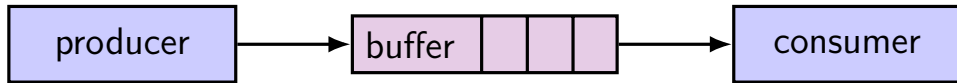


example: producer/consumer

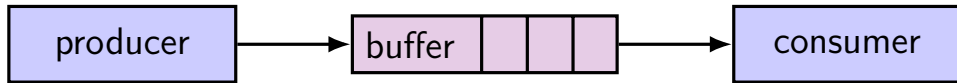


shared buffer (queue) of fixed size

one or more producers inserts into queue

one or more consumers removes from queue

example: producer/consumer



shared buffer (queue) of fixed size

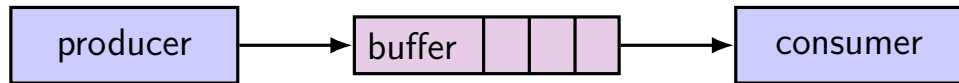
one or more producers inserts into queue

one or more consumers removes from queue

producer(s) and consumer(s) don't work in lockstep

(might need to wait for each other to catch up)

example: producer/consumer



shared buffer (queue) of fixed size

one or more producers inserts into queue

one or more consumers removes from queue

producer(s) and consumer(s) don't work in lockstep

(might need to wait for each other to catch up)

example: C compiler

preprocessor → compiler → assembler → linker

monitors/condition variables

locks for mutual exclusion

condition variables for waiting for event

operations: wait (for event); signal/broadcast (that event happened)

related data structures

monitor = lock + 0 or more condition variables + shared data

Java: every object is a monitor (has instance variables, built-in lock, cond. var)

pthread: build your own: provides you locks + condition variables

monitor idea

a monitor

lock
shared data
condvar 1
condvar 2
...
operation1(...)
operation2(...)

monitor idea

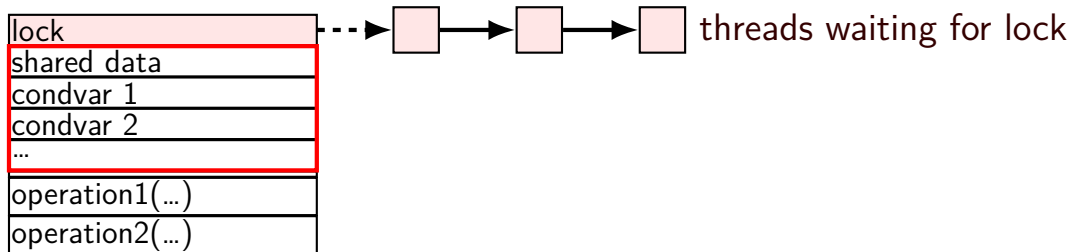
a monitor

lock
shared data
condvar 1
condvar 2
...
operation1(...)
operation2(...)

