

Concurrency: Signaling and Condition Variables

Questions Answered in this Lecture:

- How do we make fair locks perform better?
- How do we notify threads of that some condition has been met?

Recall: Ticket Lock Implementation

```
typedef struct __lock_t {  
    int ticket;  
    int turn;  
}  
  
void lock_init(lock_t *lock)  
{  
    lock->ticket = 0;  
    lock->turn   = 0;  
}
```

```
void acquire(lock_t *lock) {  
    int myturn = FAA(&lock->ticket);  
    while (lock->turn != myturn); // spin  
}  
  
void release (lock_t *lock) {  
    FAA(&lock->turn);  
}
```

Spinlock Performance

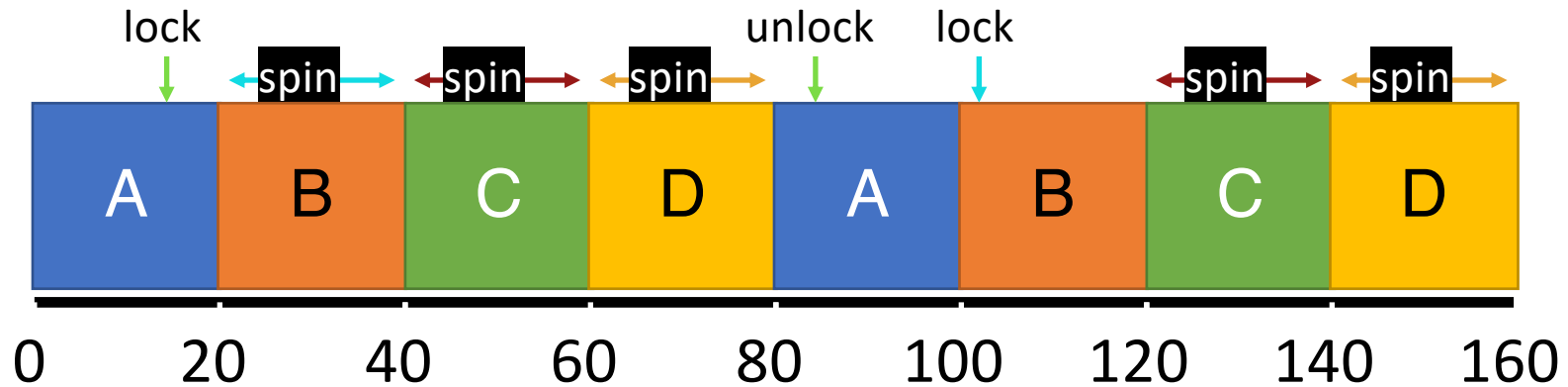
Fast when...

- many CPUs
- locks held a short time
- advantage: avoid context switch

Slow when...

- one CPU
- locks held a long time
- disadvantage: spinning is wasteful

CPU Scheduler is Ignorant



CPU scheduler may run **B** instead of **A**
even though **B** is waiting for **A**

Ticket Lock with `yield()`

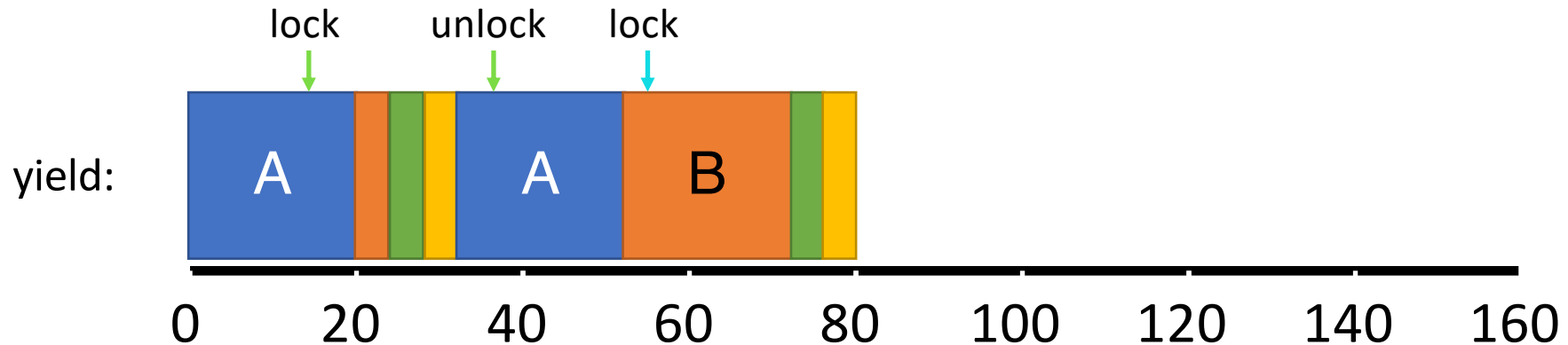
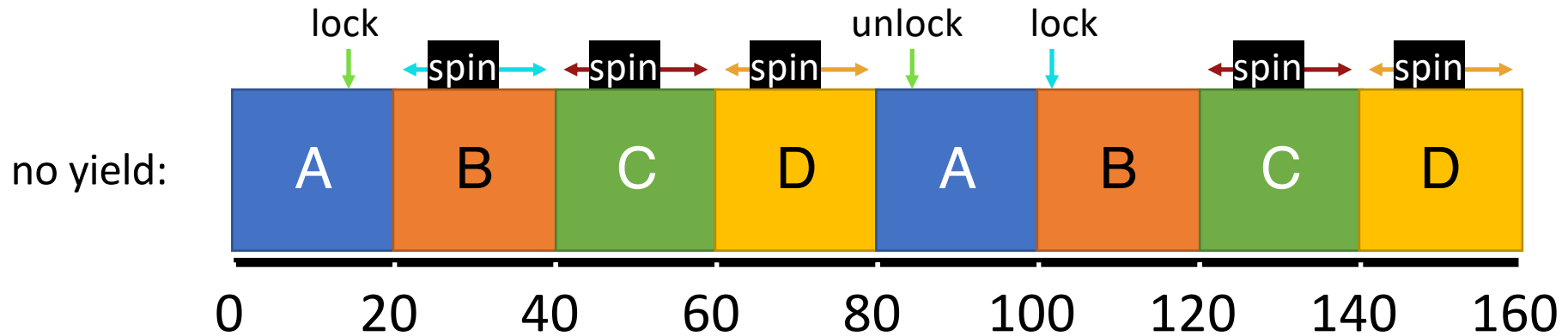
```
typedef struct __lock_t {  
    int ticket;  
    int turn;  
}
```

```
void lock_init(lock_t *lock)  
{  
    lock->ticket = 0;  
    lock->turn = 0;  
}
```

```
void acquire(lock_t *lock) {  
    int myturn = FAA(&lock->ticket);  
    while (lock->turn != myturn)  
        yield();  
}
```

```
void release (lock_t *lock) {  
    FAA(&lock->turn);  
}
```

yield() Instead of spin



Spinlock Performance

Waste...

Without yield: $O(\text{threads} * \text{time_slice})$

With yield: $O(\text{threads} * \text{context_switch})$

So even with yield, spinning is slow with high thread contention

Next improvement: Block and put thread on waiting queue instead of spinning

How do we evaluate lock performance?

- **Fairness:**

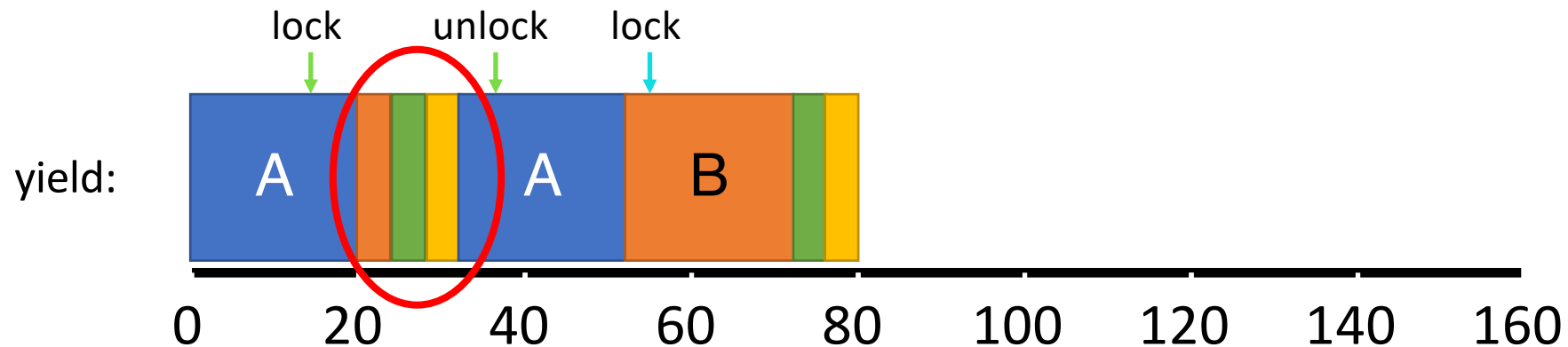
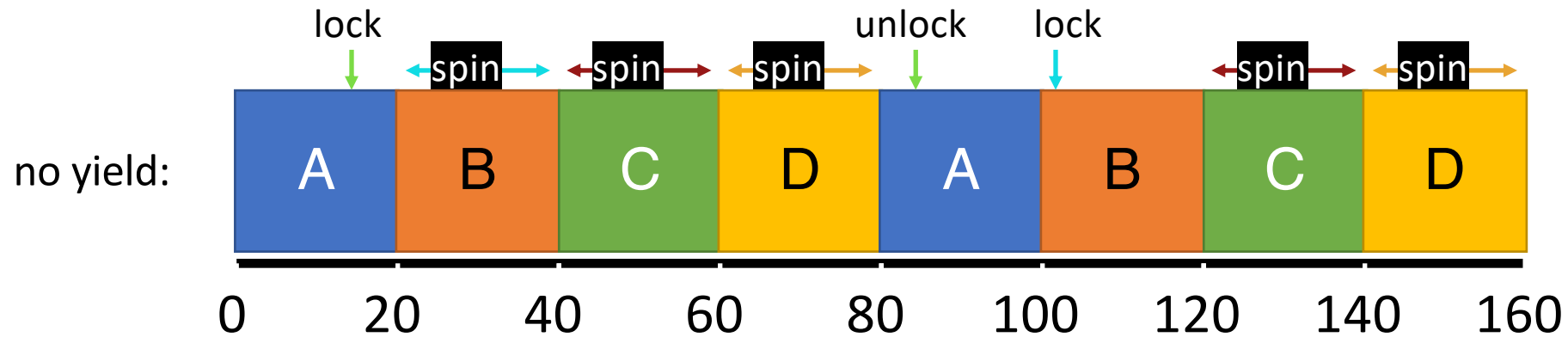
- do some processes wait more than others?
- another way: do processes get lock in order requested?

- **Performance:** it depends...

- Is the lock *contended?* (many threads attempting to acquire lock)
- Or is it *uncontended?* (likely to get the lock if we request it)

Why is `yield()` useful?

How can it affect performance?



Lock implementation: block when waiting

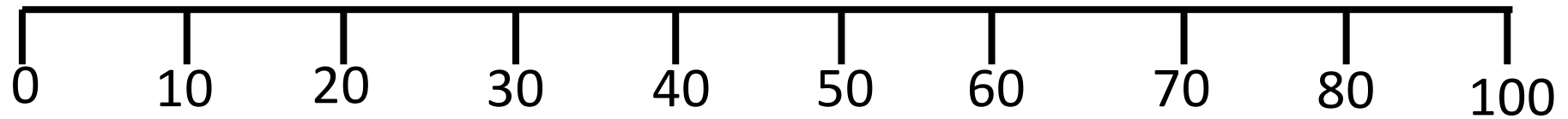
- Lock ***implementation*** removes waiting threads from the ready queue (e.g. with `park()` and `unpark()`)
- Scheduler only runs **ready** threads
- ***Separates concerns*** (programmer doesn't need to deal with yield vs not/spinning issue)
- **Quiz:** where should locks be implemented?

