

Введение в многопоточное программирование

АКОС, МФТИ

31 октября, 2024



Напомним:

Что такое поток?

Поток - это единица планирования процессорного времени.

Поток является **частью процесса**.

Потоки могут **выполняться независимо** друг от друга.

Что у всех потоков общее?

Почти всё: адресное пространство, набор дескрипторов, и прочее.

Что у потока своё?

Контекст выполнения, стек, и всякие настройки...

Многопоточно стреляем себе в ногу

Поток 1

```
1 while(true) {  
2     strcpy(MSG, "...");  
3 }
```

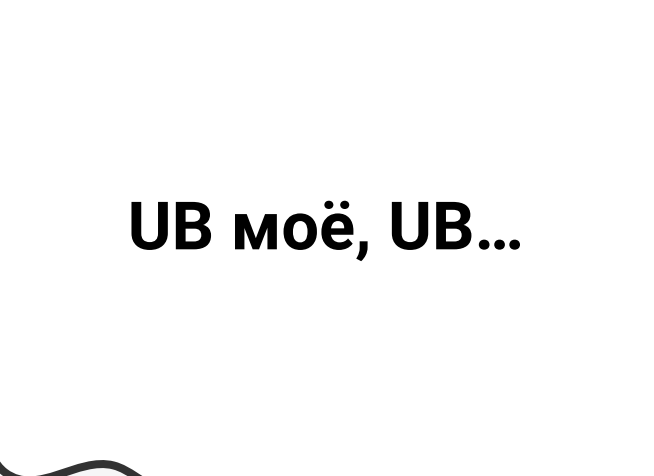
Поток 2

```
1 while(true) {  
2     strcpy(MSG, "@@@@@@");  
3 }
```

```
1 while(i < 100000) puts(MSG);
```

Главный поток

Что выведется?

[illegible]

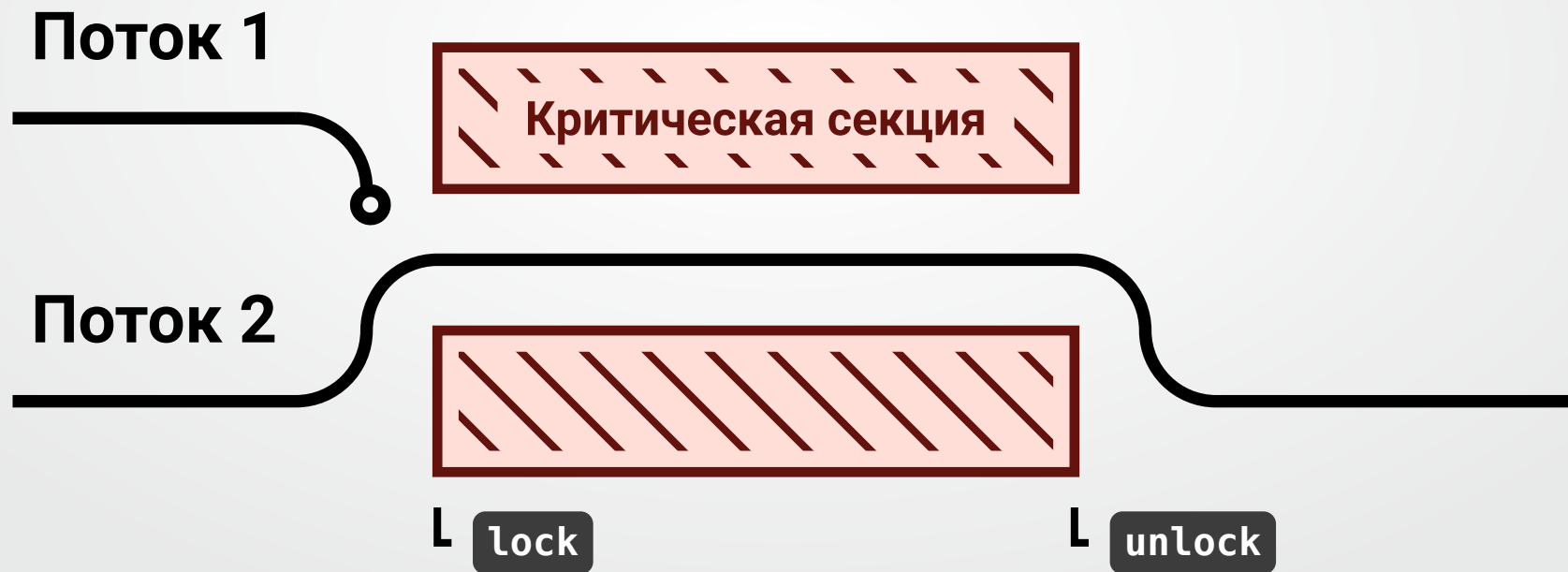
UB moë, UB...

