ILLINOIS TECH

College of Computing

CS 450 Operating Systems
Condition Variables

Yue Duan

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**

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;
```

join(): parent must wait for child thread to finish

How to implement?

Waiting for an Event

- Parent thread has to wait until child terminates
- **Option 1**: spin until that happens
  - Waste of CPU time
- **Option 2**: wait (sleep) in a queue until that happens
  - Better use of CPU time
  - Similar to the idea in queue-based lock in the previous lecture
  - Child thread will signal the parent to wake up before its termination

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

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

Thread Join: Attempt 1

Parent

```
void thr_join() {
  cond_wait(&c);
}
```

Child

```
void thr_exit() {
  cond_signal(&c);
}
```

- Does this work? If not, what's the problem?
- Child may run and call **cond_signal**() before parent called **cond_wait**()
    - Parent will sleep indefinitely.

Thread Join: Attempt 2

Parent

```
void thr_join() {
   if (done == 0) {          //a
      cond_wait(&c);         //b
   }
}
```

Child

```
void thr_exit() {
   done = 1;                 //x
   cond_signal(&c);          //y
}
```

- Let's keep some **state** then
- Is there a problem here?

Parent:	a			b
Child:		x	y	

Using Locks to Achieve Atomicity

Waiting Thread

```
mutex_lock(&m);
if (! check_cond())
    cond_wait(&c, &m);
...
mutex_unlock(&m);
```

Waking Thread

```
mutex_lock(&m);
set_cond();
cond_signal(&c);
...
mutex_unlock(&m);
```

- Need a lock (mutex) to ensure two things
  - Checking condition (waiting thread) & modifying it (waking thread) remain mutually exclusive
  - Checking condition & putting thread to sleep (waiting thread) remain atomic

Using Locks to Achieve Atomicity

Waiting Thread

```
mutex_lock(&m);
if (! check_cond())
    cond_wait(&c, &m);
…
mutex_unlock(&m);
```

Waking Thread