Example

Ready: {A, B, C, D}

Waiting: {}

Running: {}

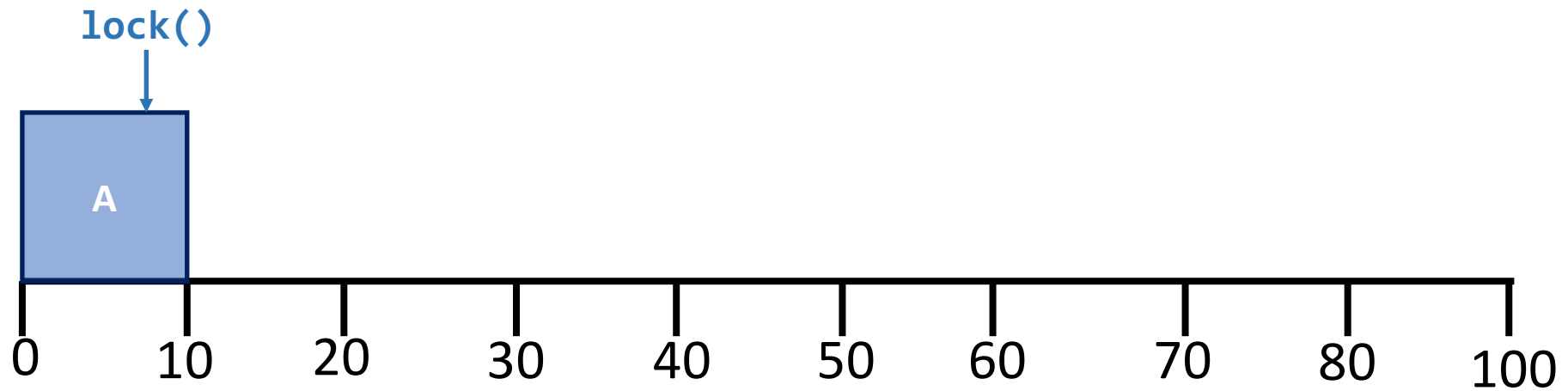


Example

Ready: {B, C, D}

Waiting: {}

Running: {A}

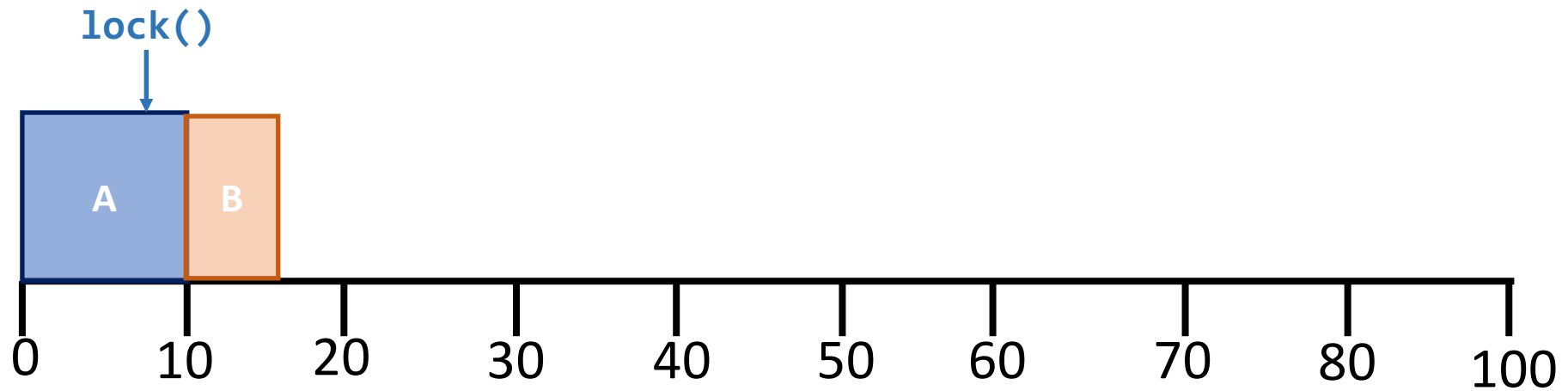


Example

Ready: {A, C, D}

Waiting: {}

Running: {B}

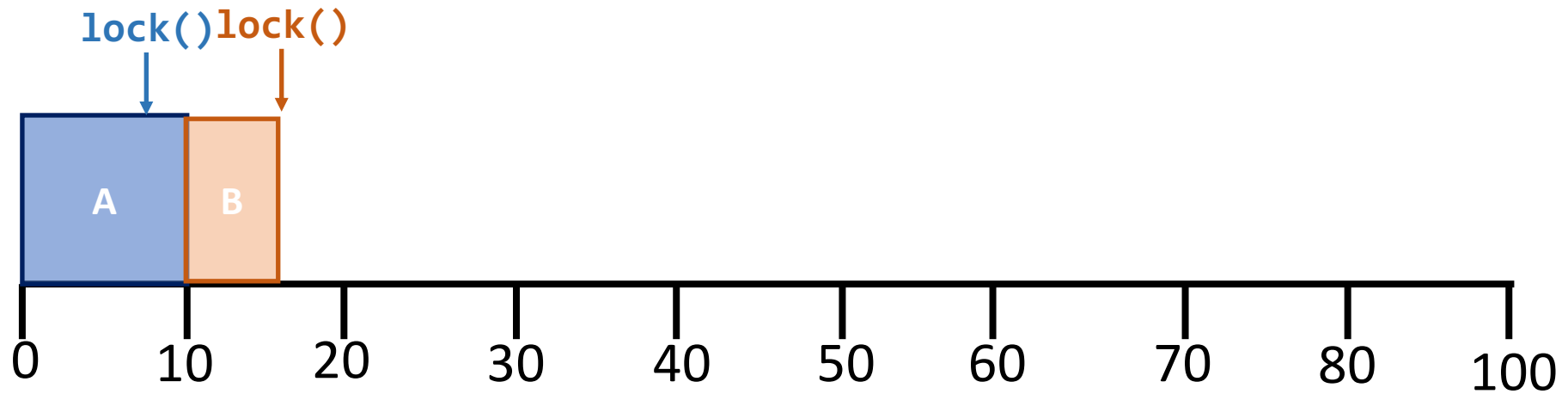


Example

Ready: {A, C, D}

Waiting: {B}

Running: {}

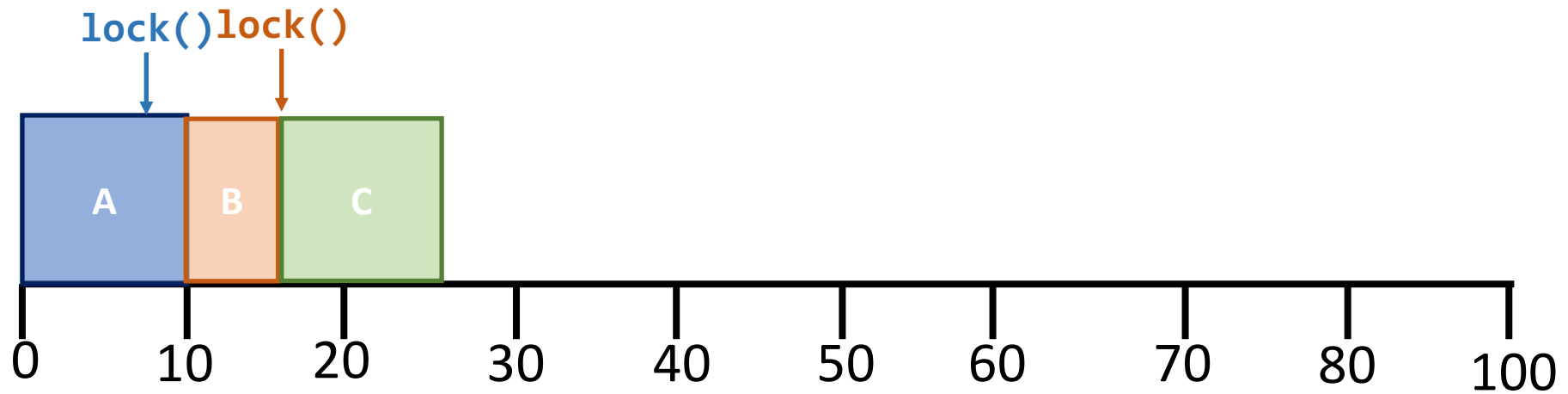


Example

Ready: {A, D}

Waiting: {B}

Running: {C}

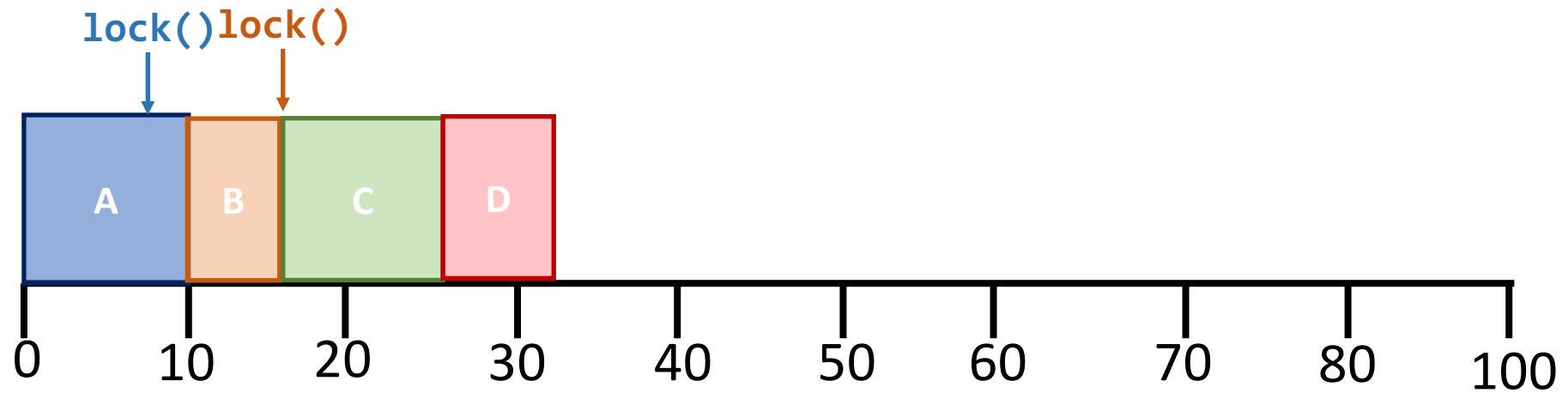


Example

Ready: {A, C}

Waiting: {B}

Running: {D}

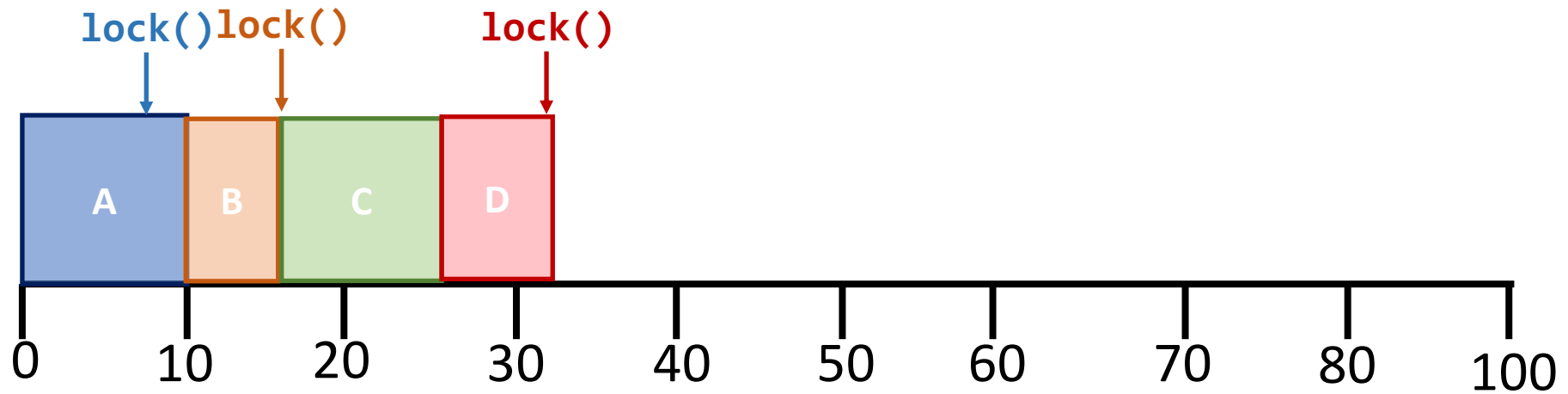


Example

Ready: {A, C}

Waiting: {B, D}

Running: {}

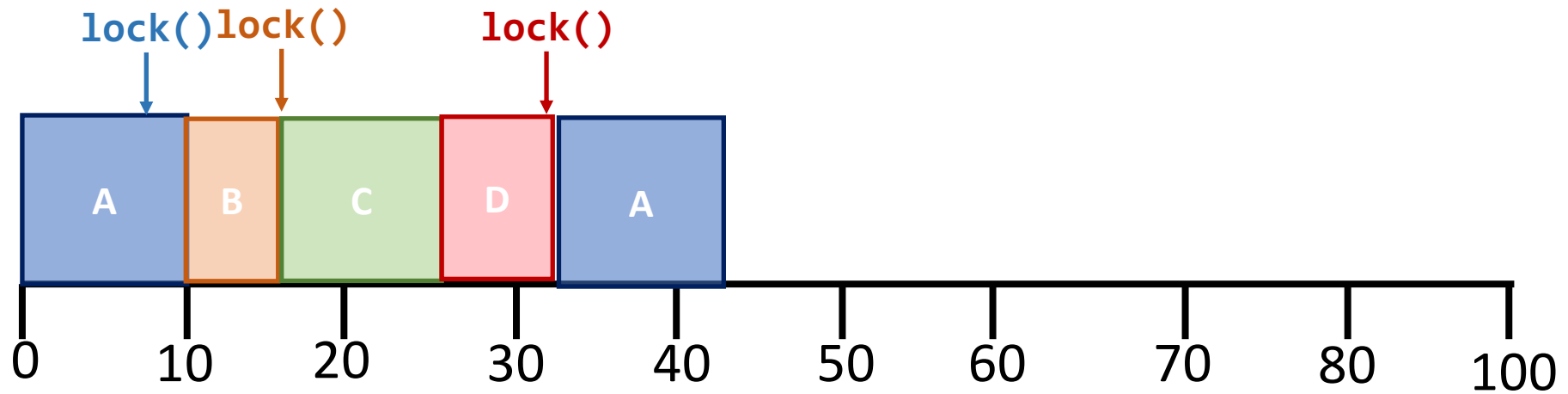