Как это исправить?

mutex

Примитив синхронизации, позволяющий гарантировать выполнение участка кода только **одним потоком** одновременно. (дословно - **mutually exclusive**)

Имеет две операции: `lock` и `unlock`. Иногда их называют `acquire` и `release`



pthread_mutex_t

Реализация мьютекса в библиотеке pthread

pthread_mutex_init(...)

Конструктор;

pthread_mutex_destroy(...)

Деструктор;

pthread_mutex_lock(...)

Захватить с ожиданием;

pthread_mutex_trylock(...)

Захватить немедленно или вернуть EBUSY ;

pthread_mutex_unlock(...)

Отпустить мьютекс.

```
1 pthread_mutex_lock(&lock);
2
3 // Эта строка исполнится не более чем одним потоком одновременно
4
5 pthread_mutex_unlock(&lock);
```

Задача об обедающих философах



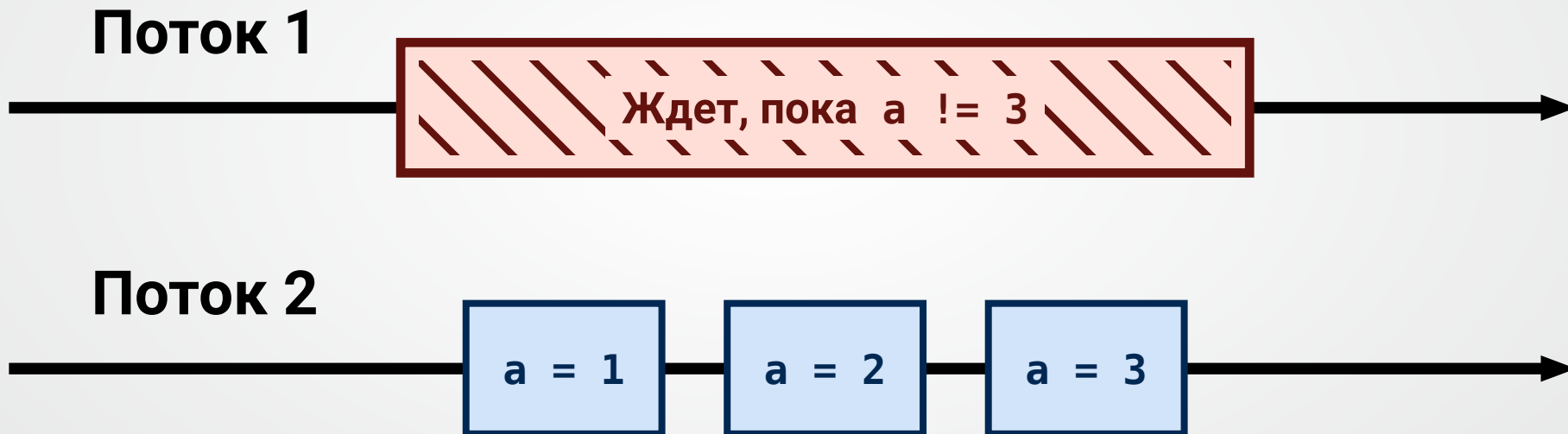
- **Философы сидят за круглым столом и едят бесконечную лапшу.**
- Каждый философ может либо есть, либо размышлять произвольное время.
- Чтобы начать есть, философу нужно взять **две вилки.**
- Число вилок равно числу философов.
- **Как нужно брать вилки, чтобы не возникло взаимоблокировки?**