```
mutex_lock(&m);
set_cond();
cond_signal(&c);
…
mutex_unlock(&m);
```

- cond_wait() should unlock mutex atomically w/ going to sleep
  - If mutex not released, waking thread cannot make progress
  - If release is not atomic, we get a race condition. Can you identify it?

Recap: CV Rules of Thumb (Take 1)

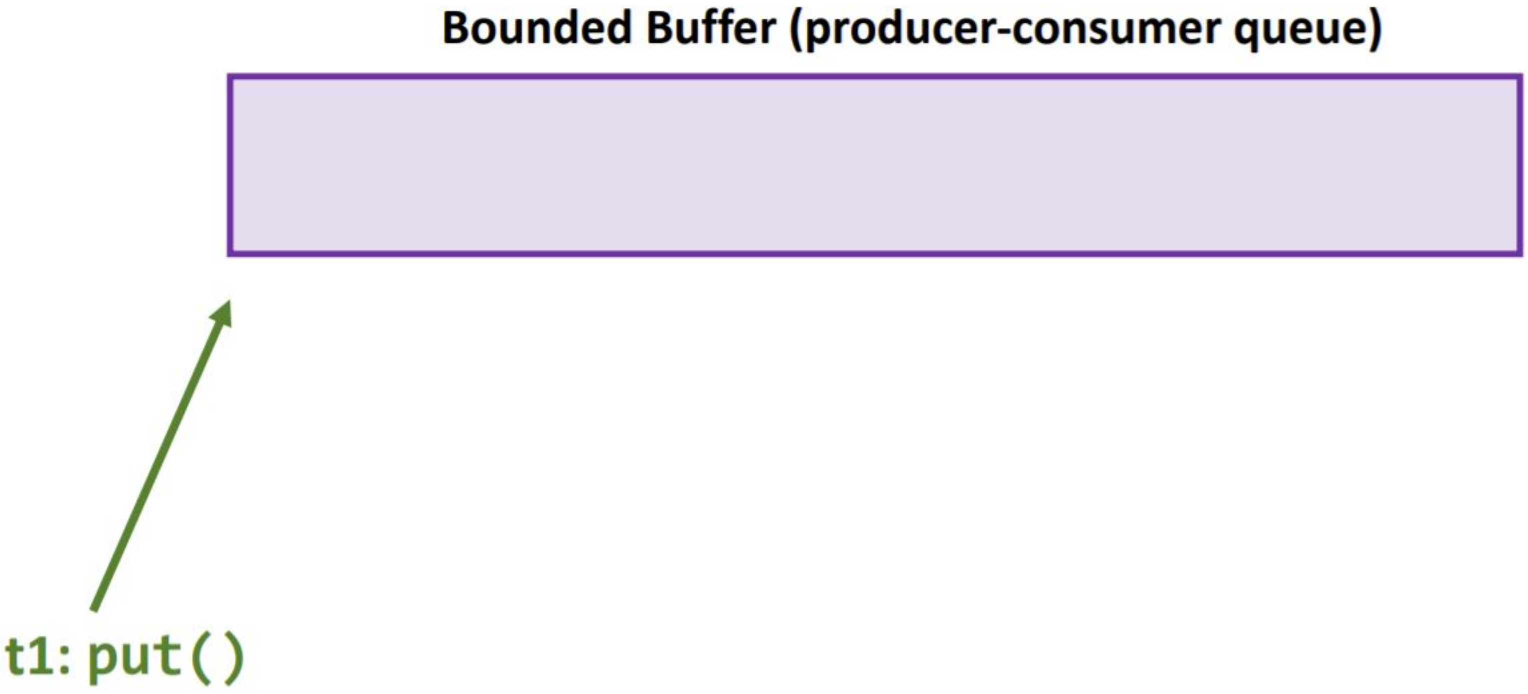
- Shared state determines if condition is true or not
- Check the state before waiting on cv
- Use a mutex to protect
  - 1) the shared state on which condition is based, as well as