Example

Ready: {C}

Waiting: {B, D}

Running: {A}

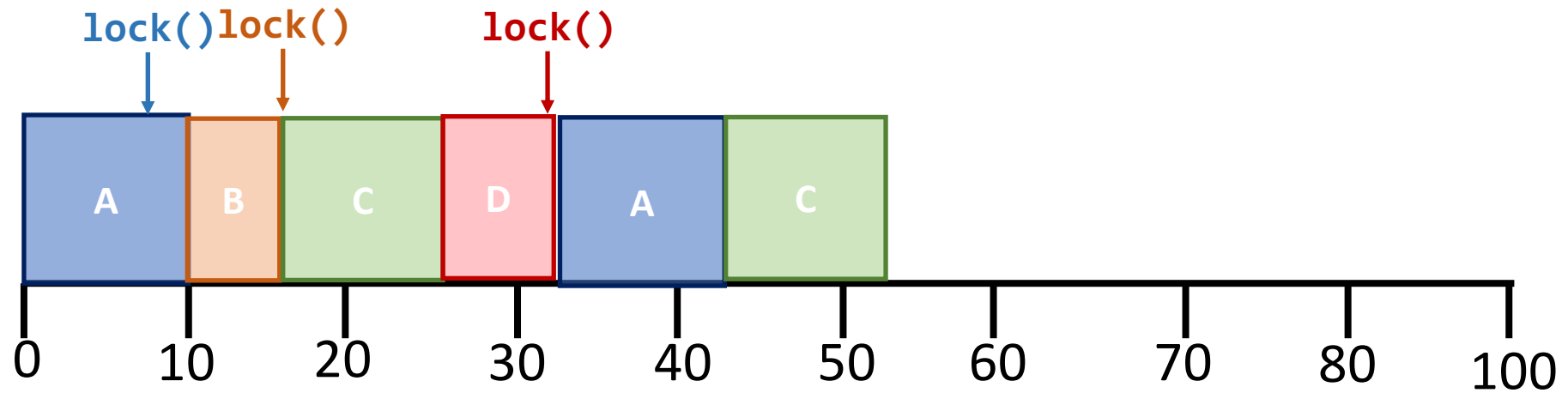


Example

Ready: {A}

Waiting: {B, D}

Running: {C}

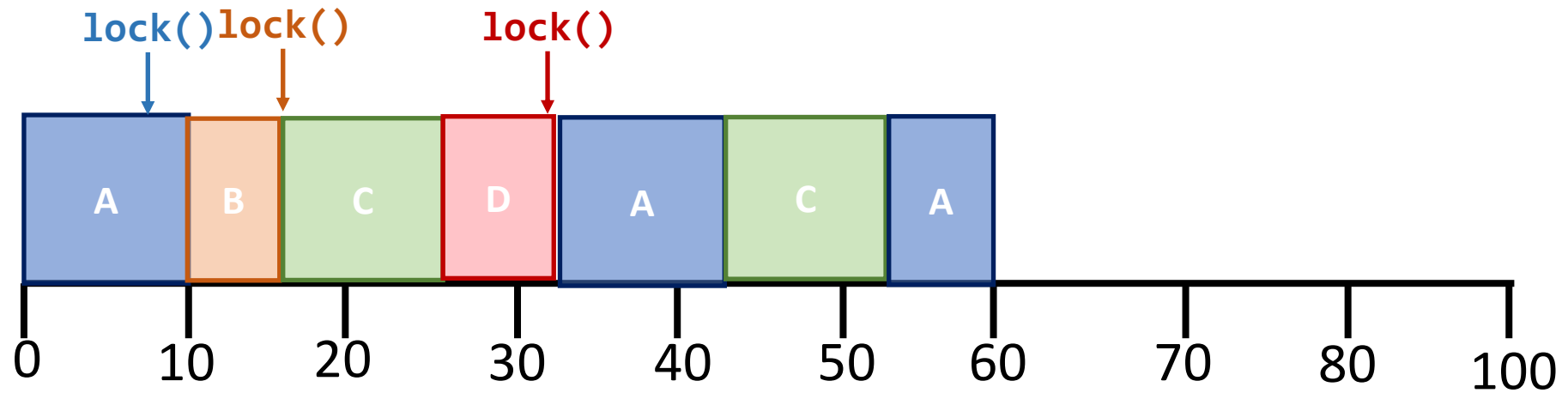


Example

Ready: {C}

Waiting: {B, D}

Running: {A}

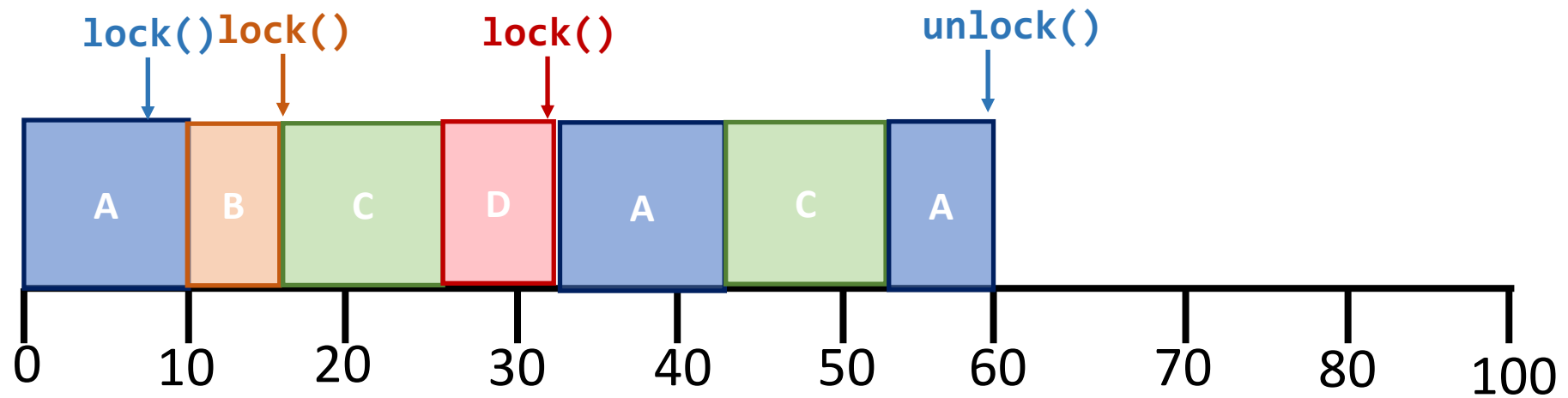


Example

Ready: {C, B, D}

Waiting: {}

Running: {A}

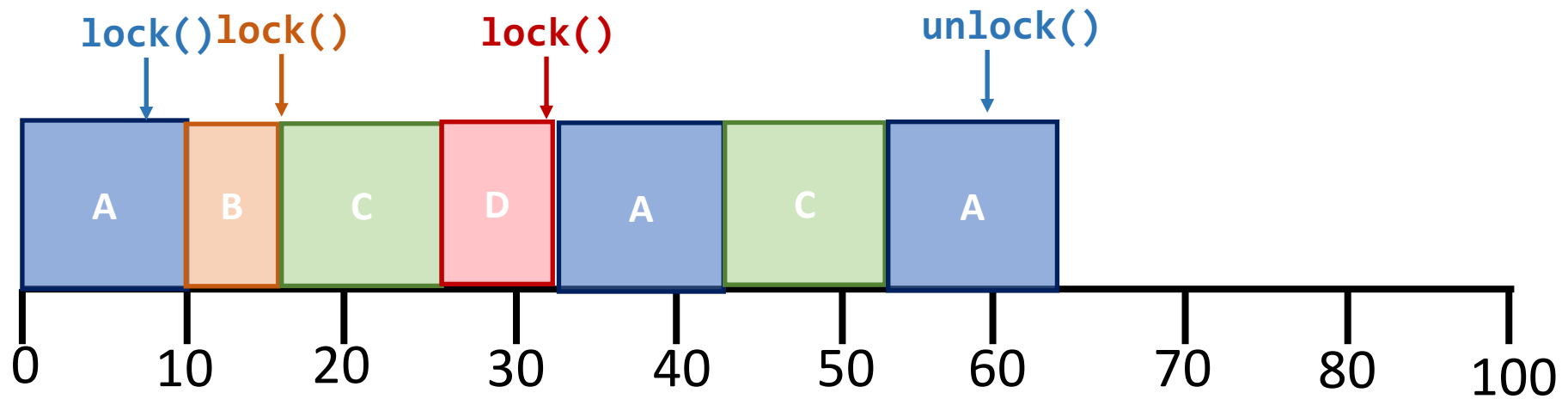


Example

Ready: {C, B, D}

Waiting: {}

Running: {A}

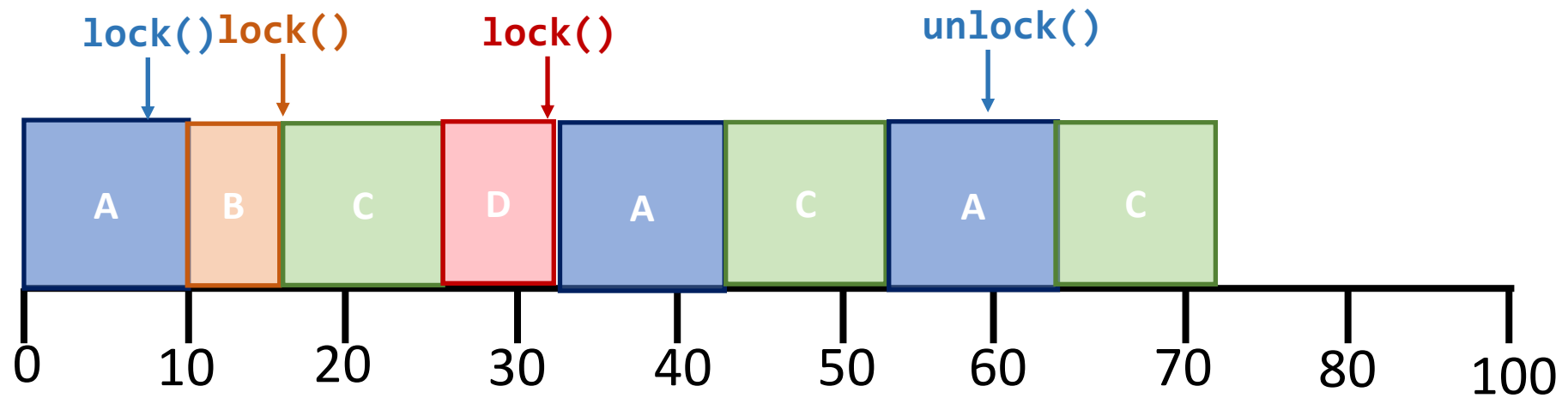


Example

Ready: {B, D, A}

Waiting: {}

Running: {C}

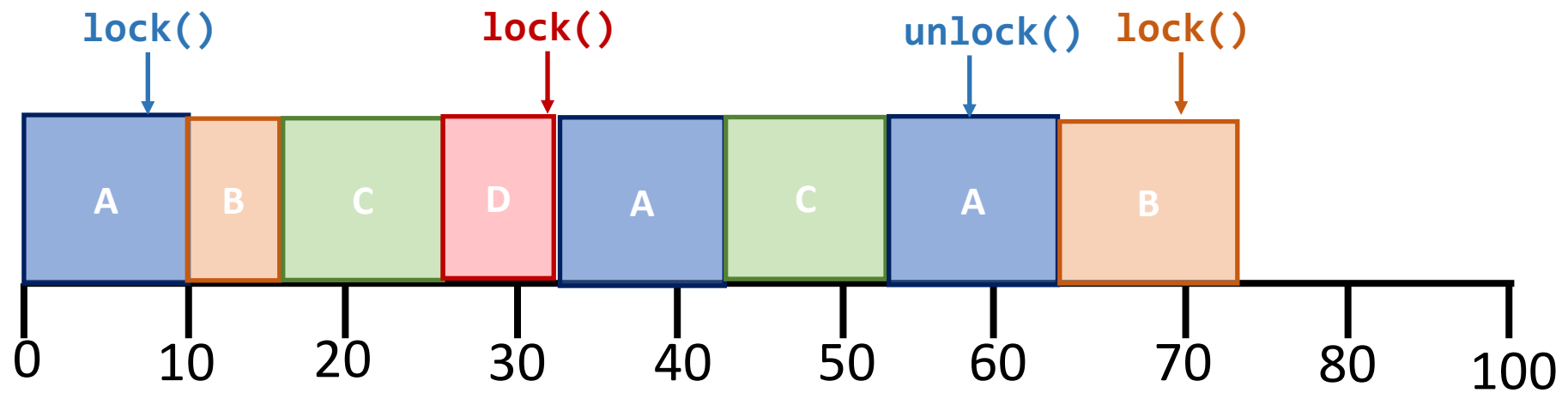


Example

Ready: {D, A, C}

Waiting: {}

Running: {B}



What do we get?