Автор иллюстрации: Benjamin D. Esham

condvar

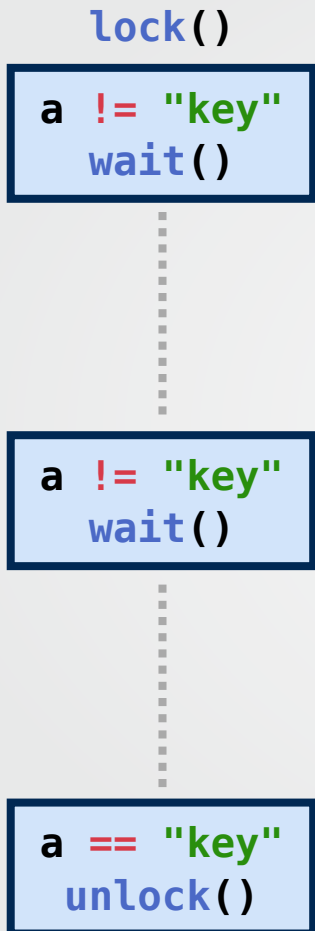
Примитив синхронизации для ожидания условия. Работает в паре с мьютексом.

- Дословно - **conditional variable**

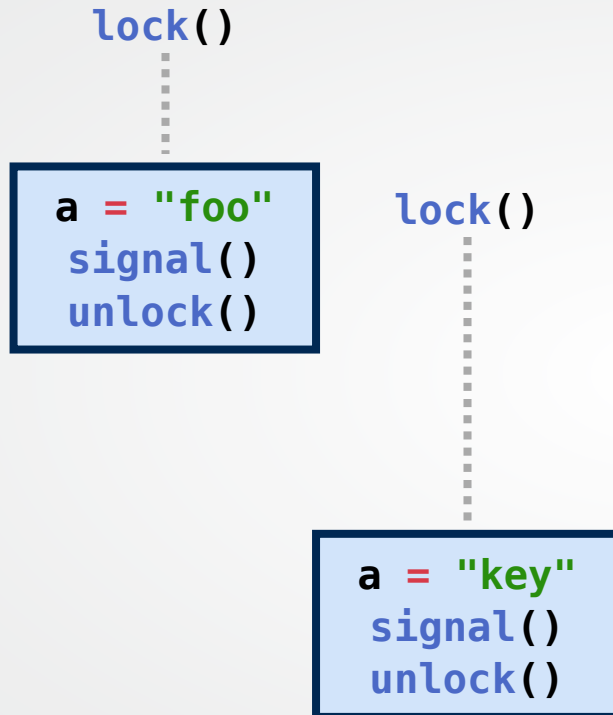


- Саму переменную и условие ожидания **можно сделать любыми**.

Ожид.поток:



Обновляющие потоки:



Интерфейс condvar

wait() :

- **Заснуть**, освободив мьютекс. При пробуждении захватить мьютекс.
- После пробуждения нужно перепроверить условие.

signal() :

