

# АКОС 8

**Многопоточная синхронизация  
(для самых маленьких)**

Inspired by [R.Lipovsky](#)

# pthread

```
#include <pthread.h>
```

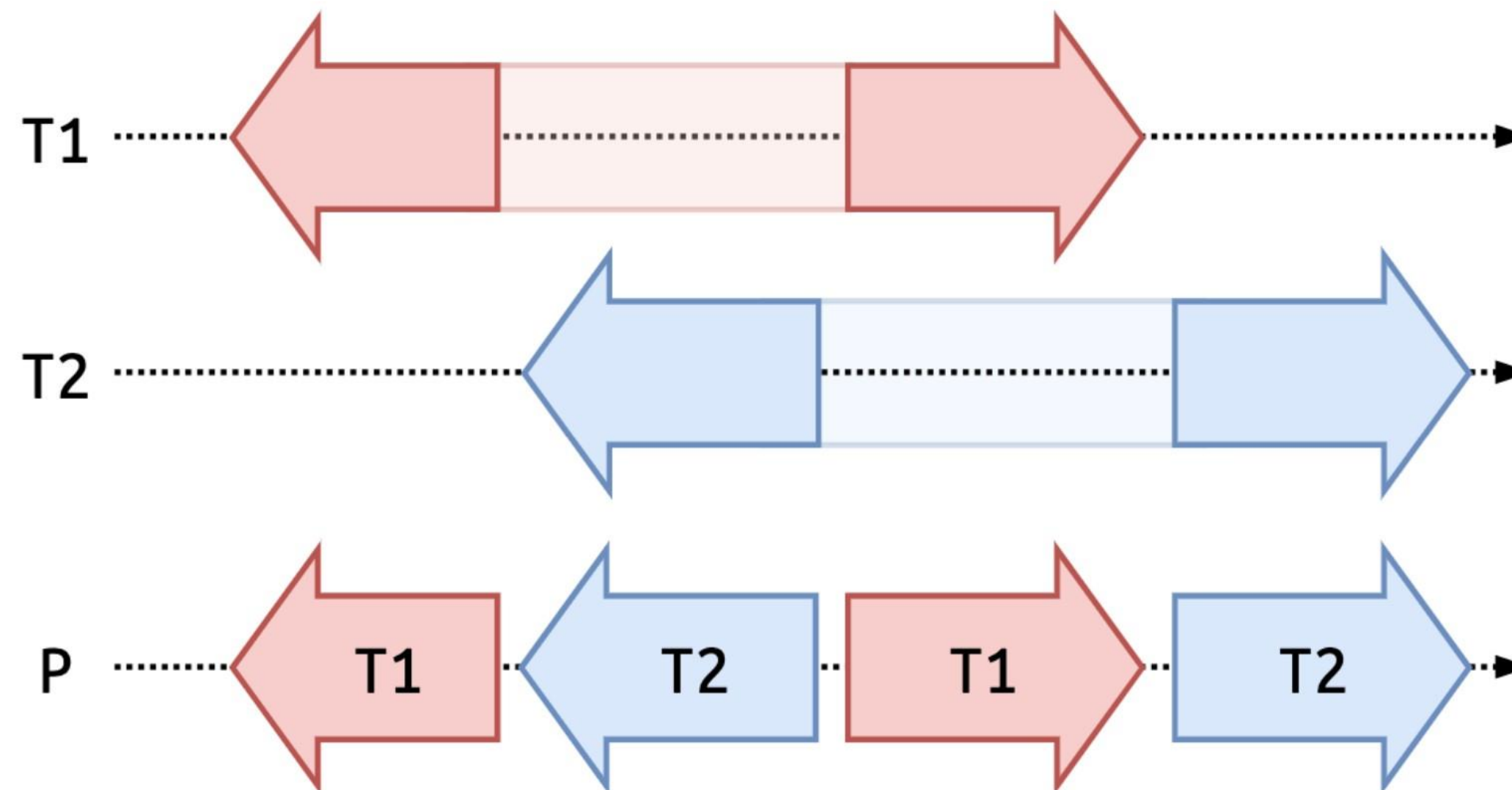
```
int pthread_create(pthread_t *thread, const pthread_attr_t  
                  *attr, void *(*start routine) (void *), void  
                  *arg);
```

```
int pthread_join(pthread_t thread, void **retval);
```