  - 2) operations on the cv
- Remember to acquire the mutex before calling **cond_signal**()

Bounded Buffer (Producer/Consumer)

- Multiple producers and multiple consumers communicate using a shared, finite-size buffer
- Producers add items to buffer
  - If buffer is full, producer has to wait until there is free space
- Consumers remove items from buffer
  - If buffer is empty, consumer has to wait until one or more items are added
- Common examples:
  - Unix pipe: bounded buffer in kernel (multiple producers & consumers)
  - Work queue in a web server (one producer, multiple consumers)

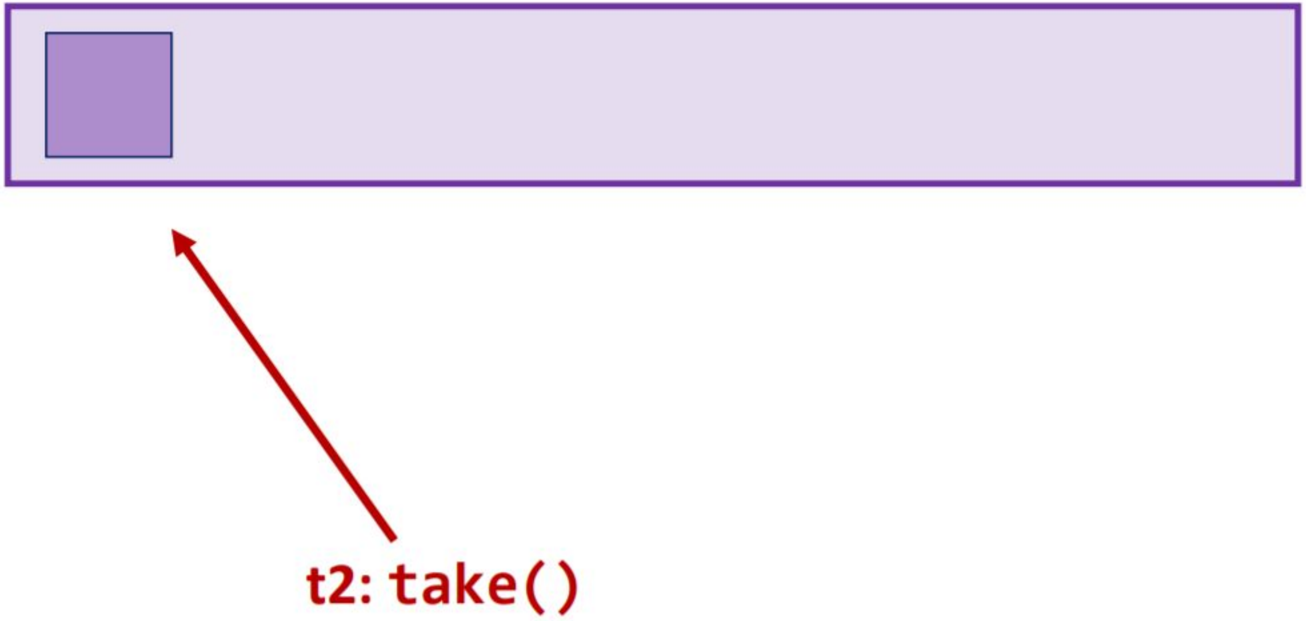
Bounded Buffer (Producer/Consumer)

Bounded Buffer (Producer/Consumer)

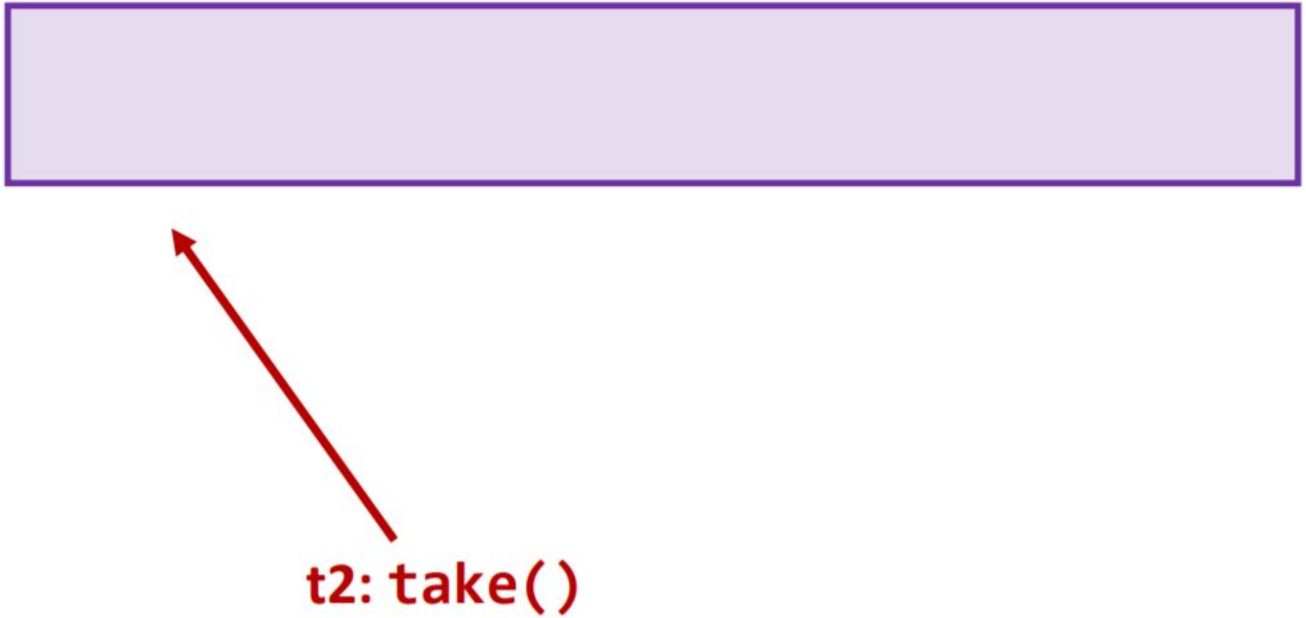


Bounded Buffer (Producer/Consumer)

Bounded Buffer (Producer/Consumer)

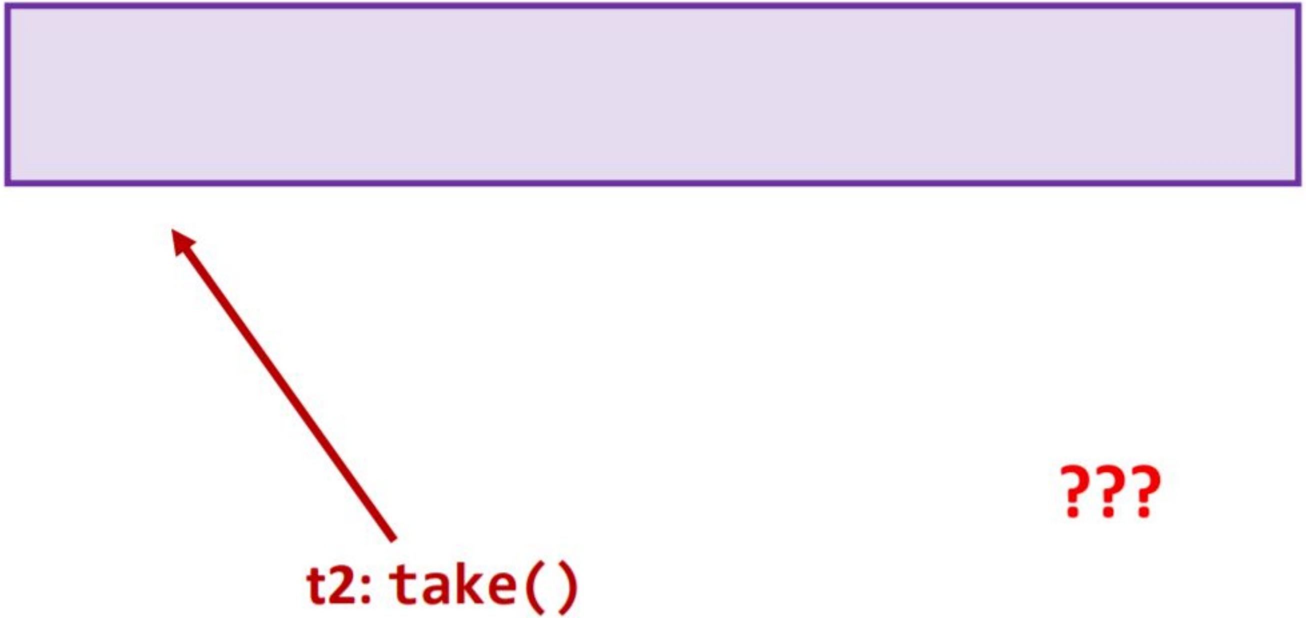


Bounded Buffer (Producer/Consumer)

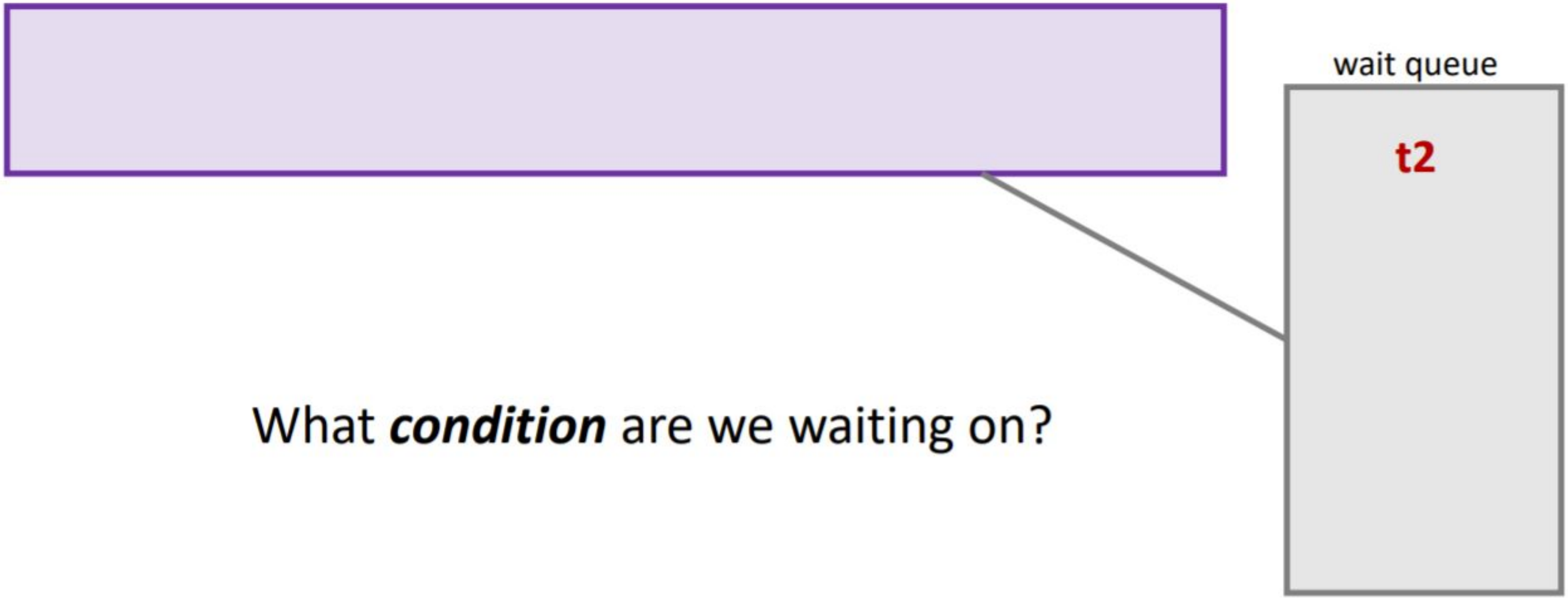


Bounded Buffer (Producer/Consumer)

