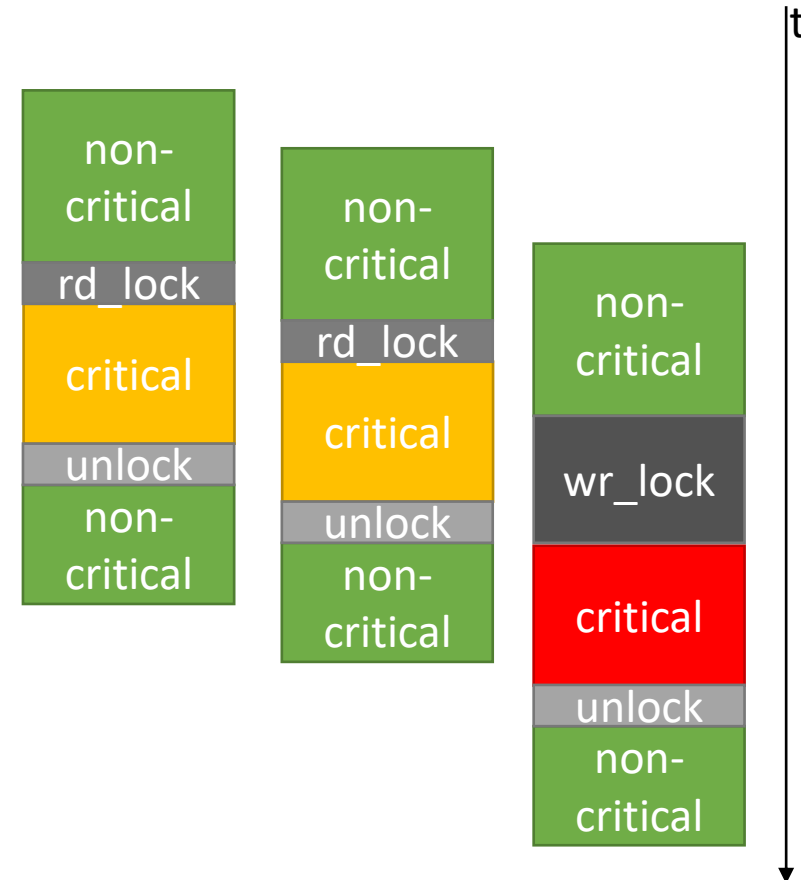
- доступ на чтение могут получить одновременно несколько читателей;
- доступ на запись может иметь только один из писателей.

# Блокировки чтения-записи

**Блокировка чтения-записи** (read-write lock) – примитив синхронизации, предоставляющий 2 типа блокировок:

- Блокировки чтения (неисключительные, несколько потоков могут получить доступ на чтение);
- Блокировки записи (исключительные, только один поток может иметь доступ на запись, остальные потоки не могут получить никакой доступ).

Блокировки чтения-записи следует использовать, если для ресурса отдельно определены операции чтения и операции изменения/записи, причем операции чтения не изменяют состояние ресурса.



# Условные переменные (пример)

**Условные переменные** (переменные состояния, conditional variables)- примитив синхронизации, позволяющий уведомить ожидающие потоки о наступлении определенного события.

//уведомить о событии и разблокировать 1 поток

```
int pthread_cond_signal(pthread_cond_t* cond);
```

//уведомить о событии и разблокировать все потоки

```
int pthread_cond_broadcast(pthread_cond_t* cond);
```

//подождать уведомления. mutex должен быть заблокирован ДО  
ВЫЗОВА

```
int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t*  
mutex);
```

# Условные переменные

Условные переменные работают в паре с мьютексами.