Создать поток / дождаться завершения потока

\$ gcc --fsanitize=thread main.c – запуск с тред-санитайзером

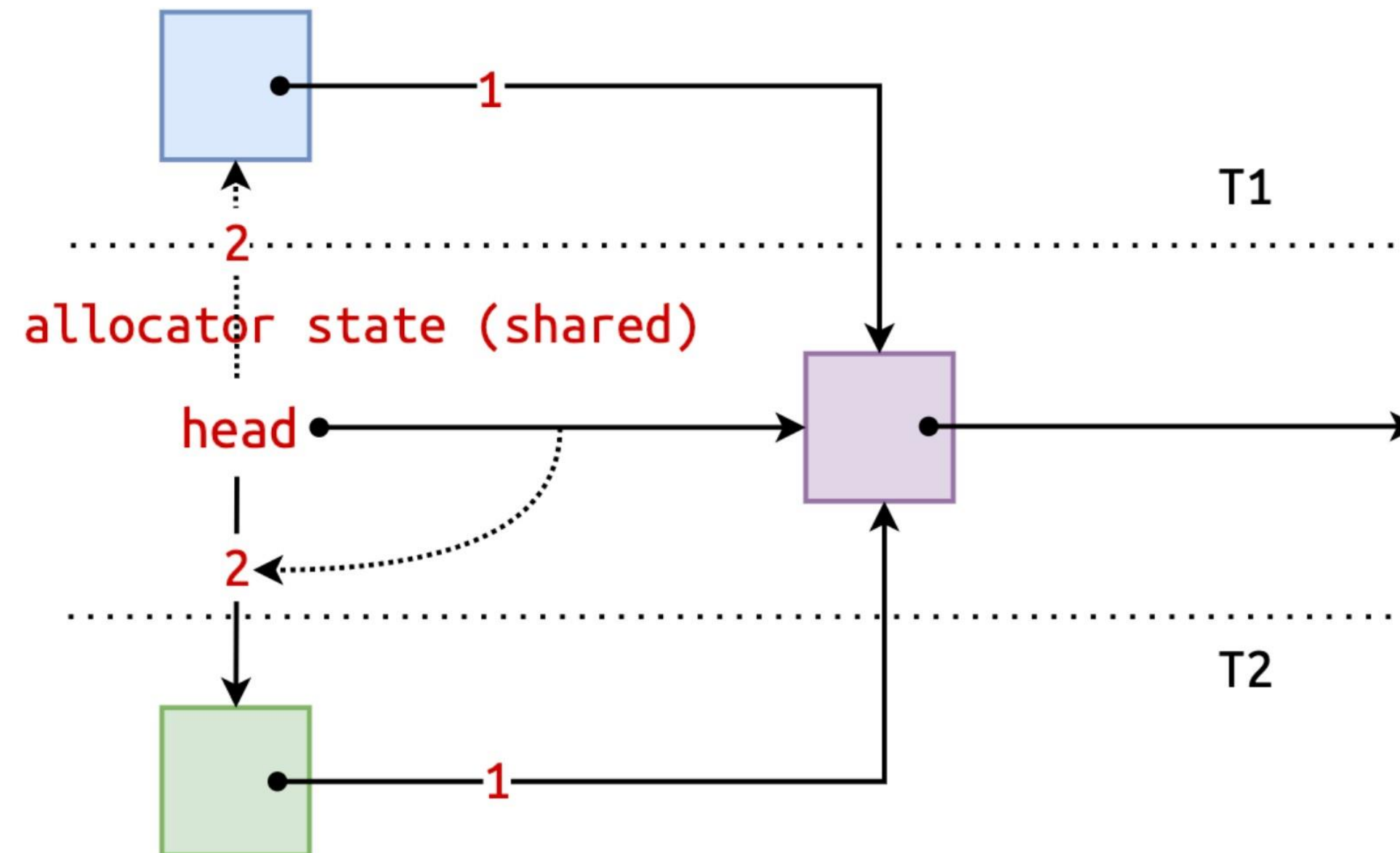
# Конкурирующие операции



Планировщик переключает потоки, причем сами потоки этих переключений не наблюдают.