- ***Threads aren't contending on the lock*** when they shouldn't be (mostly)
- **Fewer context switches**

Lock: block when waiting

```
typedef struct {
    bool lock = false;
    bool guard = false;
    queue_t q;
} lock_t;
```

1. Why do we need guard?
2. Why is it OK to spin on guard?
3. In `release()`, why not set `lock=false`?
4. What is the race condition?

```
void acquire(lock_t *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        enqueue(l->q, tid);
        l->guard = false;
        park(); // blocked
    } else {
        l->lock = true;
        l->guard = false;
    }
}
```

```
void release(lock_t *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q))
        l->lock=false;
    else
        unpark(dequeue(l->q));
    l->guard = false;
}
```

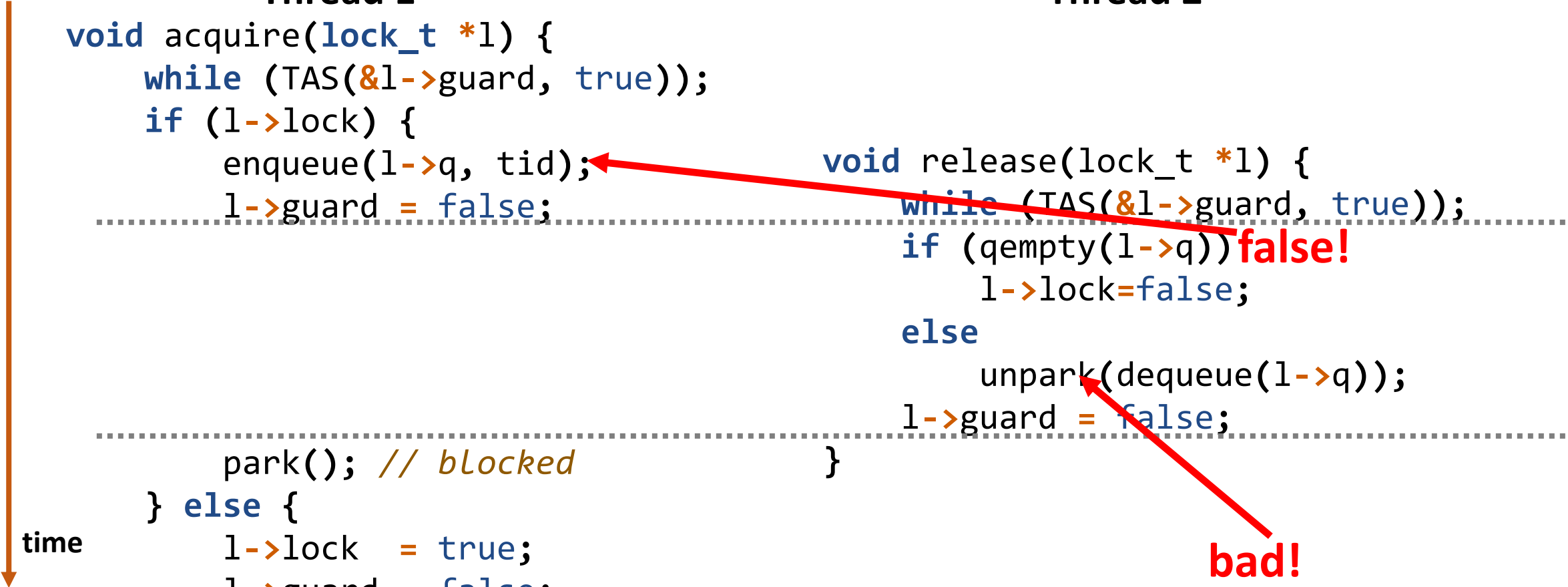
Race!

Thread 1

```
void acquire(lock_t *l) {  
  while (TAS(&l->guard, true));  
  if (l->lock) {  
    enqueue(l->q, tid);  
    l->guard = false;  
    .....  
    park(); // blocked  
  } else {  
    l->lock = true;  
    l->guard = false;  
  }  
}
```

Thread 2

```
void release(lock_t *l) {  
  while (TAS(&l->guard, true));  
  if (qempty(l->q)) false!  
    l->lock=false;  
  else  
    unpark(dequeue(l->q));  
  l->guard = false;  
}
```



bad!

Lock: the fix

```
typedef struct {
    bool lock = false;
    bool guard = false;
    queue_t q;
} lock_t;
```

1. Why does this fix the race?
2. What does the scheduler need to do here?
3. Why can't we just hold the guard when we park()?

```
void acquire(lock_t *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        enqueue(l->q, tid);
        setpark(); // notify of plan
        l->guard = false;
        park(); // unless unpark()
    } else {
        l->lock = true;
        l->guard = false;
    }
}

void release(lock_t *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q))
        l->lock=false;
    else
        unpark(dequeue(l->q));
    l->guard = false;
}
```

Spin-waiting vs. Blocking

- Each approach has *tradeoffs*:

- **Uniprocessor**

- Waiting process is scheduled -> process holding lock isn't
- Waiting process should always be descheduled
- Associate queue of waiters with each lock

- **Multiprocessor**

- Waiting process is scheduled -> **process holding lock might also be**
- Spin or block depends on **how long (t) before lock is released**
- Lock released quickly? `spin()`
- Lock taken for long time? `block()`

we'll define this relative to the context switch time

Oracle

- Suppose we know how long (t) a lock will be held
- And suppose **we make our decision to block or spin based on C , the cost of a context switch**

$$action = \begin{cases} t \leq C & \rightarrow spin \\ t > C & \rightarrow block \end{cases}$$

BUT: we have to know the future!

Applying Bounds

- The theory: **bound our worst-case performance using actual wait time over optimal (oracle)**
- Consider:

$$t \leq C$$

What would the oracle do?

What can we do (without knowing future)?

We pay $2C$, because we wait an extra C . $2C/C = 2 \Rightarrow$ 2-competitive algorithm!

$$t > C$$

What would the oracle do?

What can we do?

Concurrency Goals

- **Mutual Exclusion**

- Keep two threads from executing in a critical section concurrently
- We solved this with *locks*

- **Dependent Events**

- We want a thread to wait until some particular event has occurred
- Or some condition has been met
- Solved with *condition variables* and *semaphores*

*“I just dropped in to see what
condition my condition was in”*

Example: join()

```
pthread_t p1, p2;  
pthread_create(&p1, NULL, mythread, "A");  
pthread_create(&p2, NULL, mythread, "B");  
// join waits for the threads to finish  
pthread_join(p1, NULL);  
pthread_join(p2, NULL);  
printf("Main: done\n [balance: %d]\n", balance);  
return 0;
```

Condition Variables

- **CV**: queue of waiting threads
- **B** waits for a signal on CV before running
 - `wait(CV, ...);`
- **A** sends `signal()` on CV when time for **B** to run
 - `signal(CV, ...);`

API

- `wait(cond_t * cv, mutex_t * lock)`
 - assumes lock is held when `wait()` is called (*why?*)
 - puts caller to sleep + releases the lock (atomically)
 - when awoken, reacquires lock before returning
- `signal(cond_t * cv)`
 - wake a *single* waiting thread (if ≥ 1 thread is waiting)
 - if there is no waiting thread, NOP

Join Implementation: Attempt #1

parent

```
void thread_join() {  
    mutex_lock(&m);    // x  
    cond_wait(&c, &m); // y  
    mutex_unlock(&m); // z  
}
```

child

```
void thread_exit() {  
    mutex_lock(&m);    // a  
    cond_signal(&c);  // b  
    mutex_unlock(&m); // c  
}
```

Join Implementation: Attempt #1

parent

```
void thread_join() {  
    mutex_lock(&m);    // x  
    cond_wait(&c, &m); // y  
    mutex_unlock(&m); // z  
}
```

child

```
void thread_exit() {  
    mutex_lock(&m);    // a  
    cond_signal(&c);  // b  
    mutex_unlock(&m); // c  
}
```

Example schedule:

parent

x

y

z

child

a

b

c

Join Implementation: Attempt #1

parent

```
void thread_join() {  
    mutex_lock(&m);    // x  
    cond_wait(&c, &m); // y  
    mutex_unlock(&m); // z  
}
```

child

```
void thread_exit() {  
    mutex_lock(&m);    // a  
    cond_signal(&c);  // b  
    mutex_unlock(&m); // c  
}
```

Example (bad) schedule:

parent

x y

child

a b c

parent will wait forever!

Rule of Thumb #1

- *Always associate some state* with the CV
- CVs are used to `signal()` when state *changes*
- If state is as required, *don't need to wait* for a `signal()`

Join Implementation: Attempt #2

```
parent  
void thread_join() {  
    mutex_lock(&m);        // w  
    if (!done)           // x  
        cond_wait(&c, &m); // y  
    mutex_unlock(&m);     // z  
}
```

```
child  
void thread_exit() {  
    done = 1           // a  
    cond_signal(&c);  // b  
}
```

Join Implementation: Attempt #2

```
parent  
void thread_join() {  
    mutex_lock(&m);           // w  
    if (!done)                // x  
        cond_wait(&c, &m); // y  
    mutex_unlock(&m);        // z  
}
```

```
child  
void thread_exit() {  
    done = 1;                // a  
    cond_signal(&c);         // b  
}
```

fixed!

Example schedule:

parent

w x y z

child

a b

Join Implementation: Attempt #2

```
parent  
void thread_join() {  
    mutex_lock(&m);           // w  
    if (!done)                // x  
        cond_wait(&c, &m); // y  
    mutex_unlock(&m);        // z  
}
```

```
child  
void thread_exit() {  
    done = 1                // a  
    cond_signal(&c);       // b  
}
```

sleeps forever!

Example (bad) schedule:



Join Implementation: Attempt #3 (correct)

parent

```
void thread_join() {  
    mutex_lock(&m);           // w  
    if (!done)                // x  
        cond_wait(&c, &m); // y  
    mutex_unlock(&m);        // z  
}
```

child

```
void thread_exit() {  
    mutex_lock(&m);           // a  
    done = 1;                 // b  
    cond_signal(&c);         // c  
    mutex_unlock(&m);        // d  
}
```

Join Implementation: Attempt #3 (correct)

```
parent  
void thread_join() {  
    mutex_lock(&m);           // w  
    if (!done)                // x  
        cond_wait(&c, &m); // y  
    mutex_unlock(&m);        // z  
}
```

```
child  
void thread_exit() {  
    mutex_lock(&m);           // a  
    done = 1;                // b  
    cond_signal(&c);         // c  
    mutex_unlock(&m);        // d  
}
```

fixed!