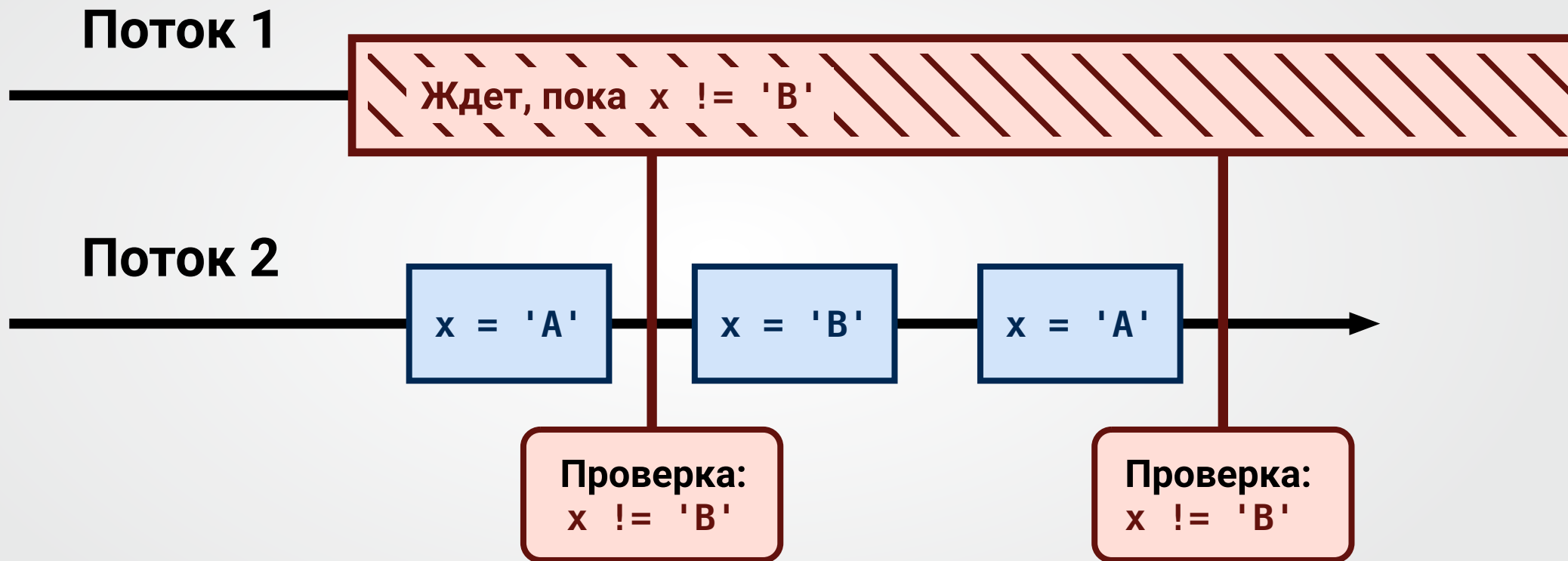
- **Разбудить один** из ожидающих потоков.

broadcast() :

- **Разбудить все** ожидающие потоки.

Синим выделены критические секции.

В чем проблема `condvar`?



Проверки могут происходить не сразу. Это может привести к [проблеме ABA](#).

Корректное использование `condvar`

```
1 lock(&mutex);
2 while(!condition) {
3     wait(&condvar, &mutex);
4 }
5
6 /* Условие выполнено */
7
8 unlock(&mutex);
```

Ожидание условия

```
1 lock(&mutex);
2 strcpy(a, "foo"); // Или key
3 signal(&condvar);
4 unlock(&mutex);
```

Изменение переменной



Осторожно, Spurious wakeups!

Иногда поток может проснуться сам, без вызова `signal()` или `broadcast()`.

pthread_cond_t

Реализация `condvar` в библиотеке `pthread`

`pthread_cond_init(...)`

Конструктор;

`pthread_cond_destroy(...)`

Деструктор;

`pthread_cond_wait(...)`

Освободить мьютекс и **ожидать сигнала**;

`pthread_cond_timedwait(...)`

Аналогично, но с таймаутом;

`pthread_cond_signal(...)`

Разбудить один ожидающий процесс.