# Race condition

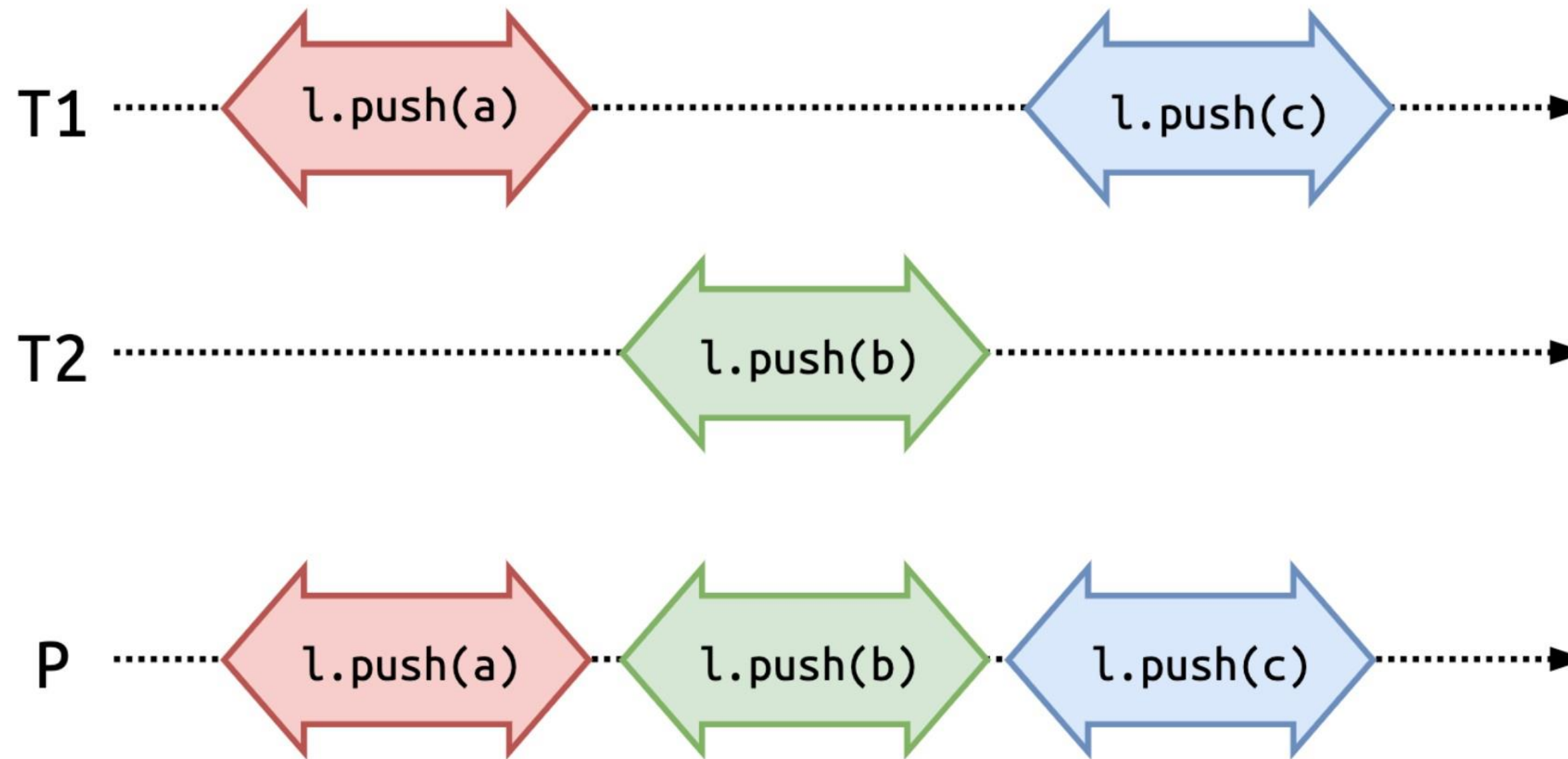
1. `block_node->next = head`
2. `head = block_node`



Различайте **race condition** и **data race**!

# Mutual exclusion

Хотим избавиться от конкуренции, упорядочить операции из разных потоков:

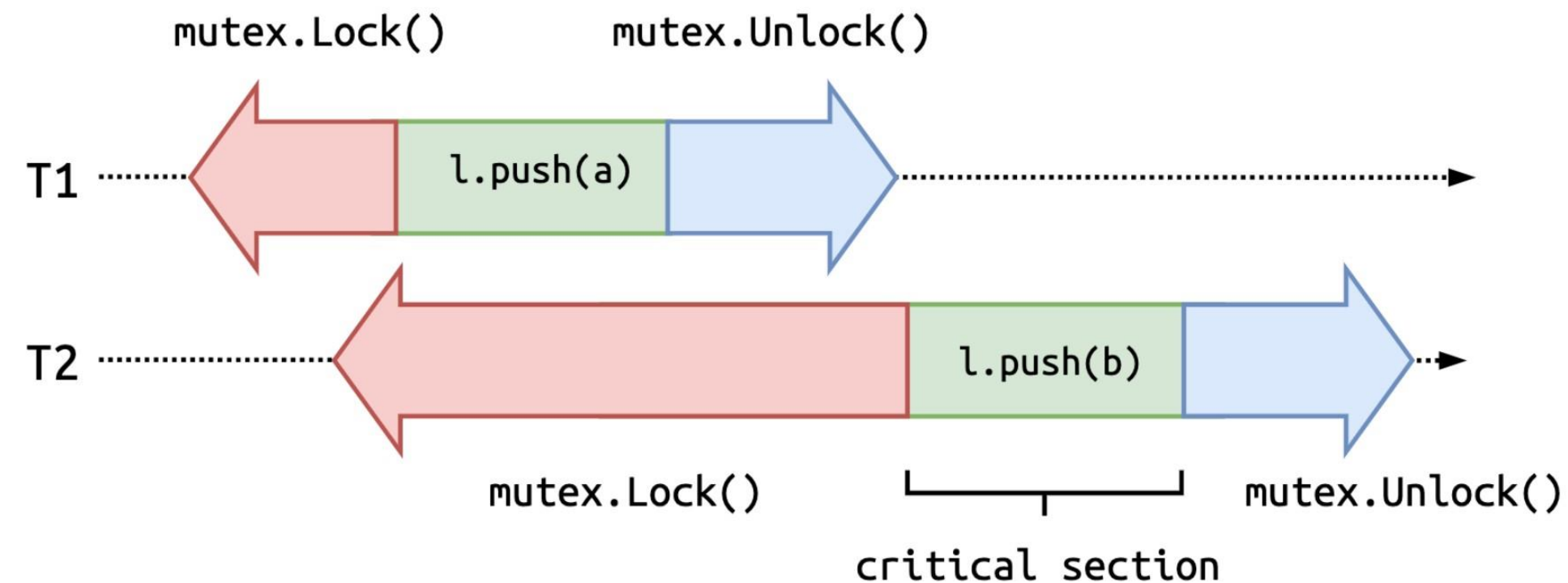


# Mutex

Поток захватывает мьютекс / берет блокировку и входит в критическую секцию. В конце – отпускает мьютекс.

```
mutex.Lock()  
// critical section starts here  
...  
block_node->next = head_  
head_ = block_node;  
...  
mutex.Unlock()
```

Mutex – **mutual exclusion**



# Свойства (гарантии)

## Взаимное исключение:

Между парными вызовами `mutex.Lock()` и `mutex.Unlock()` может находиться только один поток.