Example chedule:

parent

w x y

z

child

a b c d

Signaling between threads

Bounded Buffer (producer-consumer queue)

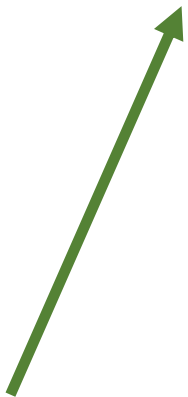


Signaling between threads

Bounded Buffer (producer-consumer queue)



t1: put()

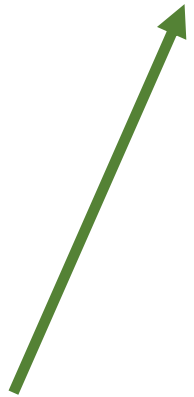


Signaling between threads

Bounded Buffer (producer-consumer queue)

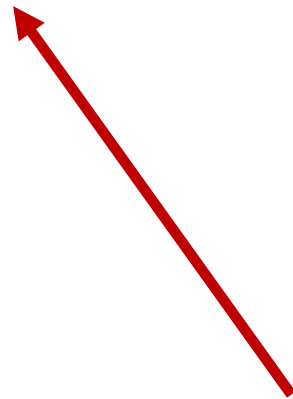


t1: put()



Signaling between threads

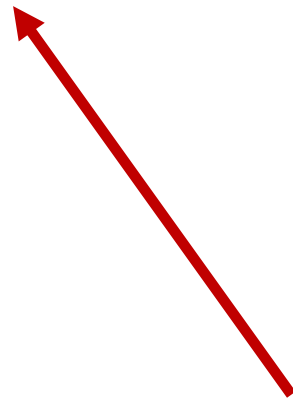
Bounded Buffer (producer-consumer queue)



t2: take()

Signaling between threads

Bounded Buffer (producer-consumer queue)



t2: take()

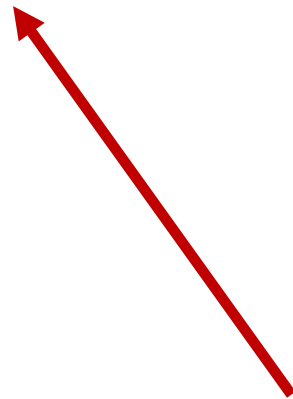
Signaling between threads

Bounded Buffer (producer-consumer queue)



Signaling between threads

Bounded Buffer (producer-consumer queue)

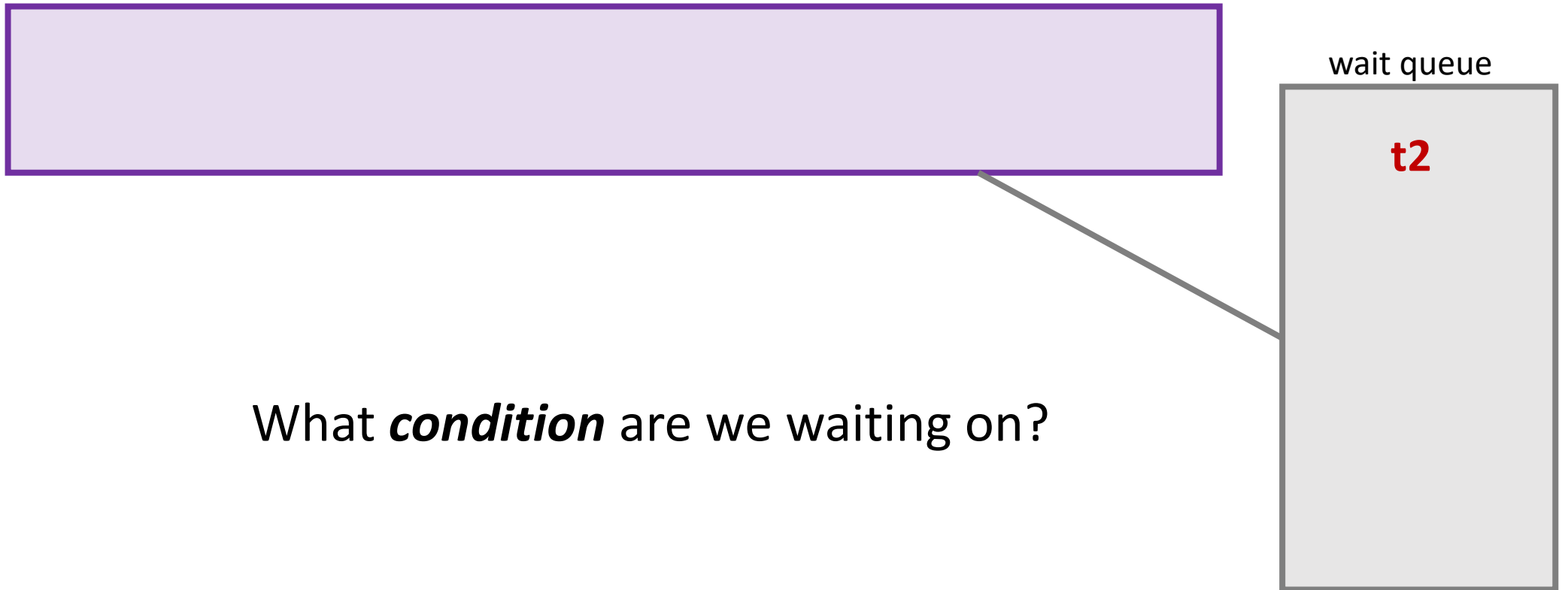


t2: take()

???

Signaling between threads

Bounded Buffer (producer-consumer queue)



What *condition* are we waiting on?

Signaling between threads

Bounded Buffer (producer-consumer queue)



Signaling between threads

Bounded Buffer (producer-consumer queue)



wait queue

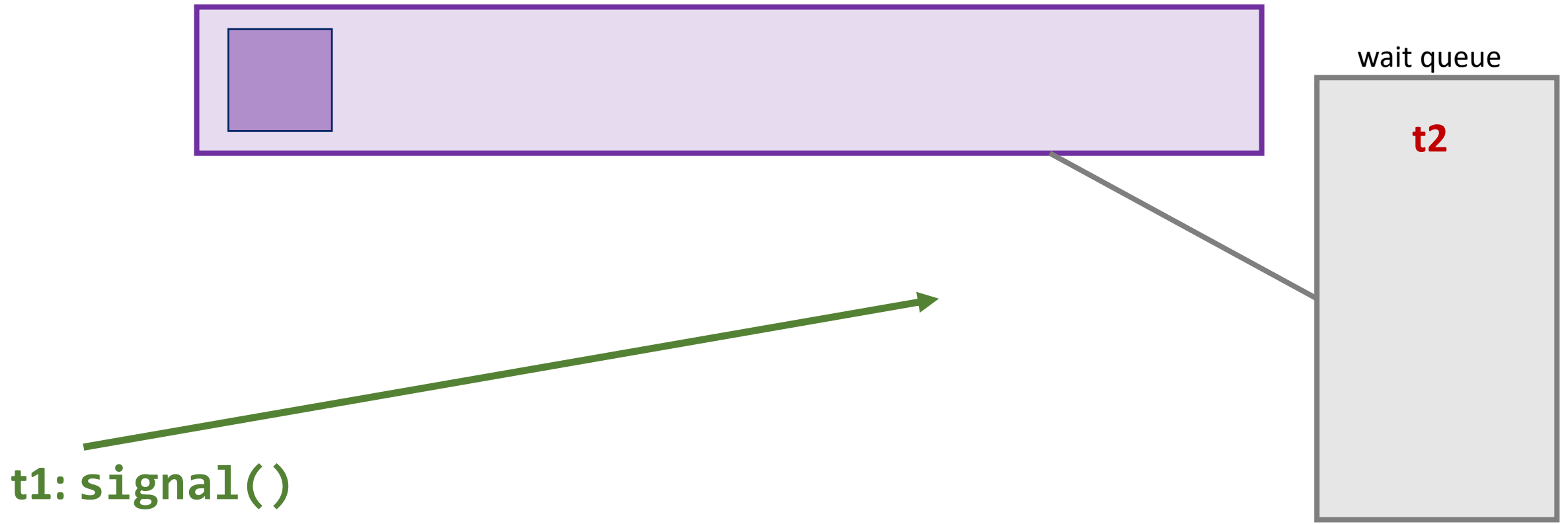
t2

What should happen?

t1: put()

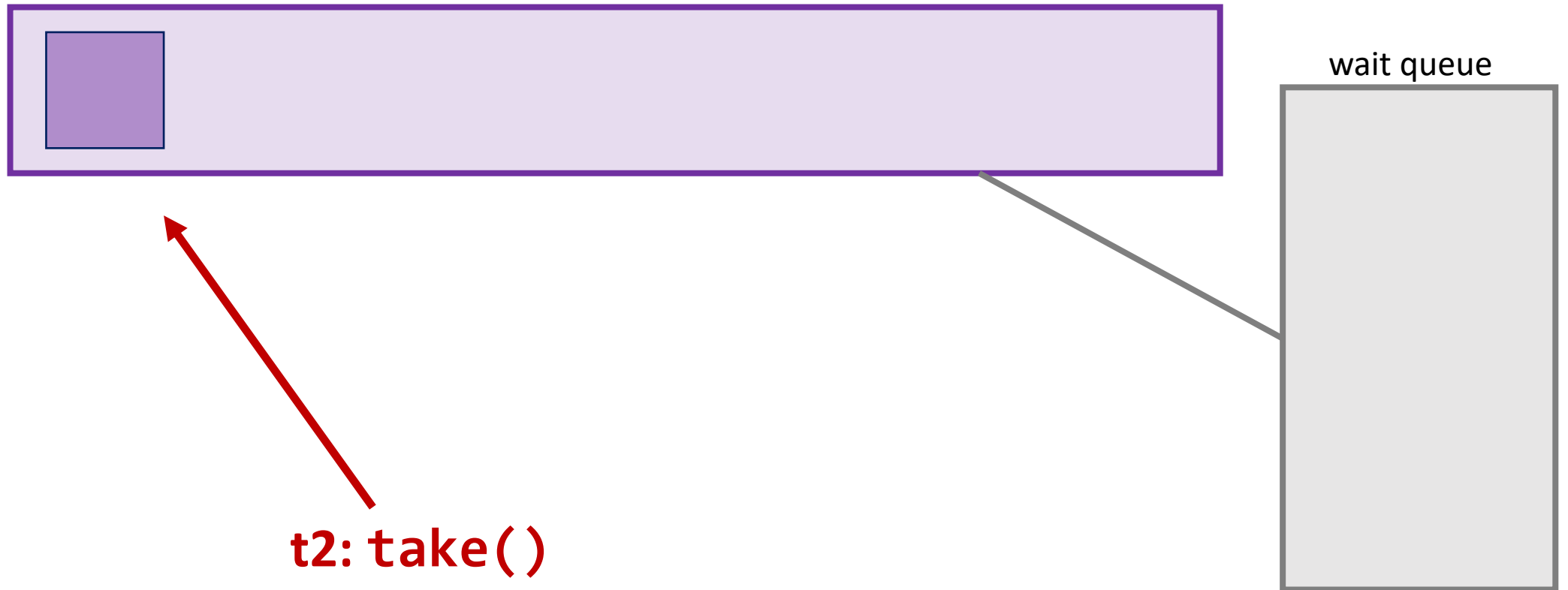
Signaling between threads

Bounded Buffer (producer-consumer queue)



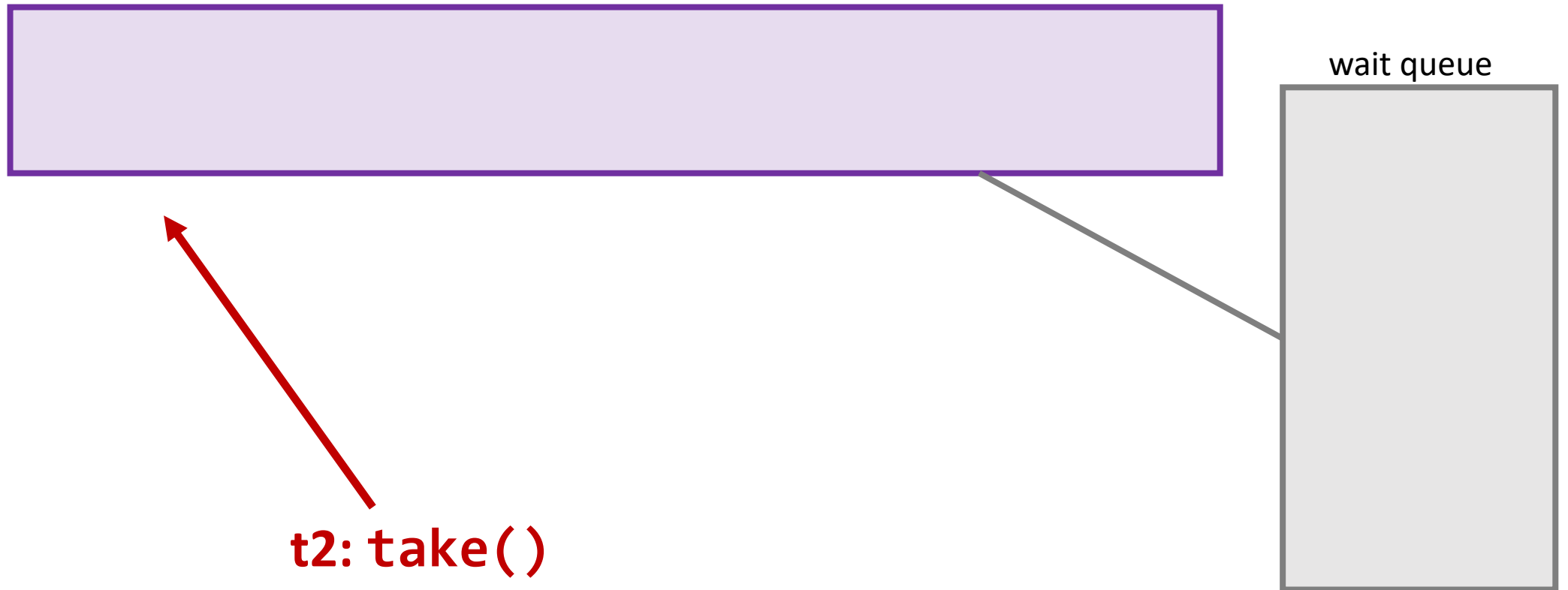
Signaling between threads

Bounded Buffer (producer-consumer queue)