lock must be acquired
before accessing
any part of monitor's stuff

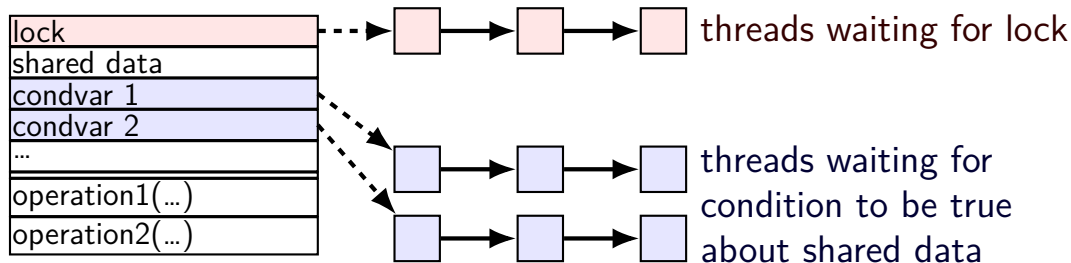
monitor idea

a monitor



monitor idea

a monitor



condvar operations

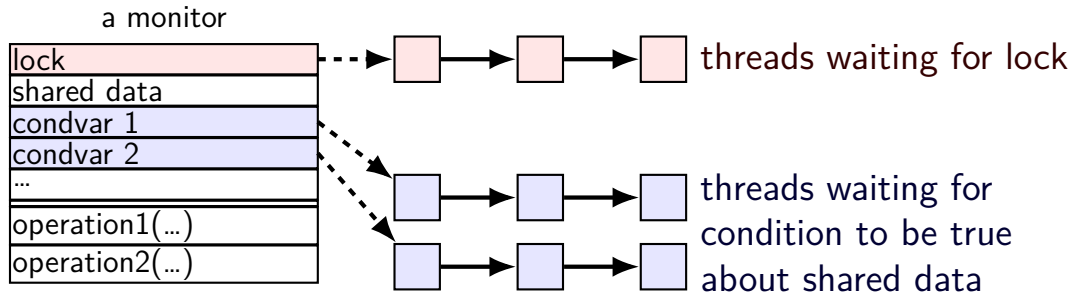
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

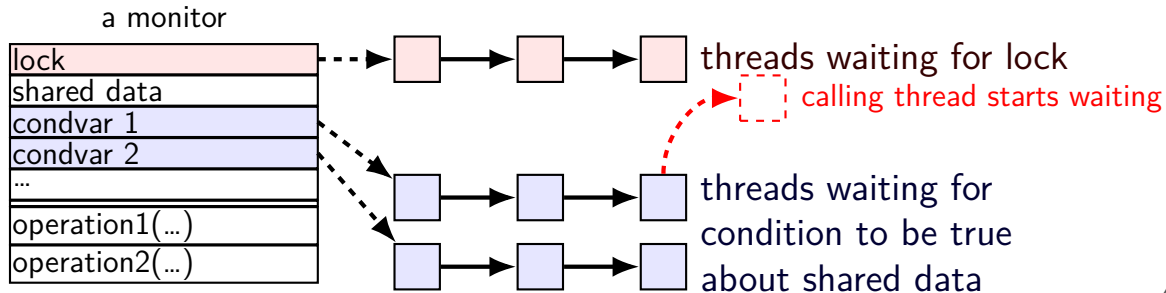
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

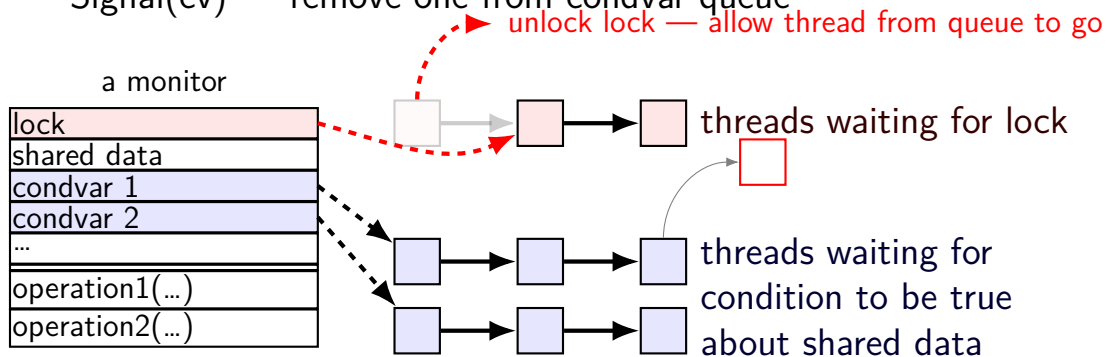
condvar operations:

Wait(cv, lock) — **unlock** lock, add current thread to cv queue

...and **reacquire** lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

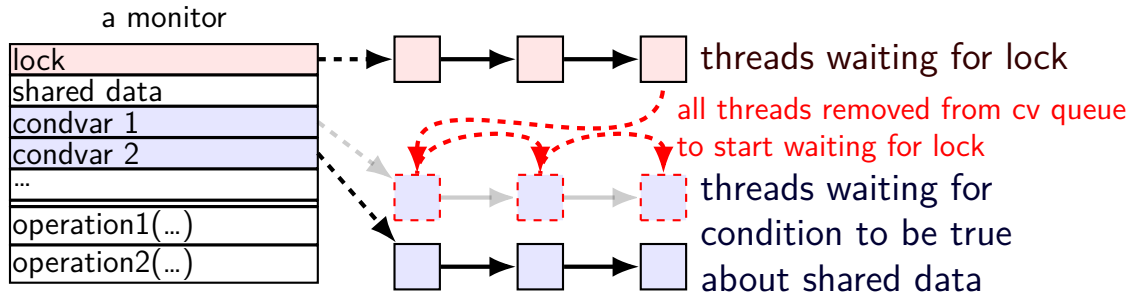
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

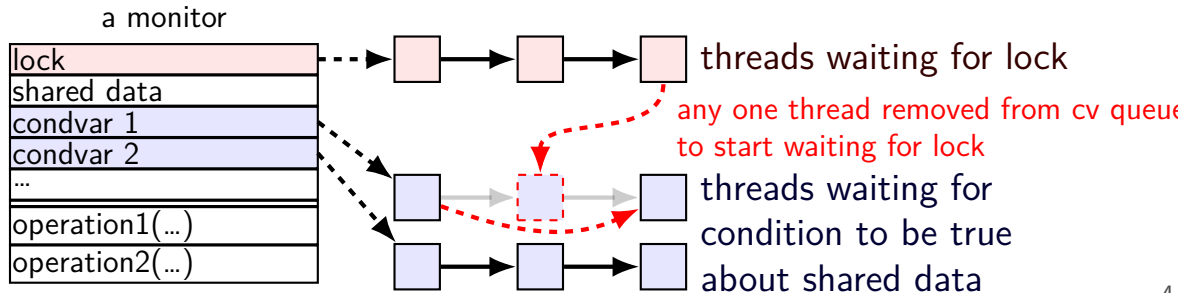
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



pthread cv usage

// MISSING: init calls, etc.

```
pthread_mutex_t lock;  
bool finished;    // data, only accessed with after acquiring lock  
pthread_cond_t finished_cv; // to wait for 'finished' to be true
```

```
void WaitForFinished() {  
    pthread_mutex_lock(&lock);  
    while (!finished) {  
        pthread_cond_wait(&finished_cv, &lock);  
    }  
    pthread_mutex_unlock(&lock);  
}
```

```
void Finish() {  
    pthread_mutex_lock(&lock);  
    finished = true;  
    pthread_cond_broadcast(&finished_cv);  
    pthread_mutex_unlock(&lock);  
}
```

pthread cv usage

// MISSING: init calls, etc.

```
pthread_mutex_t lock;
```

```
bool finished; // data, only accessed with after acquiring lock
```

```
pthread_cond_t finished_cv; // to wait for 'finished' to be true
```

```
void WaitForFinished() {
```

```
    pthread_mutex_lock(&lock);
```

```
    while (!finished) {
```

```
        pthread_cond_wait(&finished_cv, &lock);
```

```
    }
```

```
    pthread_mutex_unlock(&lock);
```

```
}
```

acquire lock before
reading or writing finished

```
void Finish() {
```

```
    pthread_mutex_lock(&lock);
```

```
    finished = true;
```

```
    pthread_cond_broadcast(&finished_cv);
```

```
    pthread_mutex_unlock(&lock);
```

```
}
```

pthread cv usage

// MISSING: init calls, etc.

```
pthread_mutex_t lock;
```

```
bool finished;    // data, only accessed with after acquiring lock
```

```
pthread_cond_t finished_cv; // to wait for 'finished' to be true
```

```
void WaitForFinished() {  
    pthread_mutex_lock(&lock);  
    while (!finished) {  
        pthread_cond_wait(&finished_cv, &lock);  
    }  
    pthread_mutex_unlock(&lock);  
}
```

check whether we need to wait at all
(why a loop? we'll explain later)

```
void Finish() {  
    pthread_mutex_lock(&lock);  
    finished = true;  
    pthread_cond_broadcast(&finished_cv);  
    pthread_mutex_unlock(&lock);  
}
```