## Свобода от взаимной блокировки:

Если один или несколько потоков пытаются захватить свободный мьютекс, то один из вызовов `mutex.Lock()` должен завершиться.

Второе свойство называют **гарантией прогресса**.



# Safety и Liveness

Взаимное исключение – это свойство **safety**, оно говорит, что **никогда** не происходит **ничего плохого**.

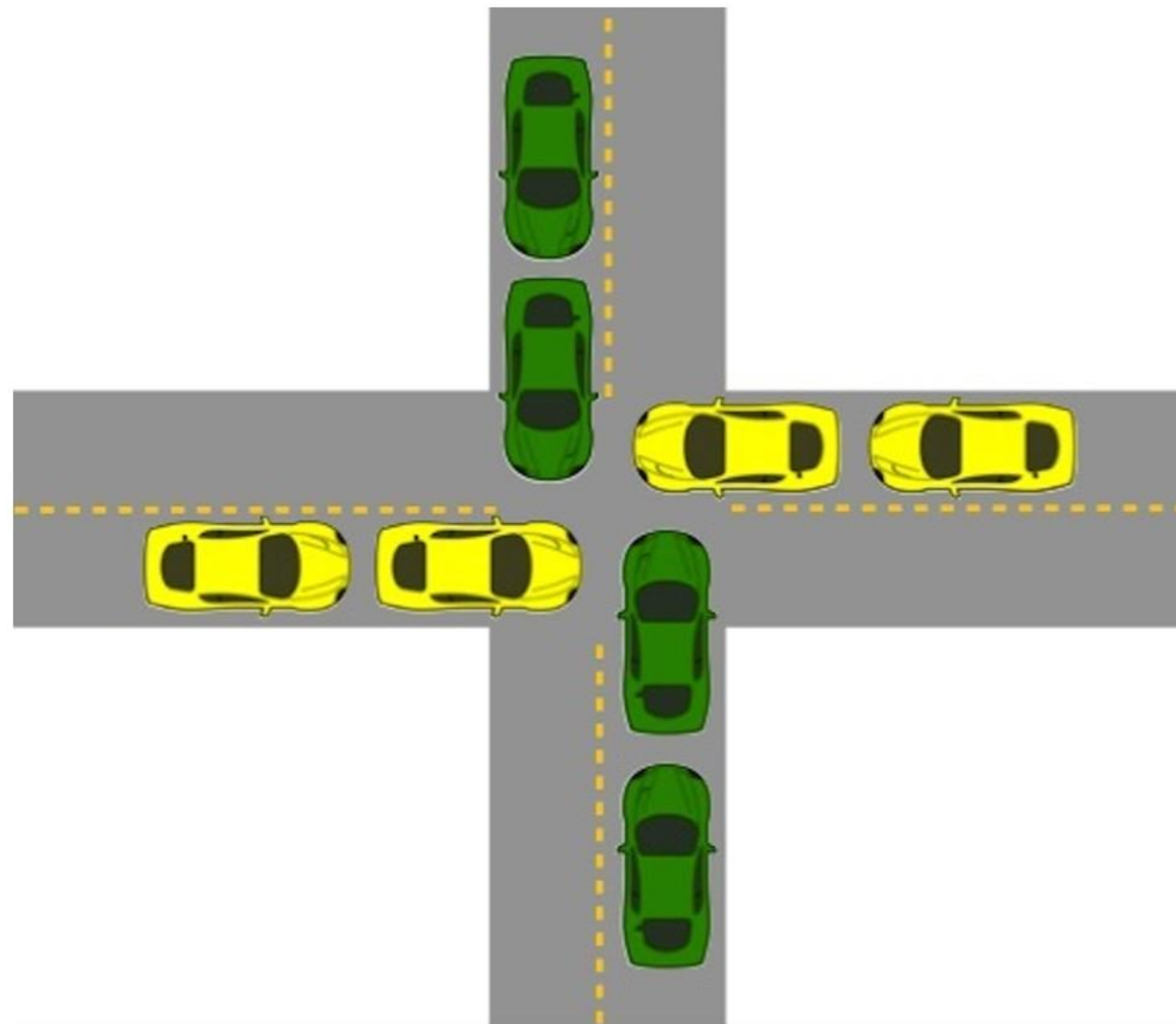
Свобода от взаимной блокировки – это свойство **liveness**, оно говорит о том, что **когда-нибудь** происходит **что-то хорошее**.