Signaling between threads

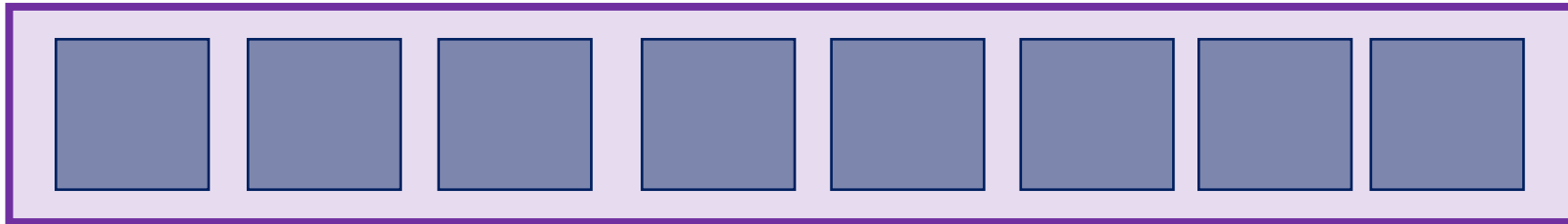
Bounded Buffer (producer-consumer queue)



The queue is a *circular queue*

- Also sometimes called a *ring buffer*

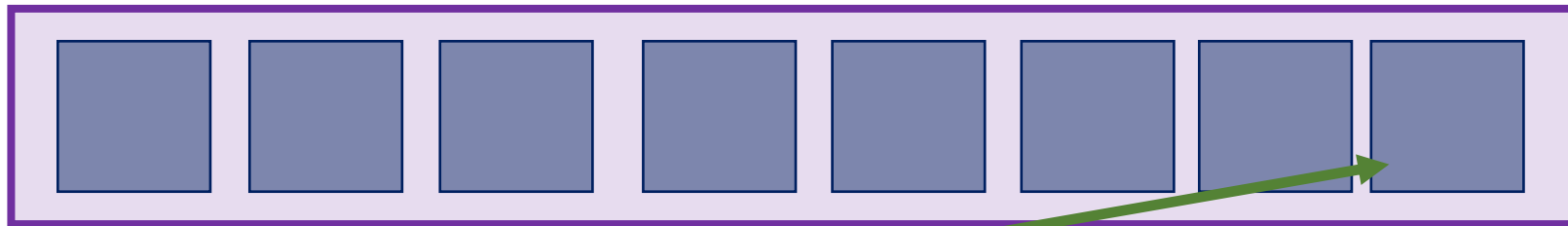
Bounded Buffer (producer-consumer queue)



The queue is a *circular queue*

- Also sometimes called a *ring buffer*

Bounded Buffer (producer-consumer queue)

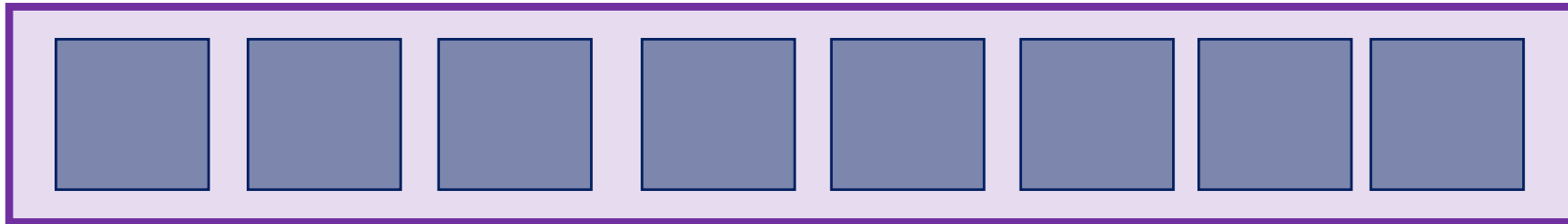


most recent put

The queue is a *circular queue*

- Also sometimes called a *ring buffer*

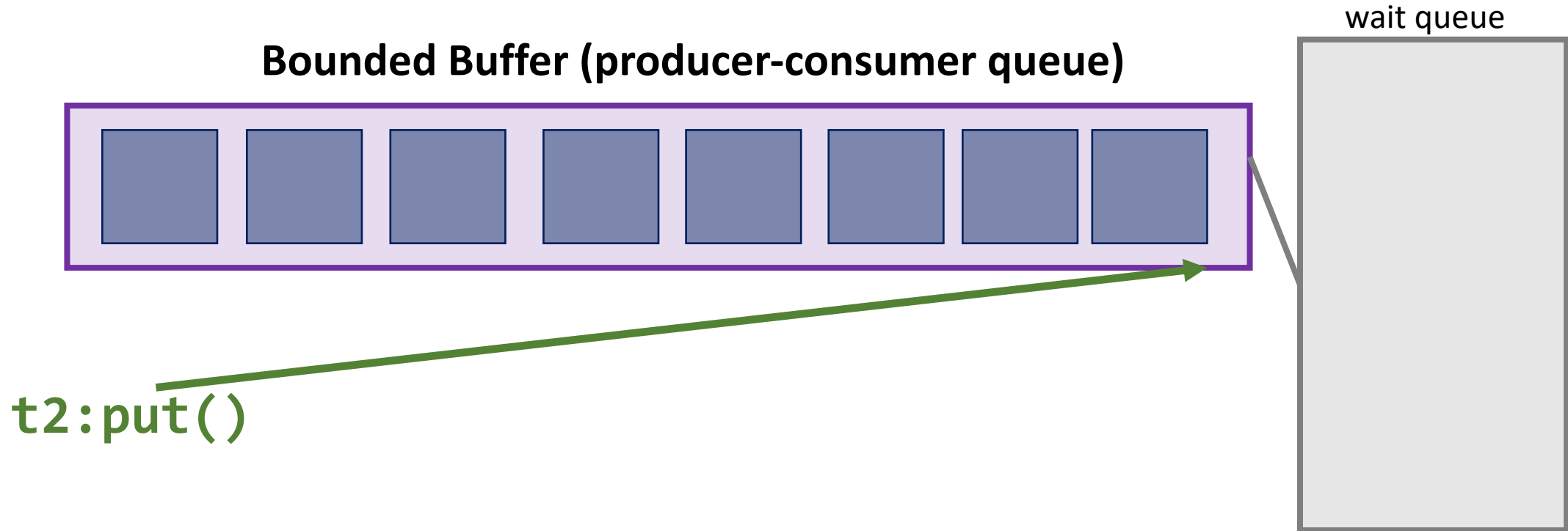
Bounded Buffer (producer-consumer queue)