Bounded Buffer (Producer/Consumer)



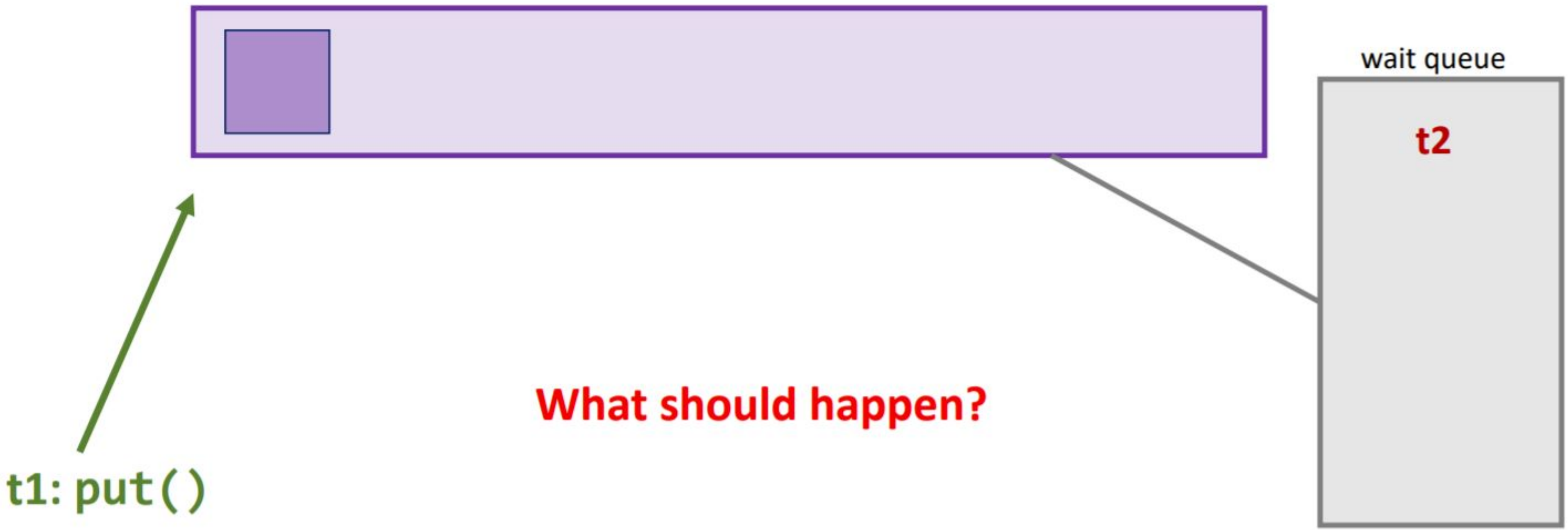
Bounded Buffer (Producer/Consumer)



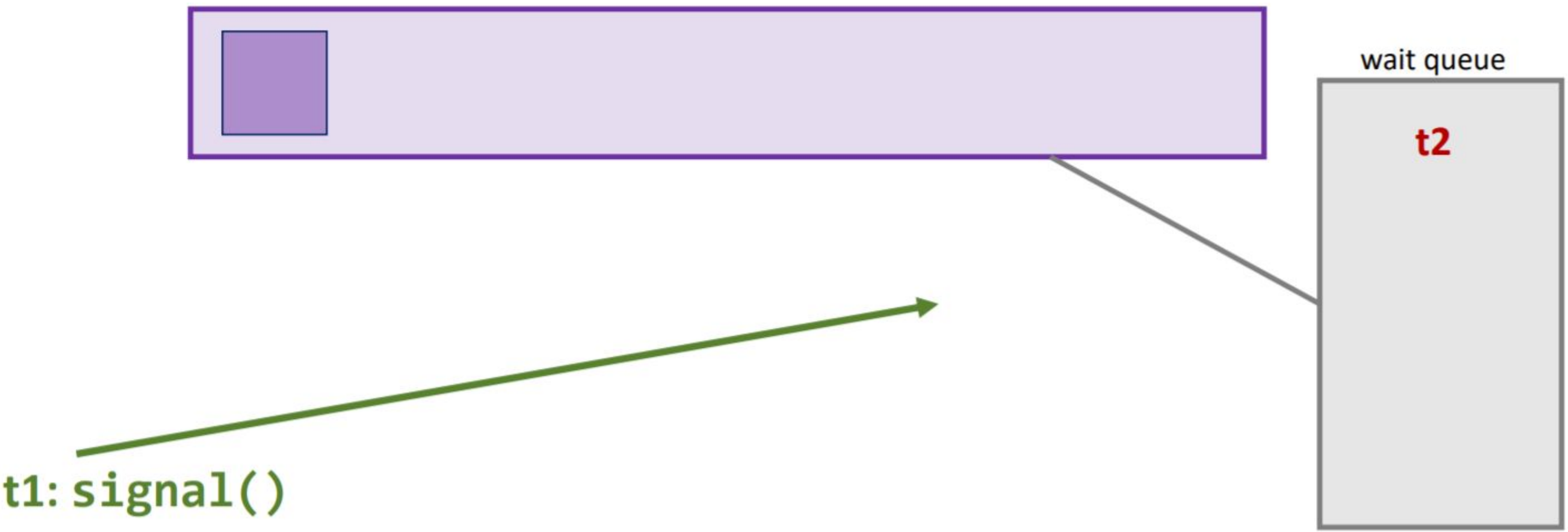
What ***condition*** are we waiting on?

Bounded Buffer (Producer/Consumer)

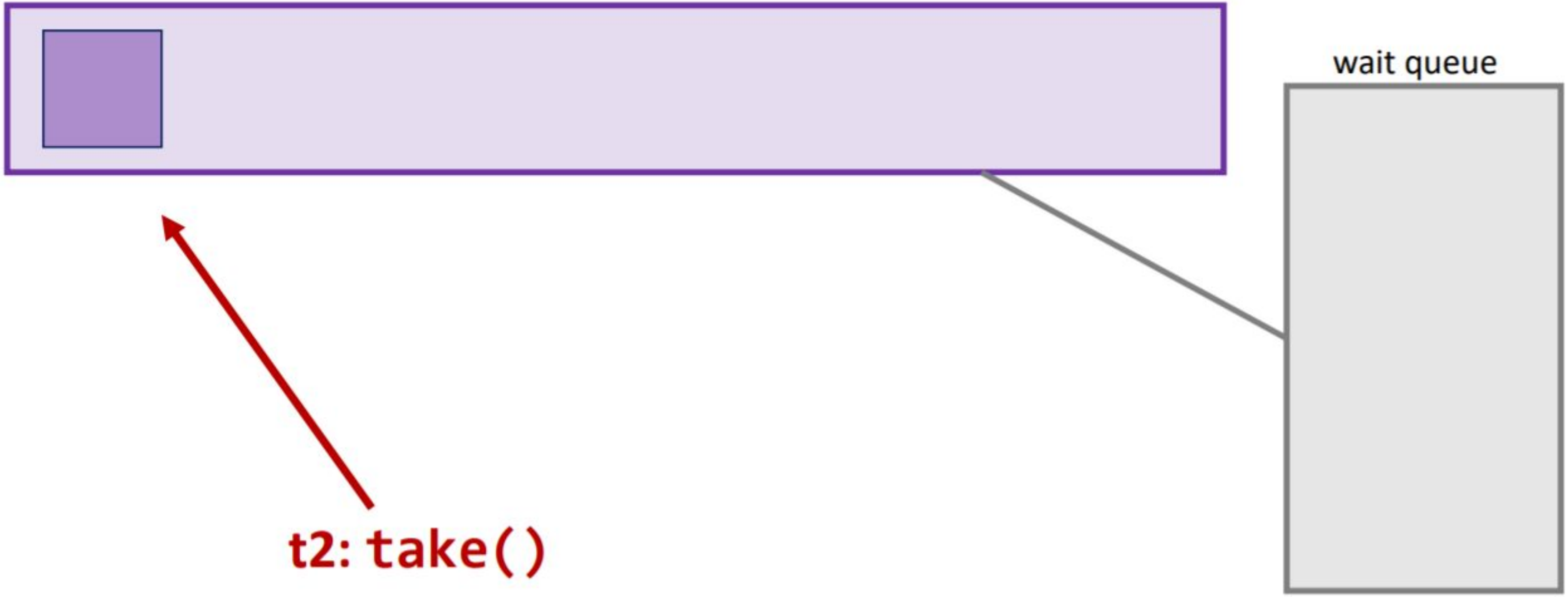
Bounded Buffer (Producer/Consumer)



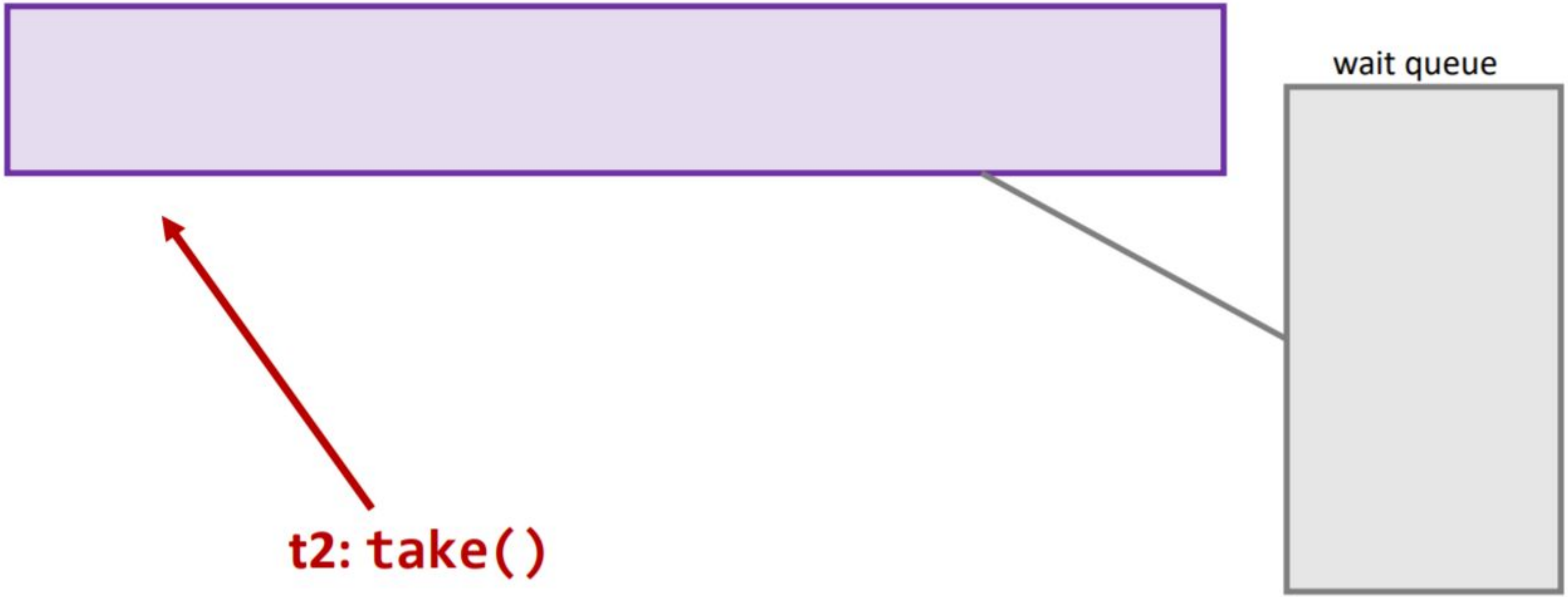
Bounded Buffer (Producer/Consumer)



Bounded Buffer (Producer/Consumer)



Bounded Buffer (Producer/Consumer)

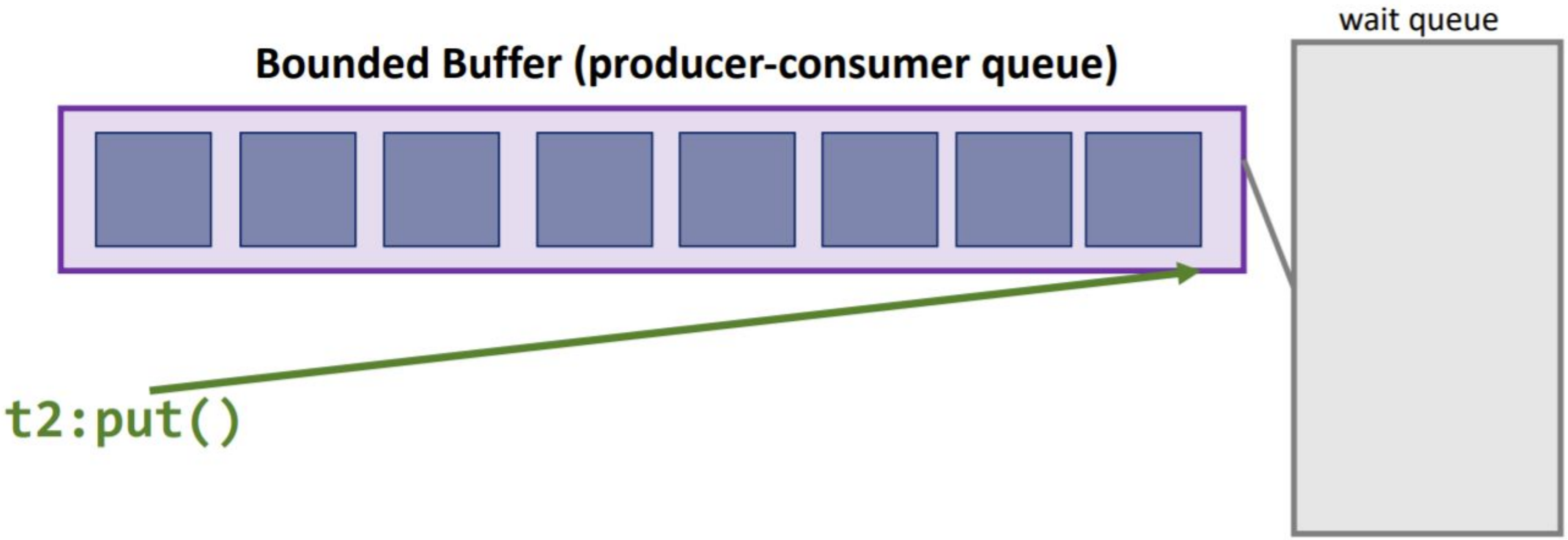