pthread cv usage

// MISSING: init calls, etc.

```
pthread_mutex_t lock;  
bool finished;    // data, only accessed with after acquiring lock  
pthread_cond_t finished_cv; // to wait for 'finished' to be true
```

```
void WaitForFinished() {  
    pthread_mutex_lock(&lock);  
    while (!finished) {  
        pthread_cond_wait(&finished_cv, &lock);  
    }  
    pthread_mutex_unlock(&lock);  
}
```

```
void Finish() {  
    pthread_mutex_lock(&lock);  
    finished = true;  
    pthread_cond_broadcast(&finished_cv);  
    pthread_mutex_unlock(&lock);  
}
```



know we need to wait
(finished can't change while we have lock)
so wait, releasing lock...

pthread cv usage

// MISSING: init calls, etc.

```
pthread_mutex_t lock;
```

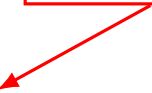
```
bool finished;    // data, only accessed with after acquiring lock
```

```
pthread_cond_t finished_cv; // to wait for 'finished' to be true
```

```
void WaitForFinished() {  
    pthread_mutex_lock(&lock);  
    while (!finished) {  
        pthread_cond_wait(&finished_cv, &lock);  
    }  
    pthread_mutex_unlock(&lock);  
}
```

```
void Finish() {  
    pthread_mutex_lock(&lock);  
    finished = true;  
    pthread_cond_broadcast(&finished_cv);  
    pthread_mutex_unlock(&lock);  
}
```

allow all waiters to proceed
(once we unlock the lock)



WaitForFinish timeline 1

WaitForFinish thread	Finish thread
mutex_lock(&lock) (thread has lock)	
	mutex_lock(&lock) (start waiting for lock)
while (!finished) ... cond_wait(&finished_cv, &lock); (start waiting for cv)	(done waiting for lock)
	finished = true cond_broadcast(&finished_cv)
(done waiting for cv) (start waiting for lock)	
	mutex_unlock(&lock)
(done waiting for lock) while (!finished) ... (finished now true, so return)	

WaitForFinish timeline 2

WaitForFinish thread	Finish thread
	<code>mutex_lock(&lock)</code> <code>finished = true</code> <code>cond_broadcast(&finished_cv)</code> <code>mutex_unlock(&lock)</code>
<code>mutex_lock(&lock)</code> <code>while (!finished) ...</code> (finished now true, so return) <code>mutex_unlock(&lock)</code>	

why the loop

```
while (!finished) {  
    pthread_cond_wait(&finished_cv, &lock);  
}
```

we only broadcast if finished is true

so why check finished afterwards?

why the loop

```
while (!finished) {  
    pthread_cond_wait(&finished_cv, &lock);  
}
```

we only broadcast if finished is true

so why check finished afterwards?

pthread_cond_wait manual page:

“**Spurious wakeups** ... may occur.”

spurious wakeup = wait returns even though nothing happened

unbounded buffer producer/consumer

```
pthread_mutex_t lock;  
pthread_cond_t data_ready;  
UnboundedQueue buffer;
```

```
Produce(item) {  
    pthread_mutex_lock(&lock);  
    buffer.enqueue(item);  
    pthread_cond_signal(&data_ready);  
    pthread_mutex_unlock(&lock);  
}  
Consume() {  
    pthread_mutex_lock(&lock);  
    while (buffer.empty()) {  
        pthread_cond_wait(&data_ready, &lock);  
    }  
    item = buffer.dequeue();  
    pthread_mutex_unlock(&lock);  
    return item;  
}
```

unbounded buffer producer/consumer

```
pthread_mutex_t lock;  
pthread_cond_t data_ready;  
UnboundedQueue buffer;
```

```
Produce(item) {  
    pthread_mutex_lock(&lock);  
    buffer.enqueue(item);  
    pthread_cond_signal(&data_ready);  
    pthread_mutex_unlock(&lock);  
}
```

```
Consume() {  
    pthread_mutex_lock(&lock);  
    while (buffer.empty()) {  
        pthread_cond_wait(&data_ready, &lock);  
    }  
    item = buffer.dequeue();  
    pthread_mutex_unlock(&lock);  
    return item;  
}
```

rule: never touch buffer
without acquiring lock

otherwise: what if two threads
simultaneously en/dequeue?
(both use same array/linked list entry?)
(both reallocate array?)