We're full

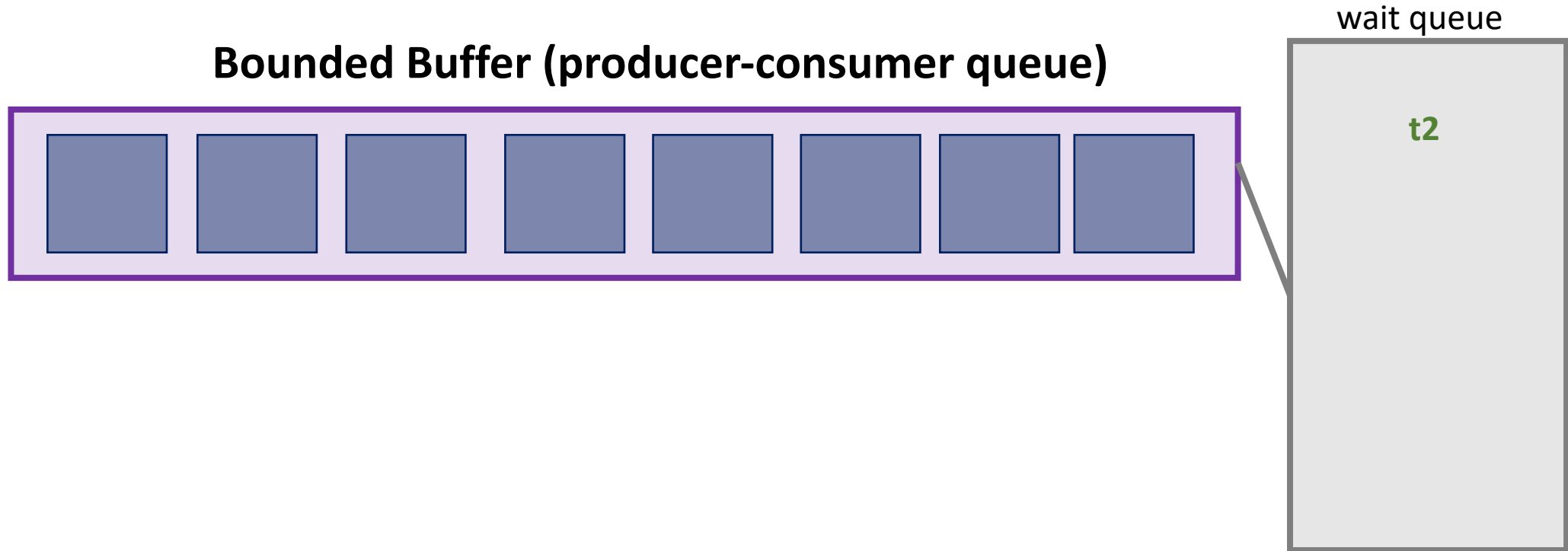
The queue is a *circular queue*

- Also sometimes called a *ring buffer*



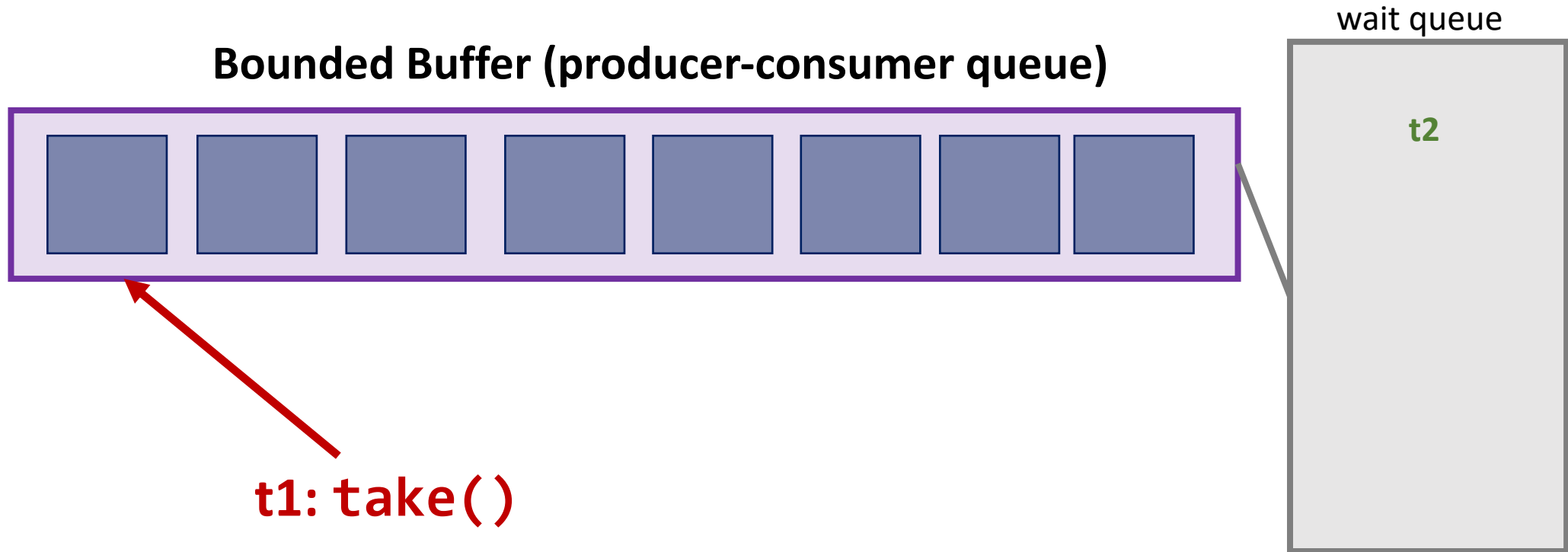
The queue is a *circular queue*

- Also sometimes called a *ring buffer*



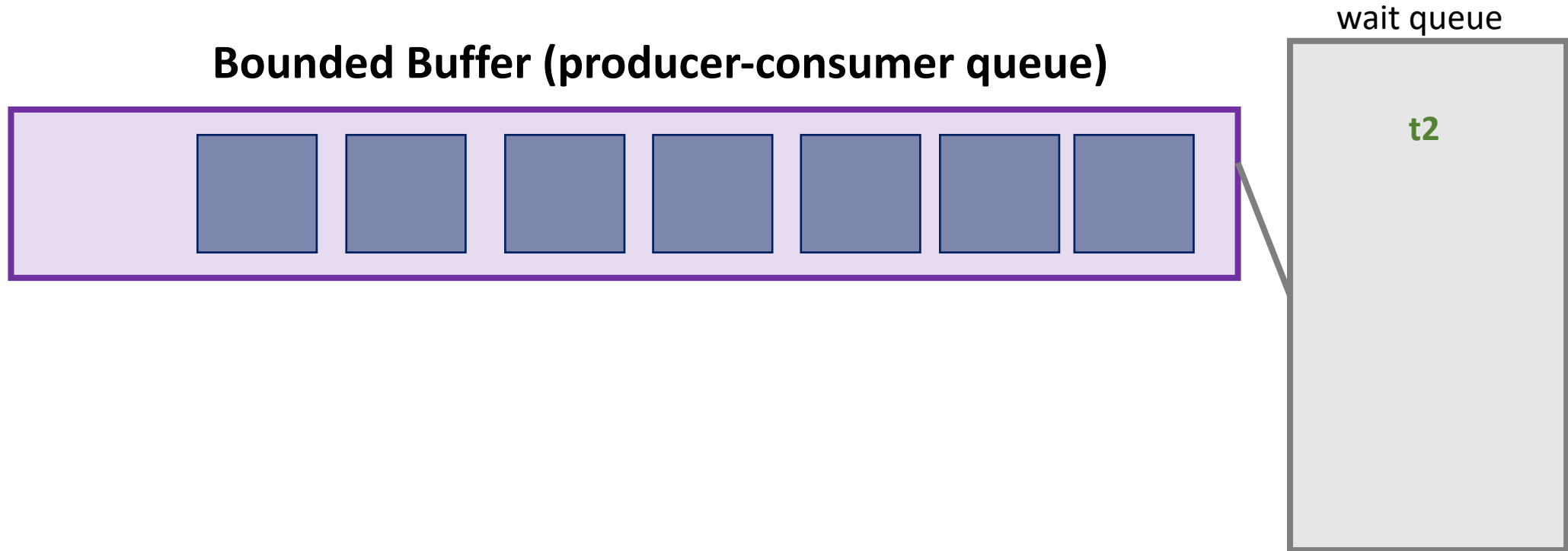
The queue is a *circular queue*

- Also sometimes called a *ring buffer*



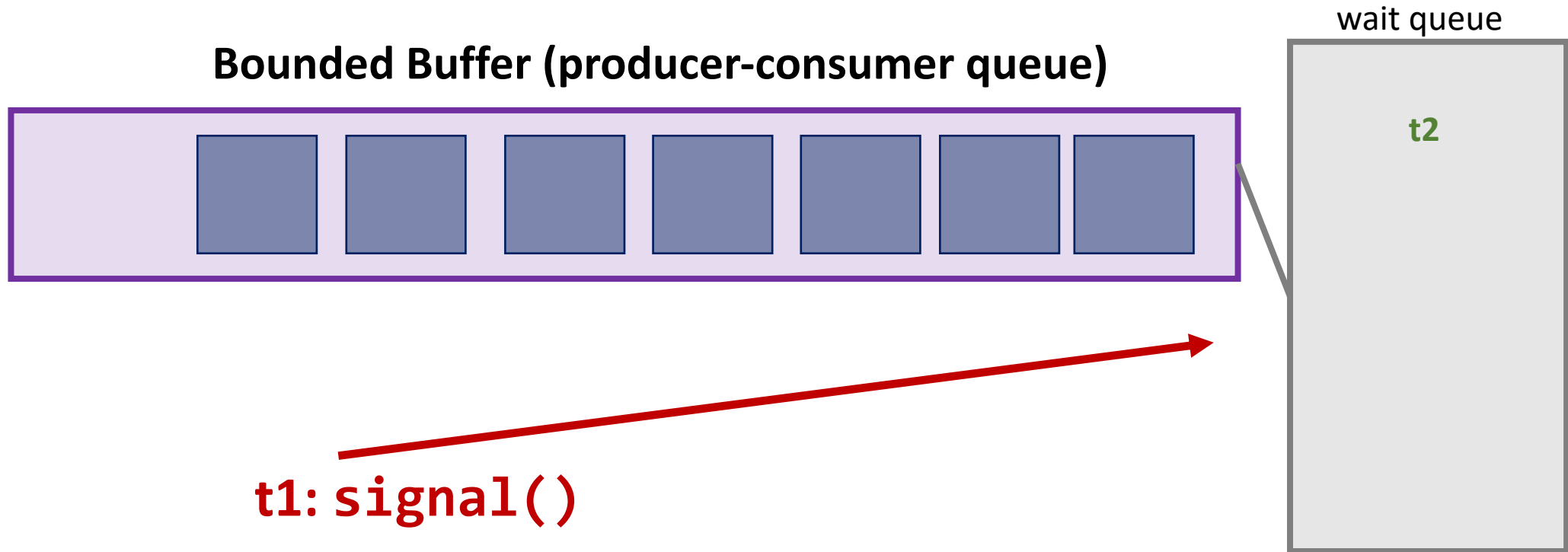
The queue is a *circular queue*

- Also sometimes called a *ring buffer*



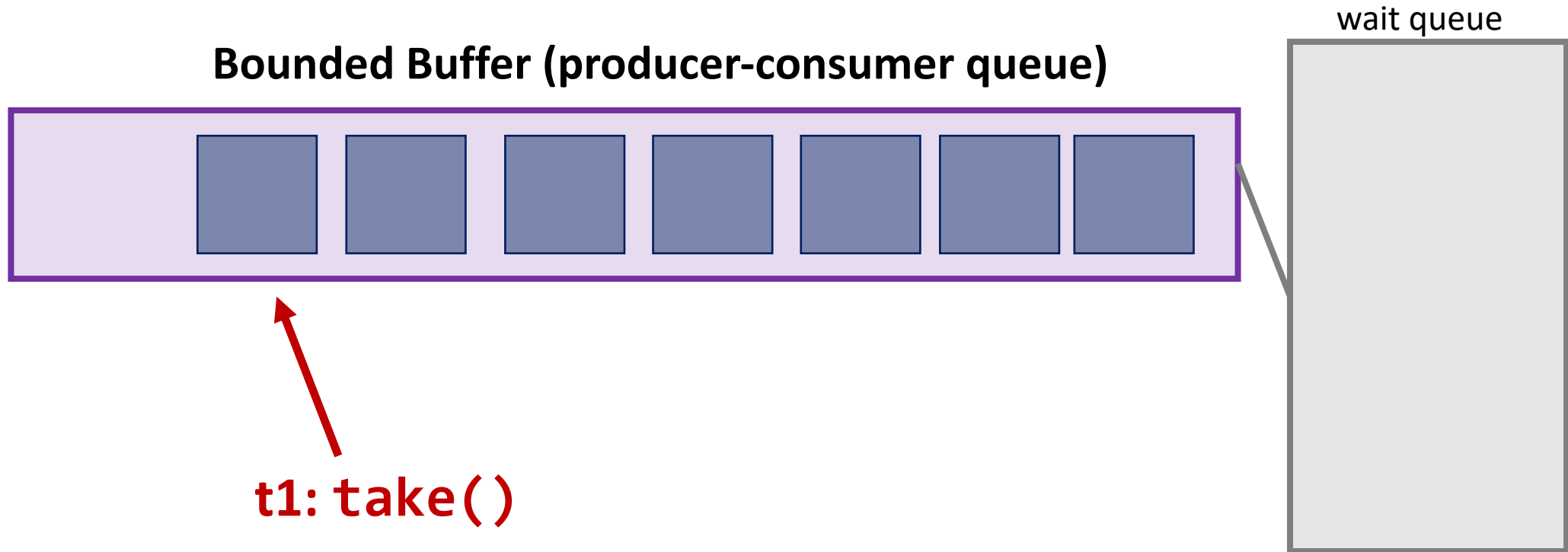
The queue is a *circular queue*

- Also sometimes called a *ring buffer*



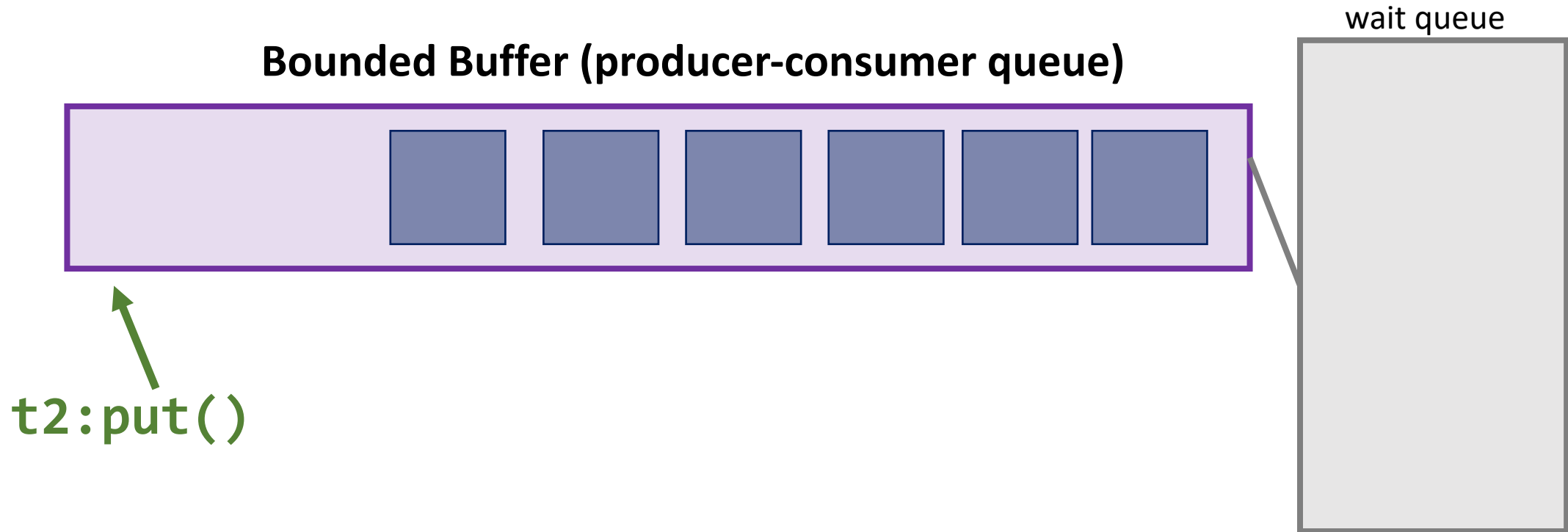
The queue is a *circular queue*

- Also sometimes called a *ring buffer*



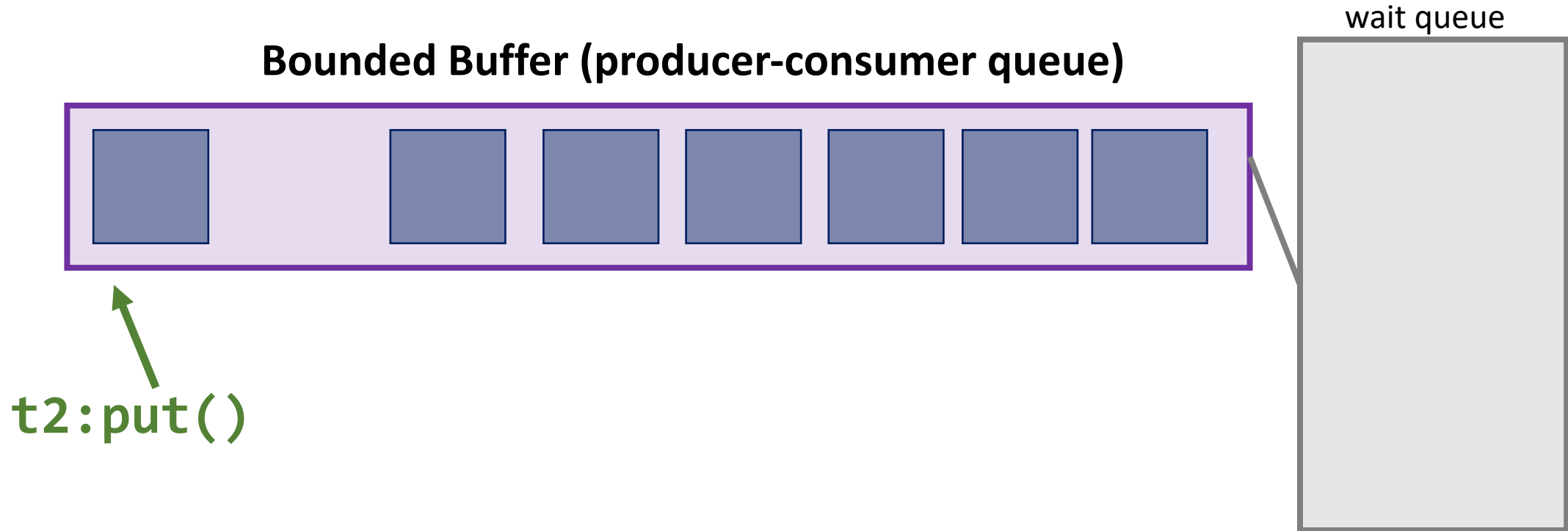
The queue is a *circular queue*

- Also sometimes called a *ring buffer*



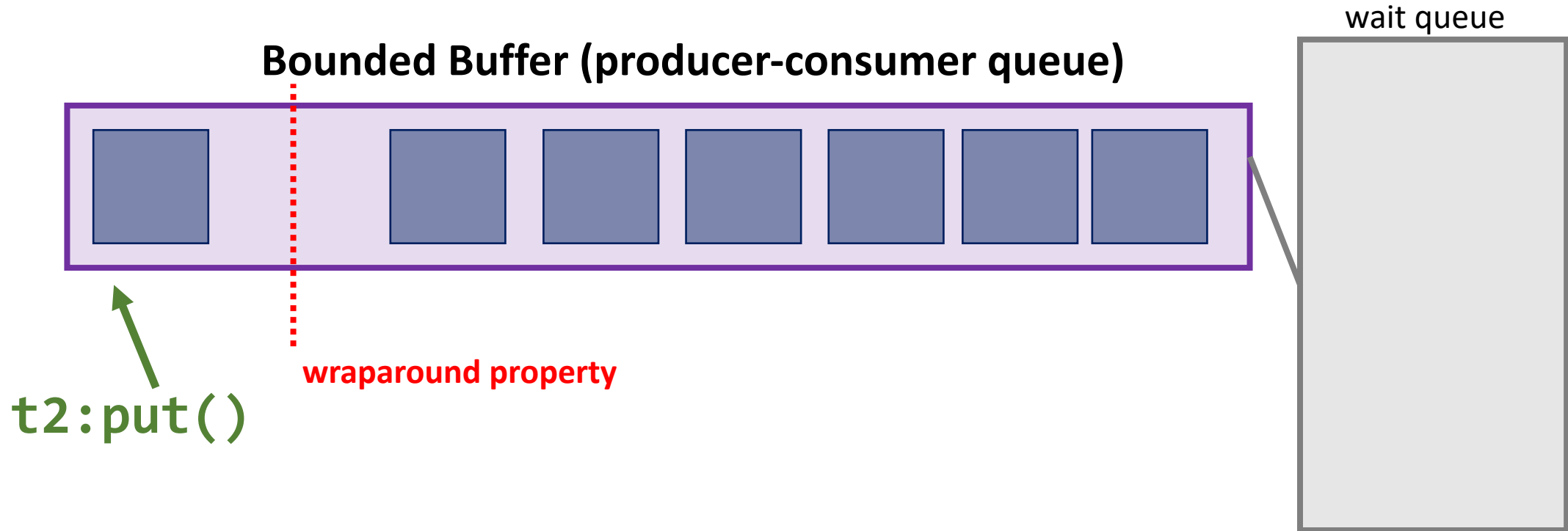
The queue is a *circular queue*

- Also sometimes called a *ring buffer*



The queue is a *circular queue*

- Also sometimes called a *ring buffer*



Bounded Buffer

- **Reads and writes** to buffer **require locking**
- when buffers are **full**, **writers** wait()
- when buffers are **empty**, **readers** wait()
- Many programming languages expose this abstraction directly
 - **chan** in Go
 - **channel** in Julia
 - Python requires a **queue()** with a **lock()**
 - Rust might use **CircularBuffer()** and **CondVar()** maybe **crossbeam_channel()**

Bounded Buffer

- **Producers** generate data
- **Consumers** take data and process it
- *Very frequent* situation encountered in systems programming
- General strategy: use CVs to notify:
 - waiting readers when data is available
 - waiting writers when slots are free

Example

- Easy case:
 - 1 consumer thread
 - 1 producer thread
 - 1 shared buffer (max slots=1)

- `numfill` is buffer slots currently filled
- let's start with some code that doesn't *quite* work

numfull = 0

[RUNNABLE]

```
void *producer (void *arg) {  
    → for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNING]

```
void *consumer (void *arg) {  
    while (1) {  
        → mutex_lock(&m);  
        while (numfull == 0)  
            cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 0

[RUNNABLE]

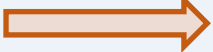
```
void *producer (void *arg) {  
    → for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        → while (numfull == 0)  
            cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 0

[RUNNABLE]

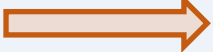
```
void *producer (void *arg) {  
     for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
             cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 0

[RUNNABLE]

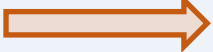
```
void *producer (void *arg) {  
     for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[SLEEPING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
             cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 0

[RUNNING]

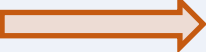
```
void *producer (void *arg) {  
     for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[SLEEPING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
             cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 0

[RUNNING]

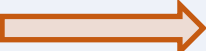
```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
         mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[SLEEPING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
             cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 0

[RUNNING]


```
void *producer (void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m);
         while (numfull == max)
            cond_wait(&cond, &m);
        do_fill(i);
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

[SLEEPING]


```
void *consumer (void *arg) {
    while (1) {
        mutex_lock(&m);
        while (numfull == 0)
             cond_wait(&cond, &m);
        int tmp = do_get();
        cond_signal(&cond);
        mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```


numfull = 0

[RUNNING]

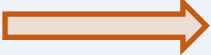
```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
         do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[SLEEPING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
            cond_wait(&cond, &m);  
         int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 1

[RUNNING]

```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
         cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

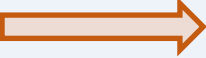
[SLEEPING]

```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
             cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 1


[RUNNING]

```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```




[RUNNABLE]

```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
            cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```




numfull = 1

[RUNNING]

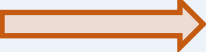
```
void *producer (void *arg) {  
     for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNABLE]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
             cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 1

[RUNNING]

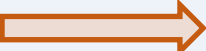
```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
         mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNABLE]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
             cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 1

[RUNNING]

```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
         while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNABLE]

```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
            cond_wait(&cond, &m);  
         int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 1

[RUNNING]

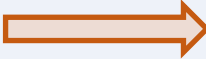
```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            → cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNABLE]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
            → cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 1

[SLEEPING]

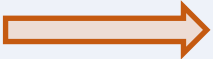
```
void *producer (void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m);
        while (numfull == max)
             cond_wait(&cond, &m);
        do_fill(i);
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

[RUNNABLE]


```
void *consumer (void *arg) {
    while (1) {
        mutex_lock(&m);
        while (numfull == 0)
             cond_wait(&cond, &m);
        int tmp = do_get();
        cond_signal(&cond);
        mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```


numfull = 1

[SLEEPING]

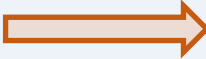
```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
             cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
             cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 1

[SLEEPING]

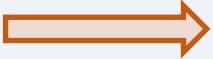
```