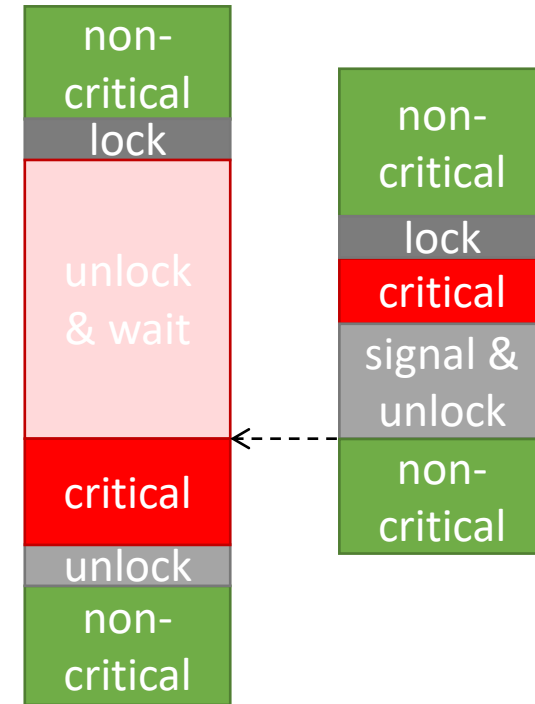
- Ожидающий поток захватывает мьютекс, проверяет условие, если условие не выполнено - вызывает `pthread_cond_wait()`. Поток засыпает, мьютекс разблокируется.
- Уведомляющий поток захватывает мьютекс, меняет состояние, вызывает `pthread_cond_signal()`.
- Один из ожидающих потоков просыпается и захватывает мьютекс (поток дождется разблокировки мьютекса, если он захвачен).
- Если уведомляющий поток вызывает `pthread_cond_broadcast()`, то просыпаются все потоки, по очереди захватывают мьютекс и выполняют действия.

Если на одной и той же условной переменной ожидают несколько потоков, то неизвестно, какой именно поток будет разбужен `pthread_cond_signal()`.

# Условные переменные

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
int data;  
bool ready = false;
```

```
void reader() {  
    pthread_mutex_lock(&mut);  
    while (!ready) //ready == false, если мы пришли слишком рано  
        pthread_cond_wait(&cond, &mut); //mut разблокируется здесь  
    std::cout << data << std::endl;  
    ready = false;  
    pthread_mutex_unlock(&mut);  
}
```

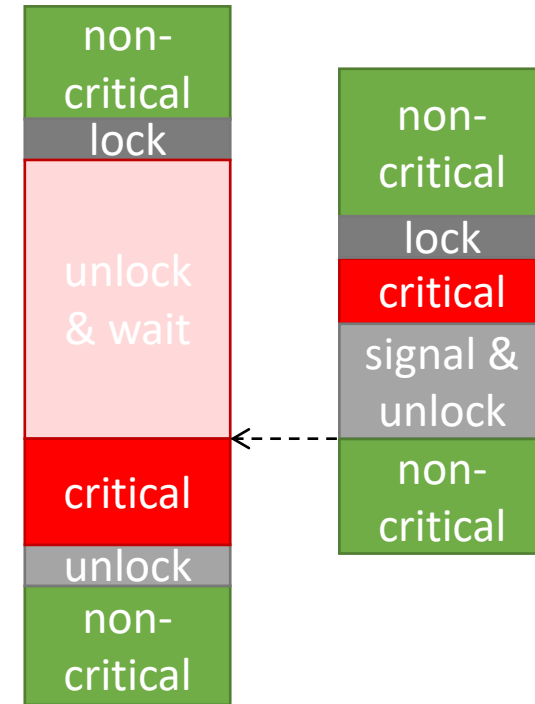




# Условные переменные

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;  
int data;  
bool ready = false;
```

```
void writer() {  
    pthread_mutex_lock(&mut);  
    data = rand();  
    ready = true;  
    pthread_cond_signal(&cond); //сигнал другим потокам  
    pthread_mutex_unlock(&mut);  
}
```



# Барьеры

**Барьер** – примитив синхронизации, позволяющий подождать прибытия N других потоков к заданной точке.

```
int pthread_barrier_init(pthread_barrier_t* barrier,  
                        const pthread_barrierattr_t* attr,  
                        unsigned int count);  
  
int pthread_barrier_destroy(pthread_barrier_t* barrier);
```

Число потоков указывается в параметре `count`.

# Барьеры

По прибытии к заданной точке отмечает прибытие к барьеру вызовом функции

```
int pthread_barrier_wait(pthread_barrier_t* barrier);
```

Потоки будут ждать до тех пор, пока все `count` потоков не заблокируются на том же барьере.