unbounded buffer producer/consumer

```
pthread_mutex_t lock;  
pthread_cond_t data_ready;  
UnboundedQueue buffer;
```

```
Produce(item) {  
    pthread_mutex_lock(&lock);  
    buffer.enqueue(item);  
    pthread_cond_signal(&data_ready);  
    pthread_mutex_unlock(&lock);  
}
```

```
Consume() {  
    pthread_mutex_lock(&lock);  
    while (buffer.empty()) {  
        pthread_cond_wait(&data_ready, &lock);  
    }  
    item = buffer.dequeue();  
    pthread_mutex_unlock(&lock);  
    return item;  
}
```

check if empty
if so, dequeue


okay because have lock
other threads cannot dequeue here

unbounded buffer producer/consumer

```
pthread_mutex_t lock;  
pthread_cond_t data_ready;  
UnboundedQueue buffer;
```

```
Produce(item) {  
    pthread_mutex_lock(&lock);  
    buffer.enqueue(item);  
    pthread_cond_signal(&data_ready);  
    pthread_mutex_unlock(&lock);  
}
```

wake one Consume thread
if any are waiting



```
Consume() {  
    pthread_mutex_lock(&lock);  
    while (buffer.empty()) {  
        pthread_cond_wait(&data_ready, &lock);  
    }  
    item = buffer.dequeue();  
    pthread_mutex_unlock(&lock);  
    return item;  
}
```

unbounded buffer producer/consumer

```
pthread_mutex_t lock;  
pthread_cond_t data_ready;  
UnboundedQueue buffer;
```

```
Produce(item) {  
    pthread_mutex_lock(&lock);  
    buffer.enqueue(item);  
    pthread_cond_signal(&data_ready);  
    pthread_mutex_unlock(&lock);  
}
```

```
Consume() {  
    pthread_mutex_lock(&lock);  
    while (buffer.empty()) {  
        pthread_cond_wait(&data_ready, &lock);  
    }  
    item = buffer.dequeue();  
    pthread_mutex_unlock(&lock);  
    return item;  
}
```

Thread 1

Produce()
...lock
...enqueue
...signal
...unlock

Thread 2

Consume()
...lock
...empty? no
...dequeue
...unlock
return

0 iterations: Produce() called before Consume()
1 iteration: Produce() signalled, probably
2+ iterations: spurious wakeup or ...?

unbounded buffer producer/consumer

```
pthread_mutex_t lock;  
pthread_cond_t data_ready;  
UnboundedQueue buffer;
```

```
Produce(item) {  
    pthread_mutex_lock(&lock);  
    buffer.enqueue(item);  
    pthread_cond_signal(&data_ready);  
    pthread_mutex_unlock(&lock);  
}  
Consume() {  
    pthread_mutex_lock(&lock);  
    while (buffer.empty()) {  
        pthread_cond_wait(&data_ready, &lock);  
    }  
    item = buffer.dequeue();  
    pthread_mutex_unlock(&lock);  
    return item;  
}
```

Thread 1

Thread 2

	Consume()
	...lock
	...empty? yes
	...unlock/start wait
Produce()	waiting for data_ready
...lock	
...enqueue	
...signal	stop wait
...unlock	lock
	...empty? no
	...dequeue
	...unlock
	return

0 iterations: Produce() called before Consume()
1 iteration: Produce() signalled, probably
2+ iterations: spurious wakeup or ...?