void *producer (void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m);
        while (numfull == max)
             cond_wait(&cond, &m);
        do_fill(i);
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

[RUNNING]


```
void *consumer (void *arg) {
    while (1) {
        mutex_lock(&m);
         while (numfull == 0)
            cond_wait(&cond, &m);
        int tmp = do_get();
        cond_signal(&cond);
        mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull = 1

[SLEEPING]

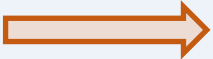
```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
             cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
            cond_wait(&cond, &m);  
         int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 0

[SLEEPING]

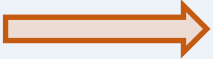
```
void *producer (void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m);
        while (numfull == max)
             cond_wait(&cond, &m);
        do_fill(i);
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

[RUNNING]


```
void *consumer (void *arg) {
    while (1) {
        mutex_lock(&m);
        while (numfull == 0)
            cond_wait(&cond, &m);
         int tmp = do_get();
        cond_signal(&cond);
        mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull = 0

[RUNNABLE]

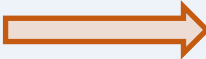
```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
             cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
            cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
         mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 0

[RUNNABLE]

```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
             cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNING]

```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
            cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
         printf("%d\n", tmp);  
    }  
}
```

numfull = 0

[RUNNABLE]

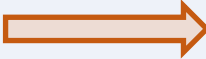
```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
            cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 0

[RUNNABLE]

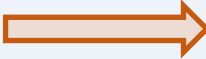
```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
             cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[RUNNING]


```
void *consumer (void *arg) {  
    while (1) {  
         mutex_lock(&m);  
        while (numfull == 0)  
            cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```


numfull = 0

[RUNNABLE]

```
void *producer (void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m);
        while (numfull == max)
             cond_wait(&cond, &m);
        do_fill(i);
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

[RUNNING]

```
void *consumer (void *arg) {
    while (1) {
        mutex_lock(&m);
         while (numfull == 0)
            cond_wait(&cond, &m);
        int tmp = do_get();
        cond_signal(&cond);
        mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull = 0

[RUNNABLE]

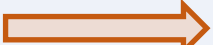
```
void *producer (void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m);
        while (numfull == max)
            → cond_wait(&cond, &m);
        do_fill(i);
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

[RUNNING]


```
void *consumer (void *arg) {
    while (1) {
        mutex_lock(&m);
        while (numfull == 0)
            → cond_wait(&cond, &m);
        int tmp = do_get();
        cond_signal(&cond);
        mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull = 0

[RUNNING]

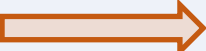
```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
             cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[SLEEPING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
             cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 0

[RUNNING]

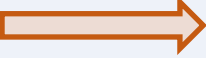
```
void *producer (void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m);
         while (numfull == max)
            cond_wait(&cond, &m);
        do_fill(i);
        cond_signal(&cond);
        mutex_unlock(&m);
    }
}
```

[SLEEPING]


```
void *consumer (void *arg) {
    while (1) {
        mutex_lock(&m);
        while (numfull == 0)
             cond_wait(&cond, &m);
        int tmp = do_get();
        cond_signal(&cond);
        mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

numfull = 0

[RUNNING]

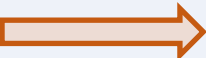
```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
         do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

[SLEEPING]


```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
             cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 1

[RUNNING]

```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
         cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```

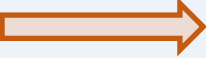
[SLEEPING]

```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
             cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

numfull = 1


[RUNNING]

```
void *producer (void *arg) {  
    for (int i = 0; i < loops; i++) {  
        mutex_lock(&m);  
        while (numfull == max)  
            cond_wait(&cond, &m);  
        do_fill(i);  
        cond_signal(&cond);  
        mutex_unlock(&m);  
    }  
}
```



[RUNNABLE]

```
void *consumer (void *arg) {  
    while (1) {  
        mutex_lock(&m);  
        while (numfull == 0)  
            cond_wait(&cond, &m);  
        int tmp = do_get();  
        cond_signal(&cond);  
        mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```



What about two consumers?

- Can you find a *bad schedule*?

[RUNNING]

[RUNNABLE]

```

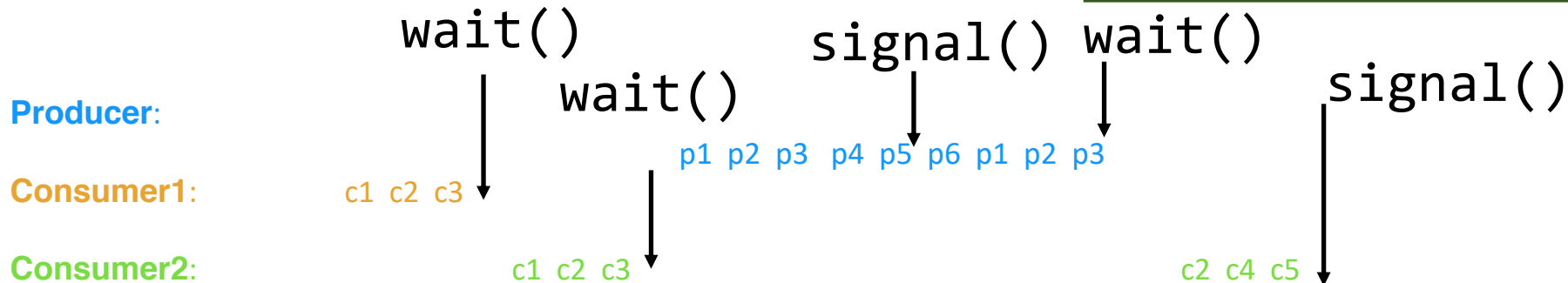
void *producer (void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&m); // p1
        while (numfull == max) //p2
            cond_wait(&cond, &m); //p3
        do_fill(i); // p4
        cond_signal(&cond); // p5
        mutex_unlock(&m); // p6
    }
}

```

```

void *consumer (void *arg) {
    while (1) {
        mutex_lock(&m); // c1
        while (numfull == 0) // c2
            cond_wait(&cond, &m);
        int tmp = do_get(); // c4
        cond_signal(&cond); // c5
        mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}

```



does last signal wake **producer** or **consumer1**?

How to wake the right thread?

- One solution:
 - wake all threads!

API

- **wait**(cond_t * cv, mutex_t * lock)
 - assumes lock is held when wait() is called (*why?*)
 - puts caller to sleep + releases the lock (atomically)
 - when awoken, reacquires lock before returning
- **signal**(cond_t * cv)
 - wake a *single* waiting thread (if ≥ 1 thread is waiting)
 - if there is no waiting thread, NOP
- **broadcast**(cond_t * cv)
 - wake *all* waiters
 - if no waiting threads, NOP

Summary

- **Rules of thumb**

- Keep state in addition to CVs
- Always `wait()` or `signal()` *with lock held*
- Recheck state assumptions when waking up from waiting