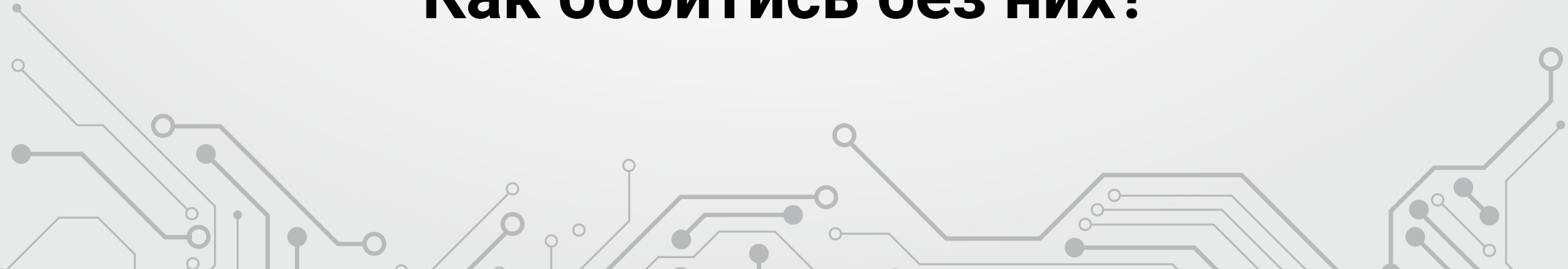
`pthread_cond_broadcast(...)`

Разбудить все ожидающие процессы.

Чем плохи блокировки?

- Накладные расходы на системные вызовы;
- Можно долго ждать планировщик;
- Можно поймать deadlock.

Как обойтись без них?



Атомарные операции

и Lock-Free

Атомарность

Это гарантия того, что операция будет выполнена целиком и неделимо.

- Процессоры умеют атомарно работать с простыми типами. В x86 за это отвечает префикс **lock**. Например, так:

lock add [ebx], ecx		Атомарно $*(\text{ebx}) += \text{ecx};$
lock xchg [ebx], ecx		Атомарно $*(\text{ebx}) \rightleftharpoons \text{ecx};$
lock cmpxchg [ebx], ecx		Атомарно if (eax == *(ebx)) $*(\text{ebx}) \rightleftharpoons \text{ecx};$

- У некоторых инструкций префикс **lock** есть неявно. Например, у **xchg**.
- Некоторые простые структуры данных можно сделать потокобезопасными с помощью лишь атомарных операций, без мьютексов и блокировок.

Атомарность в Си

Начиная с C11, переменные можно аннотировать как атомарные через `_Atomic`

```
void atomic_init(A* obj, C desired)
```

Конструктор;

```
C atomic_load(A* obj)
```

Атомарное чтение;

```
void atomic_store(A* obj, C desired)
```

Атомарная запись;

```
C atomic_exchange(A* obj, C desired)
```

Атомарно обменивать значения;

```
C atomic_add(A* obj, M desired)
```

Атомарное сложение;

И так далее...

Интерфейс к `cmpxchg` на x86, или аналогичным инструкциям на других платформах:

- `bool atomic_compare_exchange_weak(A* obj, C* expected, C desired)`
- `bool atomic_compare_exchange_strong(A* obj, C* expected, C desired)`

`weak` отличается от `strong` тем, что может дать ложный сбой. [See docs](#).

Неблокирующий стек

Основан на связном списке. За кадром – определения структур `node` и `stack`.

```
1 void push(_Atomic stack_t *s, node_t  
  *node)  
2 {  
3     stack_t next = {};  
4     stack_t orig = atomic_load(s);  
5     do {  
6         node->next = orig.node;  
7         next.tag = orig.tag + 1;  
8         next.node = node;  
9     } while(  
10    !atomic_compare_exchange_weak(  
11    s, &orig, next))  
12 }
```

```
1 node *pop(_Atomic stack_t *s)  
2 {  
3     stack_t next = {};  
4     stack_t orig = atomic_load(s);  
5     do {  
6         if (orig.node == NULL)  
7             return NULL;  
8         next.tag = orig.tag + 1;  
9         next.node = orig.node->next;  
10    } while(  
11    !atomic_compare_exchange_weak(  
12    s, &orig, next))  
13    return orig.node;  
14 }
```

Lock-Free структуры писать сложно. Даже стек.
А Lock-Free очереди вообще посвящена [научная работа](#).

Спасибо за внимание!



 github.com/JakMobius/courses/tree/main/mipt-os-basic-2024