Bounded Buffer (Producer/Consumer)

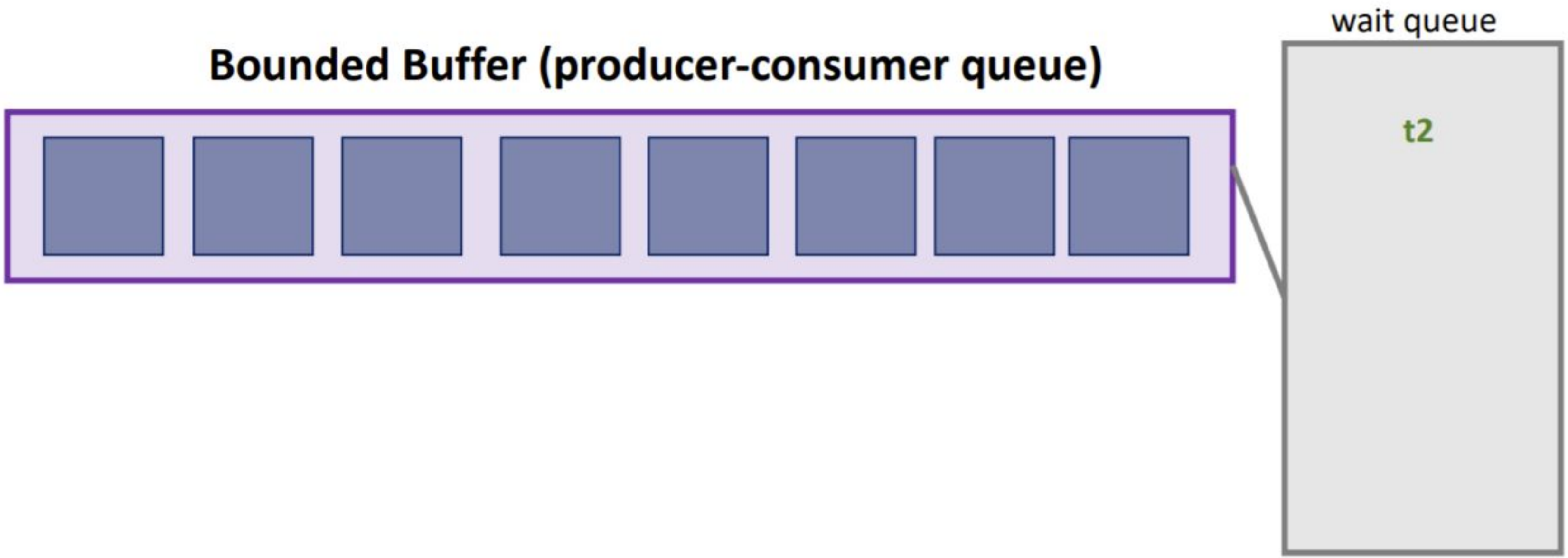
- The queue is a **circular** queue
- Sometimes also called a **ring buffer**

We're full

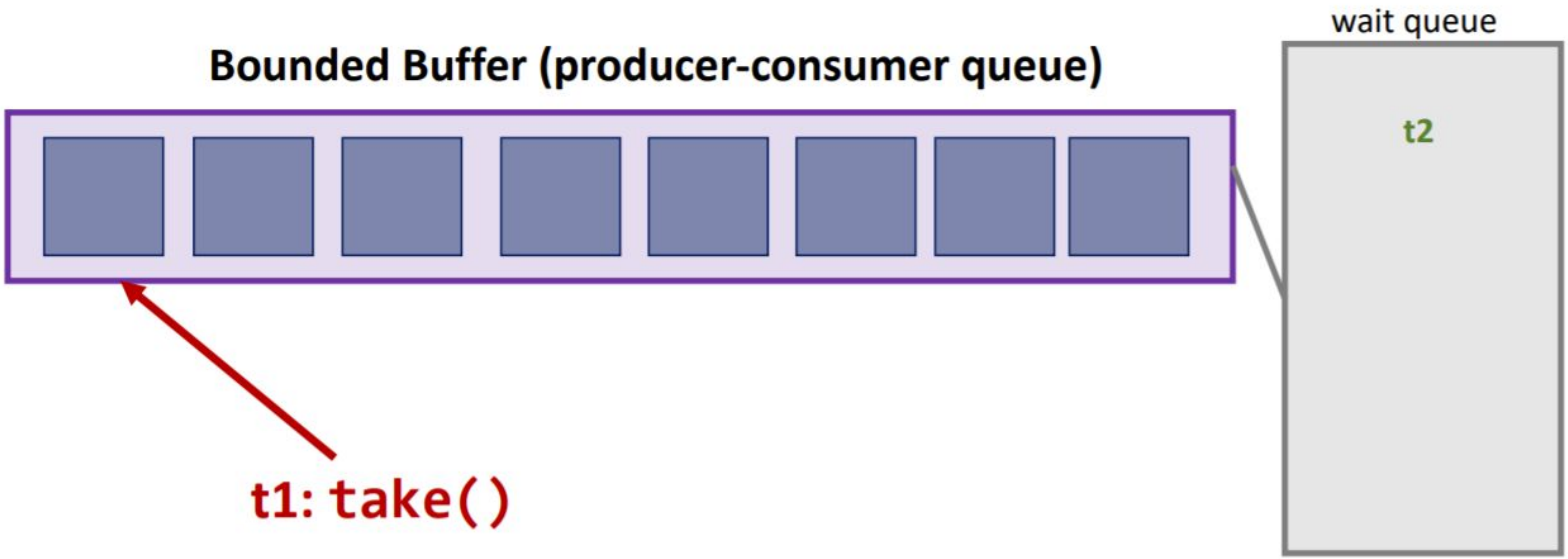
Bounded Buffer (Producer/Consumer)



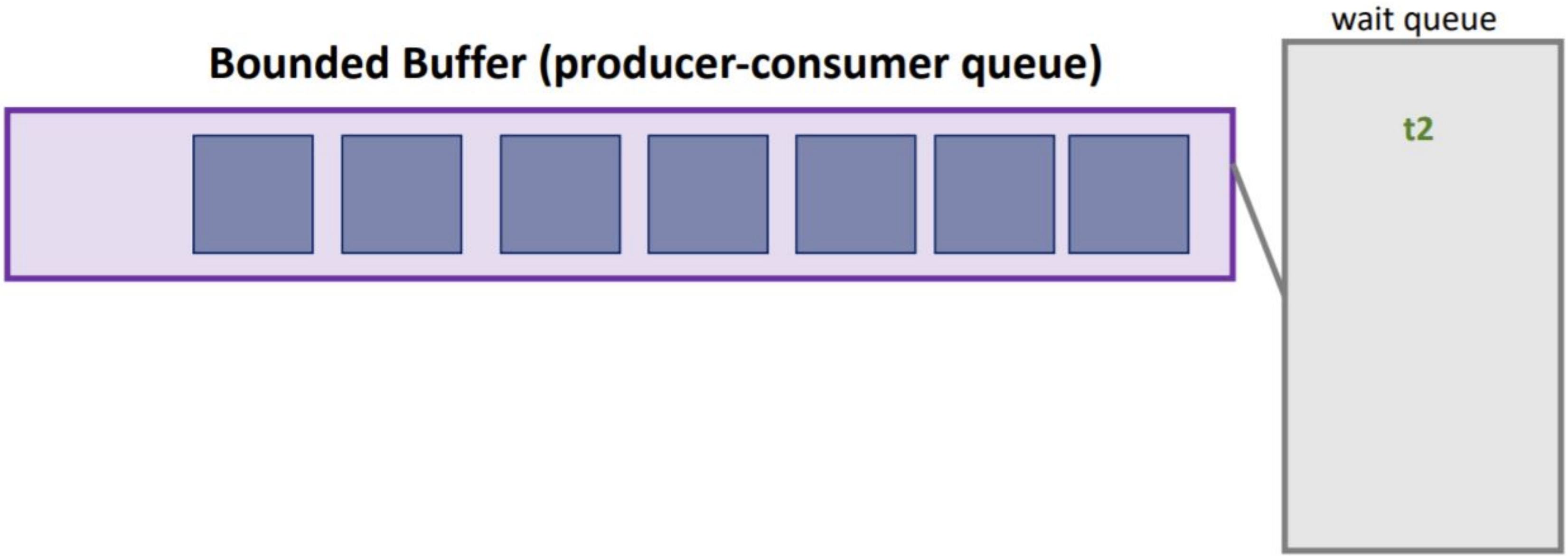
Bounded Buffer (Producer/Consumer)



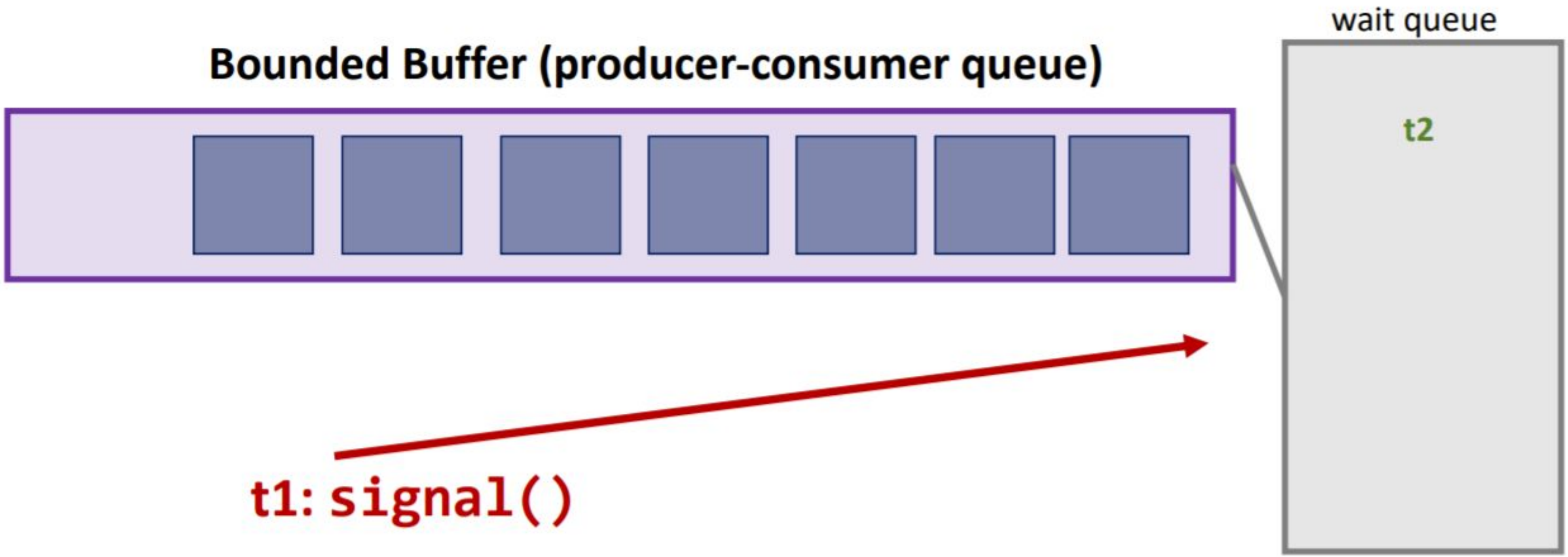
Bounded Buffer (Producer/Consumer)



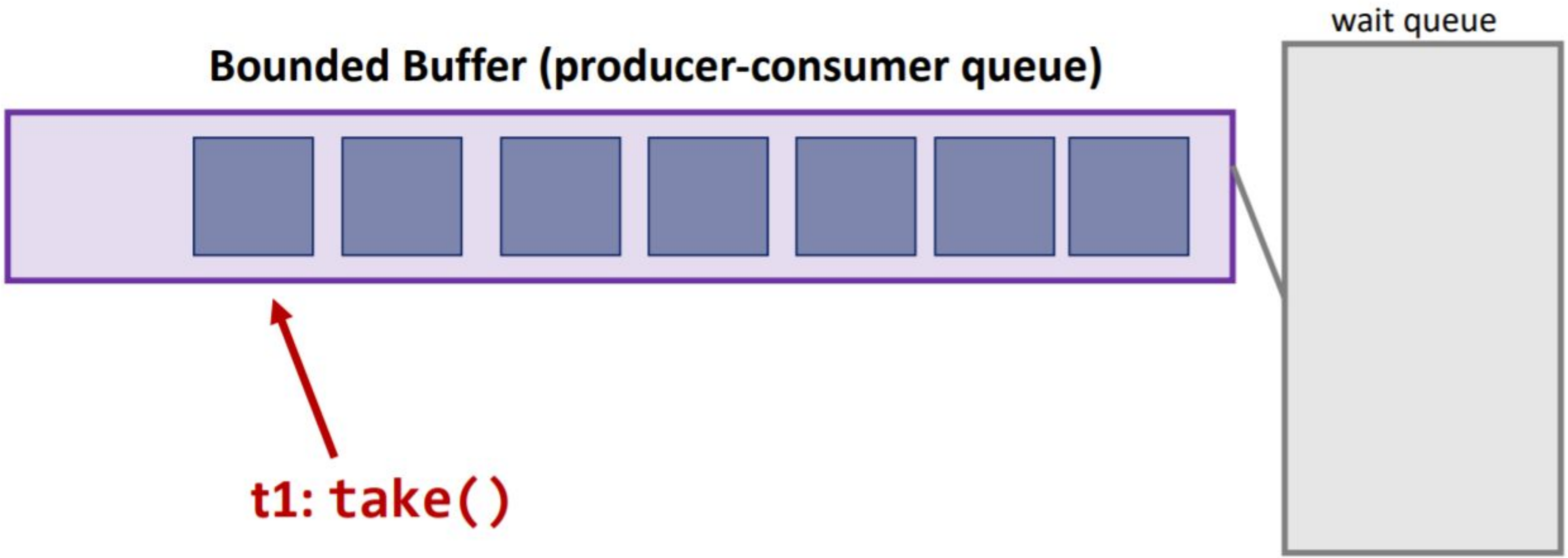
Bounded Buffer (Producer/Consumer)



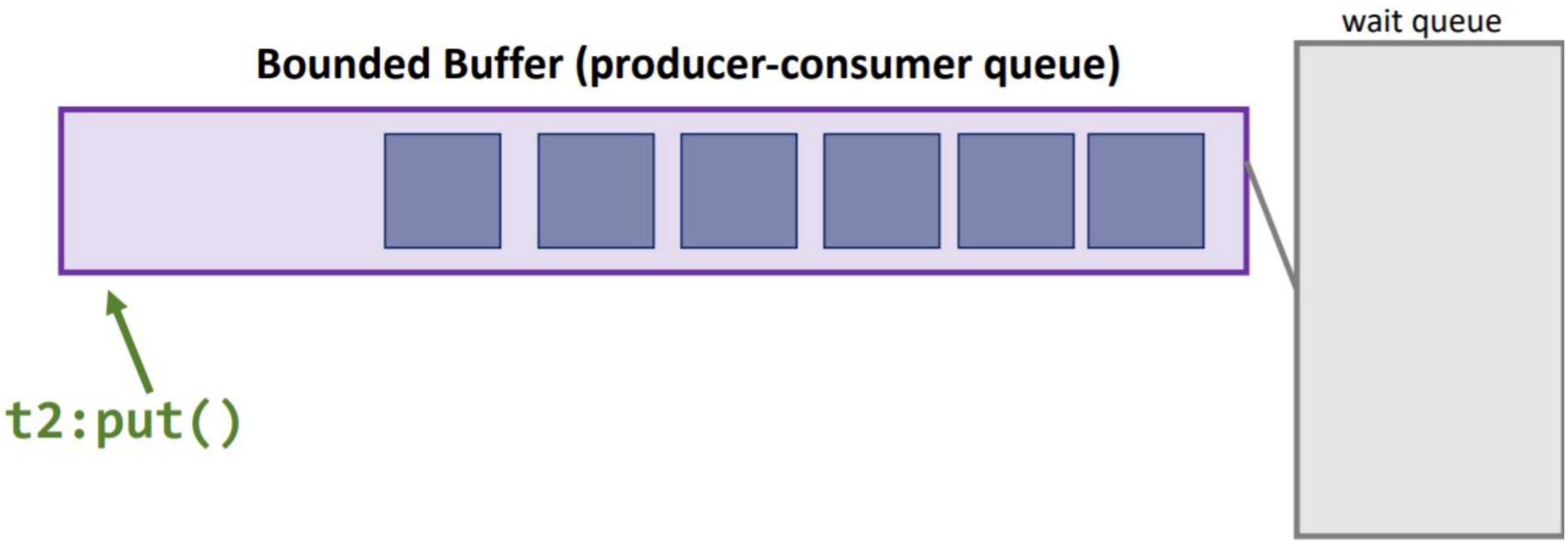