**Safety** свойства нарушаются на конечных исполнениях, а **liveness** – на бесконечных.



# Deadlock

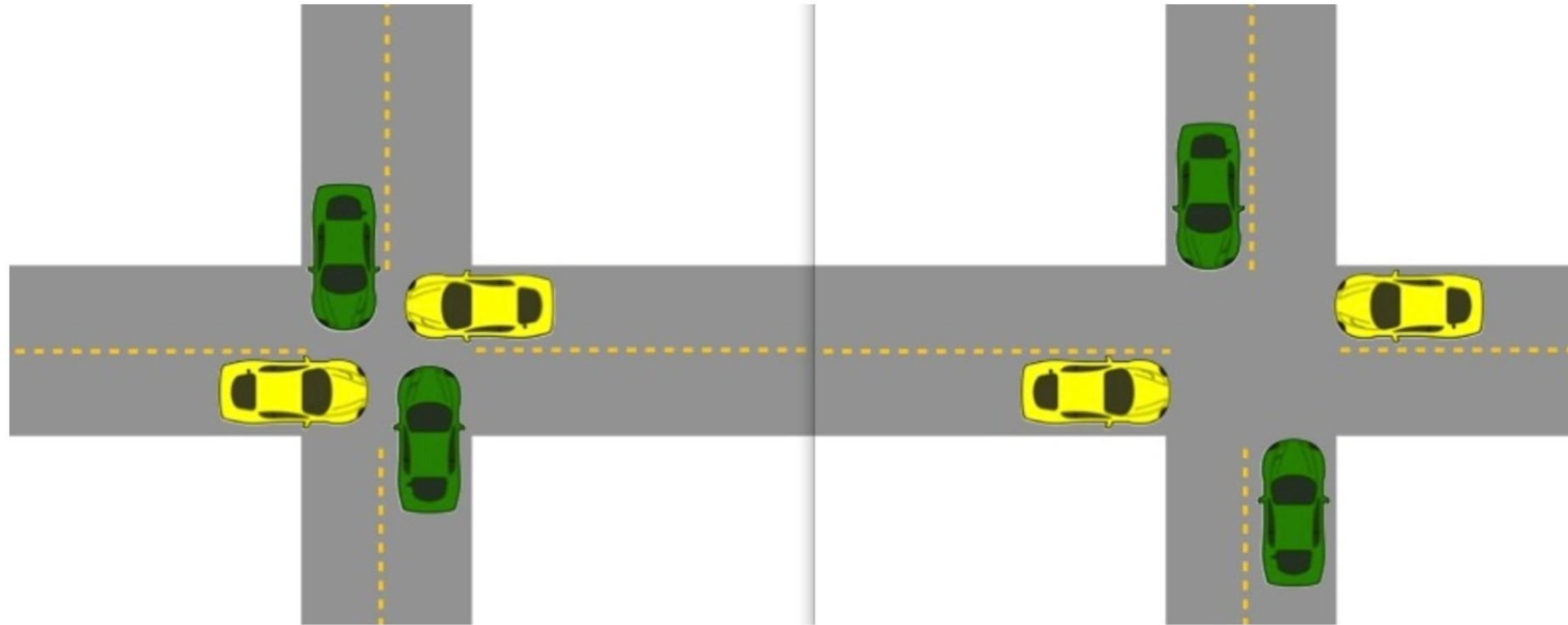
Тяжелая форма взаимной блокировки – **дэдлок (deadlock)**.



Терминальное состояние: потоки не выйдут из него, что бы ни делал планировщик.