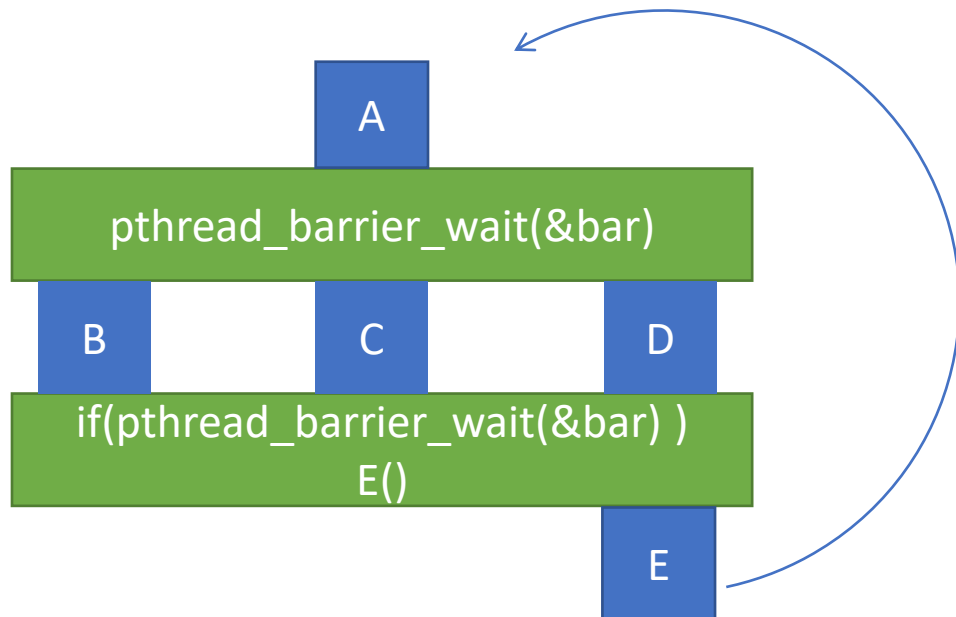
В `count - 1` потоках результатом функции будет 0.

В одном (неизвестно, каком конкретно) потоке результатом будет константа `PTHREAD_BARRIER_SERIAL_THREAD`.

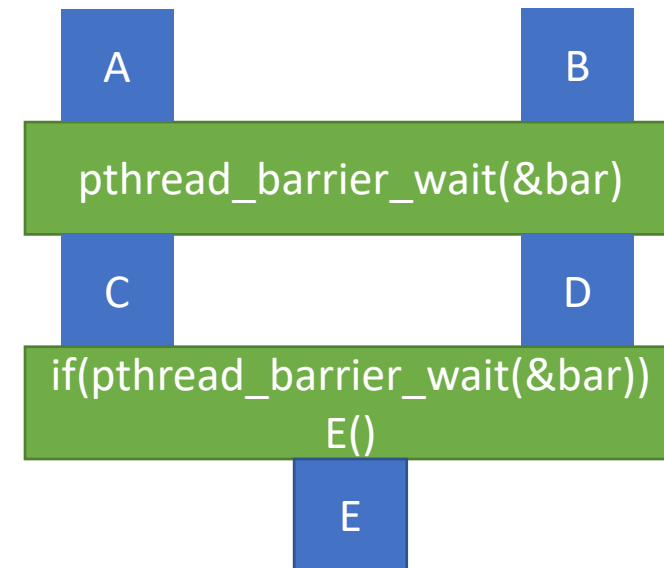
После «срабатывания» барьер может быть использован вновь.

# Барьеры (пример)

Барьеры используются для циклического выполнения параллельных операций либо для выполнения последовательных блоков параллельных операций.



Операции A-E выполняются циклично, при этом B-C-D могут выполняться параллельно. Выгодно создать для этого 3 потока и синхронизировать их барьером.



Операциям C и D необходим результат обеих операций A и B, но A-B и C-D между собой не зависят. Выгодно создать 2 потока и синхронизировать их барьером.

# Использование примитивов pthreads для межпроцессной синхронизации

Если примитив синхронизации поместить в разделяемую память, то становится возможной синхронизация потоков в разных процессах.

Для некоторых примитивов при создании в атрибутах следует явно указать, что он будет использоваться несколькими процессами.

```
pthread_mutexattr_t attr;  
pthread_mutexattr_init(&attr);  
pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);  
pthread_mutex_init(..., &attr);
```