Bounded Buffer (Producer/Consumer)



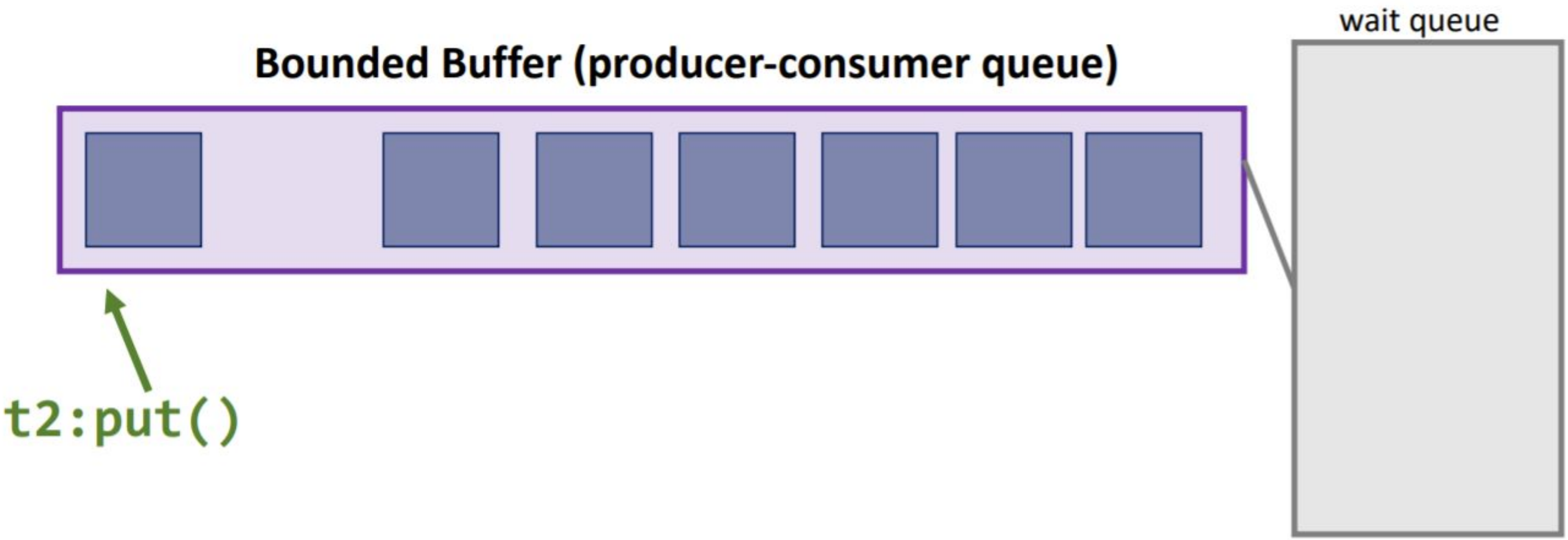
Bounded Buffer (Producer/Consumer)



Bounded Buffer (Producer/Consumer)



Bounded Buffer (Producer/Consumer)



Bounded Buffer (Producer/Consumer)

- Reads and writes to buffer require locking
- when buffers are full
  - writers wait
- when buffers are empty
  - readers wait

Bounded Buffer (Producer/Consumer)

- **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

Bounded Buffer: Attempt 1

Producer