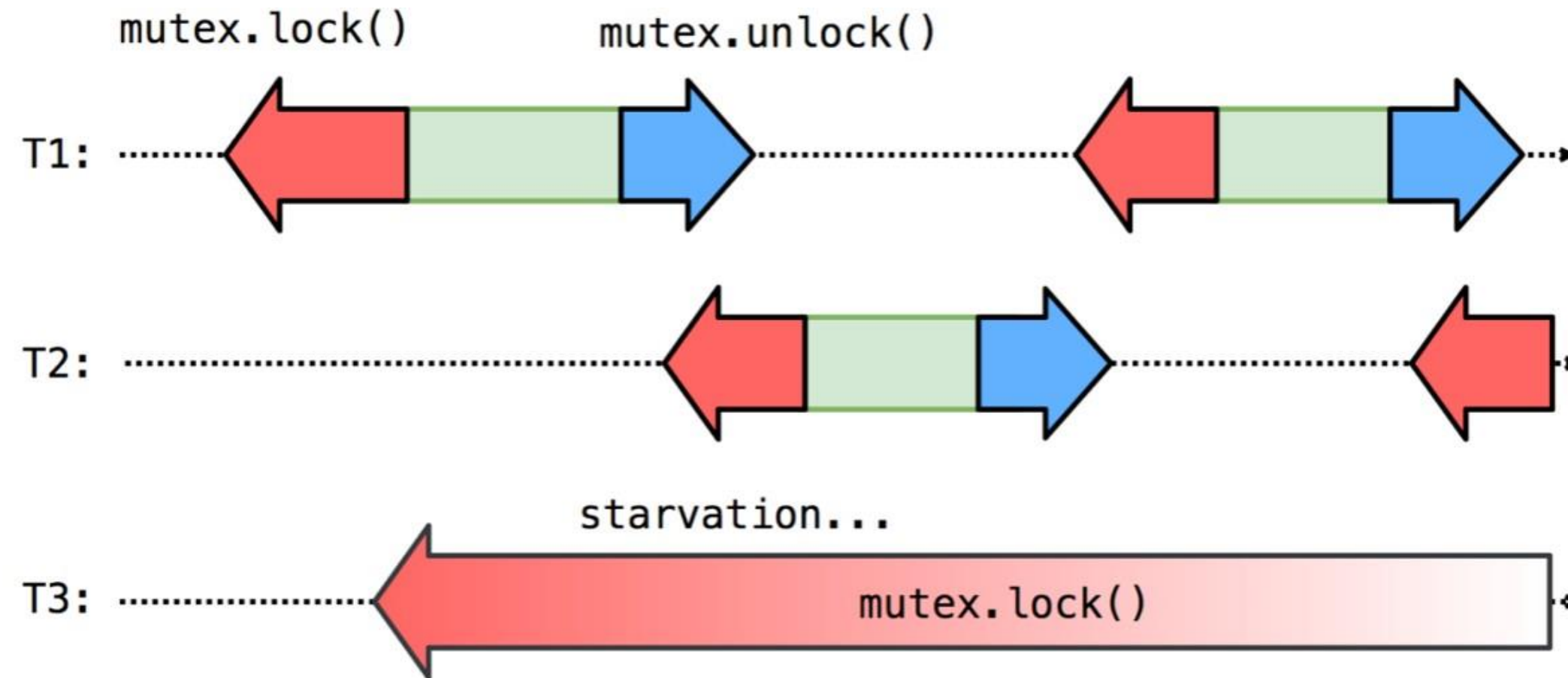
# Livelock

Легкая форма взаимной блокировки – лайвлок (livelock).



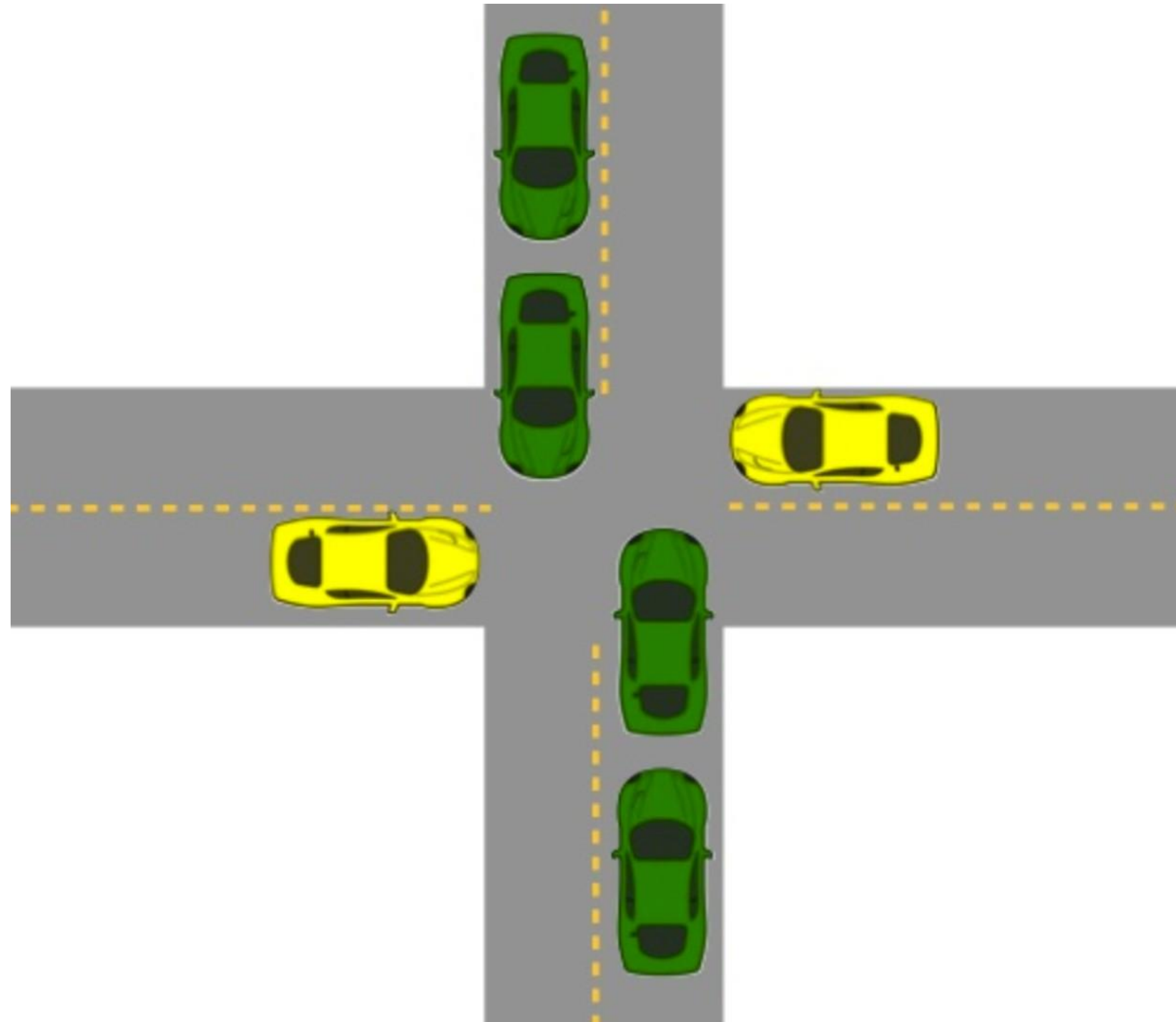
Потоки мешают **прогрессу** друг друга, но при удачном планировании разойдутся.