```
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);
}
```

Consumer

```
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);
}
```

- Starting simple: assume one producer, one consumer
- **numfull**: number of elements in the buffer
- Does this code work for 1P and 1C?
  - Yes ☺️

Bounded Buffer: Attempt 1

Producer

```
for (int i=0; i<loops; i++) {
  mutex_lock(&m);
  while (numfull == MAX)
    cond_wait(&cond, &m); //a
  do_fill(i);             //b
  cond_signal(&cond);     //c
  mutex_unlock(&m);
}
```

Consumer

```
while(1) {
  mutex_lock(&m);
  while (numfull == 0)
    cond_wait(&cond, &m); //x
  int tmp = do_get();     //y
  cond_signal(&cond);     //z
  mutex_unlock(&m);
  printf("%d\n", tmp);
}
```

- How about 1P and 2C?
  - No ☹️, **why**?

Bounded Buffer: Attempt 1

Producer

```
for (int i=0; i<loops; i++) {
  mutex_lock(&m);
  while (numfull == MAX)
    cond_wait(&cond, &m); //a
  do_fill(i);             //b
  cond_signal(&cond);     //c
  mutex_unlock(&m);
}
```

Consumer

```
while(1) {
  mutex_lock(&m);
  while (numfull == 0)
    cond_wait(&cond, &m); //x
  int tmp = do_get();     //y
  cond_signal(&cond);     //z
  mutex_unlock(&m);
  printf("%d\n", tmp);
}
```

- Assume C1 and C2 are all waiting at x due to empty queue
- 1) P adds an item to buffer (**line b**), signals cond **(line c**), waking up **C1**, waits on cond until signaled (**line a**)
- 2) C1 is awoken, removes item from buffer (**line y**), signals cond (**line z**), waking up C2, finds buffer empty, goes to sleep (**line x**)
- 3) C2, being woken up by C1, finds buffer empty, goes to sleep waiting on cond (**line x**)

Bounded Buffer: Attempt 1

Producer

```
for (int i=0; i<loops; i++) {
  mutex_lock(&m);
  while (numfull == MAX)
    cond_wait(&cond, &m); //a
  do_fill(i);             //b
  cond_signal(&cond);     //c
  mutex_unlock(&m);
}
```

Consumer

```
while(1) {
  mutex_lock(&m);
  while (numfull == 0)
    cond_wait(&cond, &m); //x
  int tmp = do_get();     //y
  cond_signal(&cond);     //z
  mutex_unlock(&m);
  printf("%d\n", tmp);
}
```

- Everyone is sleeping ➔ P can't produce ➔ no forward progress
- C1's signal was meant to awaken P but it awoke C2

Solution 1: Wake up Everyone

- When not sure if next waiting thread is the right one to wake up, just wake up all
- Not the most elegant solution (that’s Solution 2)
  - Probably bad for performance: all awoken threads will compete for mutex again
  - But a good fallback mechanism to ensure correctness
- Need a new API: **cond_broadcast**(cv)
  - Semantic: wakes up all the queues waiting on cv

Solution 2: Use Multiple CVs

- Identify different conditions that need waiting for
- Use a separate CV for each condition using **cond_wait**() and **cond_signal**()
- More elegant, better-performing solution than using **cond_broadcast**()
- Different conditions in bounded buffer problem?
    - Two
    - Waiting for queue to become non-full
    - Waiting for queue to become non-empty

Bounded Buffer: Correct & Elegant Solution

Producer

```
for (int i=0; i<loops; i++) {
  mutex_lock(&m);
  while (numfull == MAX)
    cond_wait(&non_full, &m);
  do_fill(i);
  cond_signal(&non_empty);
  mutex_unlock(&m);
}
```

Consumer

```
while(1) {
  mutex_lock(&m);
  while (numfull == 0)
    cond_wait(&non_empty, &m);
  int tmp = do_get();
  cond_signal(&non_full);
  mutex_unlock(&m);
  printf("%d\n", tmp);
}
```

- Would it be okay also to use two mutexes?
- Why not?
  - Because mutex protects associated with the shared state (buffer, in this case)

Summary: CV Rules of Thumb

- Keep state in addition to CVs
- Always **cond_wait**() or **cond_signal**() with lock held
- Use different CVs for different conditions
- Recheck state assumptions when waking up from waiting

THANK YOU!