# Глобальный прогресс

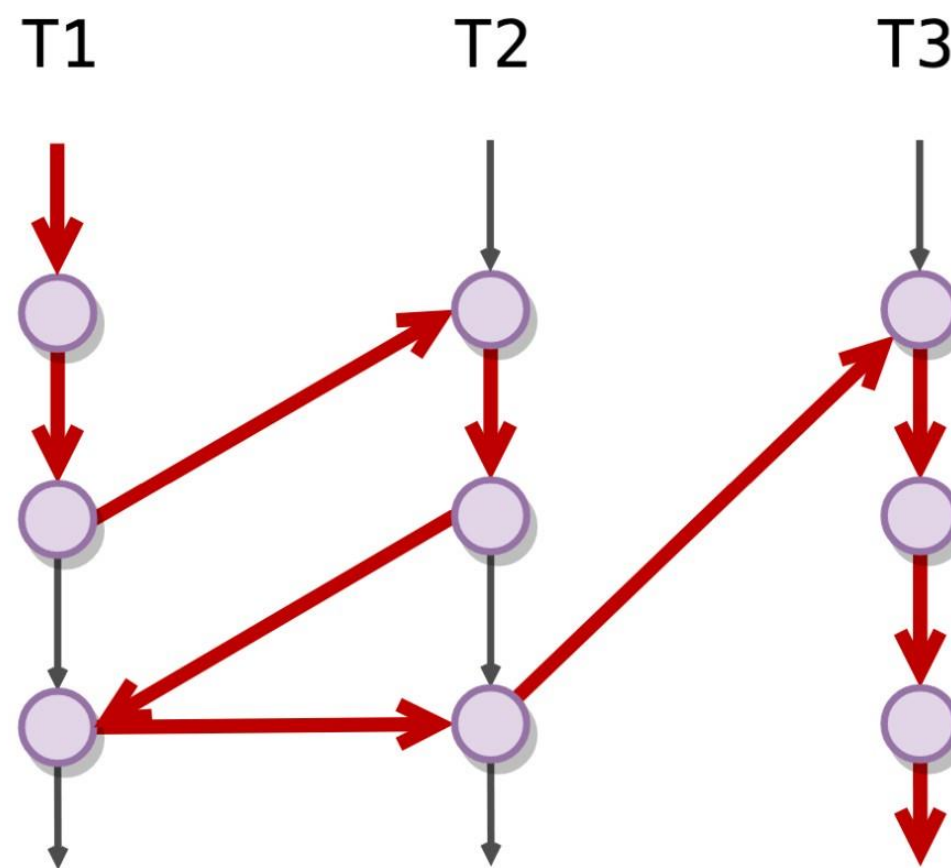


В то же время какой-то неудачливый поток может провести в вызове `mutex.Lock()` бесконечное время.

# Starvation



# Interleaving



Будем моделировать исполнение потоков в **модели чередования** на одном процессоре.

Игнорируем параллельность!

# pthread\_mutex

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *mutexattr);  
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);  
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

[Mutex](#)



# pthread\_condvar

```
#include <pthread.h>
```

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
int pthread_cond_init(pthread_cond_t *cond,  
                      pthread_condattr_t *cond_attr);
```

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

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

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

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,  
                           const struct timespec *abstime);
```

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

## Condvar

signal – restart no more than one thread waiting on a cv

broadcast – restart all thread waiting on a cv

Wait – atomically unlocks mutex and waits for a signal. Before starting, re-acquires mutex

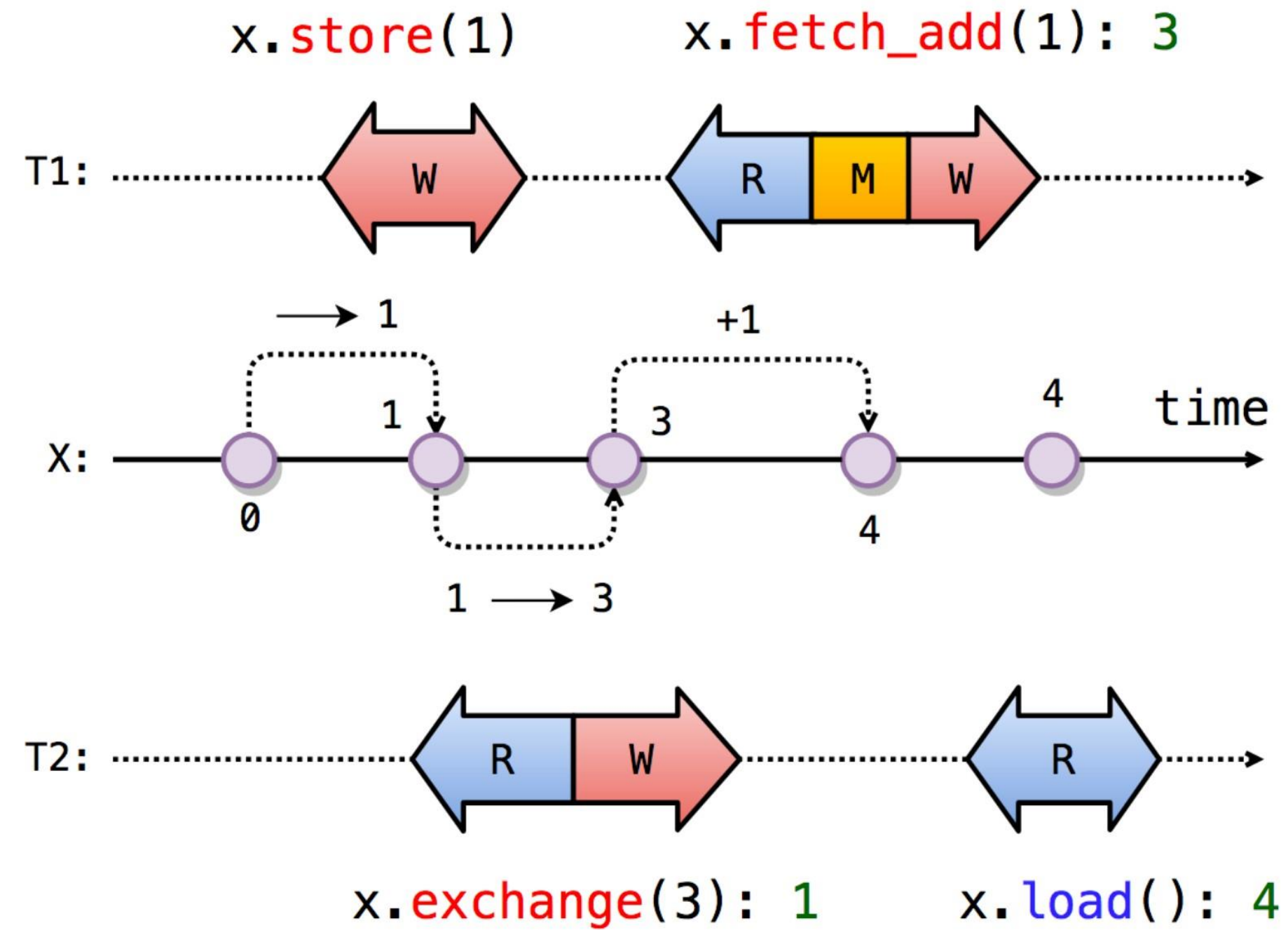
# Condvar usage notes

- Перед засыпанием на кондваре мьютекс всегда должен быть захвачен
- Wait всегда должен вызываться в цикле из-за spurious wake-ups

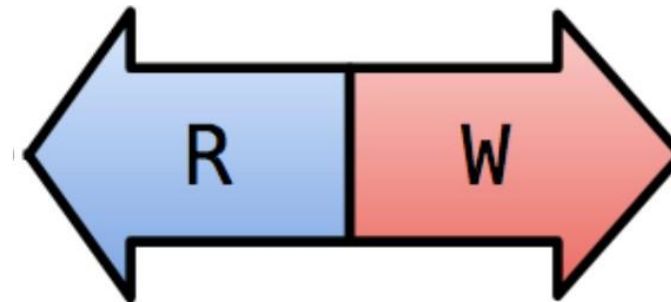
```
pthread_mutex_lock(&mutex);  
while (ready == 0) { // Wait for the condition to be true  
    pthread_cond_wait(&cond, &mutex);  
}  
printf("Consumer: Consuming data...\n");  
pthread_mutex_unlock(&mutex);
```

# АТОМИКИ

`std::atomic<int>`



# Exchange



```
old_value = atomic.exchange(new_value)
```

**Атомарно** обменивает содержимое **ячейки памяти** и **регистра процессора**:

Псевдокод:

```
atomically {  
    old_value = atomic.load() // Read  
    atomic.store(new_value) // Write  
    return old_value  
}
```

# Compare-and-Swap (CAS)

compare\_exchange

`bool compare_exchange_strong( T& expected, T desired)`

Atomically compares the **value representation** of `*this` with that of `expected`. If those are bitwise-equal, replaces the former with `desired` (performs read-modify-write operation). Otherwise, loads the actual value stored in `*this` into `expected` (performs load operation).

# Memory models

SB

Proc 0	Proc 1
MOV [x] ← 1 MOV EAX ← [y]	MOV [y] ← 1 MOV EBX ← [x]
Allowed Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	



# Yield

Если цикл ожидания затянулся, то стоит уступить ядро другому потоку:

**`std::this_thread::yield()`**

Передаем управление планировщику операционной системы и перемещаемся в конец очереди на исполнение.

`pause` – **инструкция**,

`yield` – под капотом **системный вызов** `sched_yield`

**`*yield`** – уступить