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;
```

```
Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}
```

```
Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

Thread 1

Produce()
...lock
...enqueue
...signal
...unlock

Thread 2

Consume()
...lock
...empty? yes
...unlock/start wait
waiting for data_ready
stop wait
waiting for lock
...lock
...empty? yes
...unlock/start wait

Thread 3

Consume()
waiting for lock
lock
...empty? no
...dequeue
...unlock
return

0 iterations: Produce() called before Consume()
 1 iteration: Produce() signalled, probably
 2+ iterations: spurious wakeup or ...?

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;
```

in pthreads: signalled thread not
gaurenteed to hold lock next

alternate design:
signalled thread gets lock next
called "Hoare scheduling"
not done by pthreads, Java, ...

```
pthread_cond_wait(&data_r
```

```
}
item = buffer.dequeue();
pthread_mutex_unlock(&lock);
return item;
```

```
}
```

Thread 1

```
Produce()
...lock
...enqueue
...signal
...unlock
```

Thread 2

```
Consume()
...lock
...empty? yes
...unlock/start wait
waiting for
data_ready
stop wait
waiting for
lock
...lock
...empty? yes
...unlock/start wait
```

Thread 3

```
Consume()
waiting for
lock
lock
...empty? no
...dequeue
...unlock
return
```

0 iterations: Produce() called before Consume()
1 iteration: Produce() signalled, probably
2+ iterations: spurious wakeup or ...?

Hoare versus Mesa monitors

Hoare-style monitors

- signal 'hands off' lock to awoken thread

Mesa-style monitors

- any eligible thread gets lock next
(maybe some other idea of priority?)

every current threading library I know of does Mesa-style

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;
Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}
Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

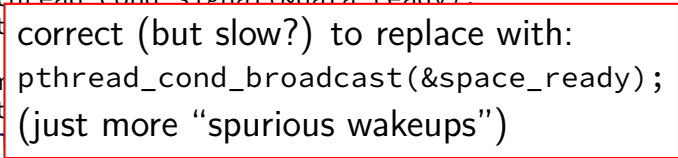

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;
Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}
Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;
Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
}
pthread_cond_broadcast(&space_ready);
pthread_cond_wait(&data_ready, &lock);
}
item = buffer.dequeue();
pthread_cond_signal(&space_ready);
pthread_mutex_unlock(&lock);
return item;
}
```

correct (but slow?) to replace with:
(just more “spurious wakeups”)



bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;
Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}
Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

correct but slow to replace
data_ready and space_ready
with 'combined' condvar ready
and use broadcast
(just more "spurious wakeups")

monitor pattern

```
pthread_mutex_lock(&lock);
while (!condition A) {
    pthread_cond_wait(&condvar_for_A, &lock);
}
... /* manipulate shared data, changing other conditions */
if (set condition A) {
    pthread_cond_broadcast(&condvar_for_A);
    /* or signal, if only one thread cares */
}
if (set condition B) {
    pthread_cond_broadcast(&condvar_for_B);
    /* or signal, if only one thread cares */
}
...
pthread_mutex_unlock(&lock)
```

monitors rules of thumb

never touch shared data without holding the lock

keep lock held for **entire operation**:

verifying condition (e.g. buffer not full) *up to and including*
manipulating data (e.g. adding to buffer)

create condvar for every kind of scenario waited for

always write **loop** calling cond_wait to wait for condition X

broadcast/signal condition variable **every time you change X**

monitors rules of thumb

never touch shared data without holding the lock

keep lock held for **entire operation**:

verifying condition (e.g. buffer not full) *up to and including*
manipulating data (e.g. adding to buffer)

create condvar for every kind of scenario waited for

always write **loop** calling `cond_wait` to wait for condition X

broadcast/signal condition variable **every time you change X**

correct but slow to...

broadcast when just signal would work

broadcast or signal when nothing changed

use one condvar for multiple conditions

mutex/cond var init/destroy

```
pthread_mutex_t mutex;  
pthread_cond_t cv;  
pthread_mutex_init(&mutex, NULL);  
pthread_cond_init(&cv, NULL);  
// --OR--  
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;  
  
// and when done:  
...  
pthread_cond_destroy(&cv);  
pthread_mutex_destroy(&mutex);
```

wait for both finished

// MISSING: init calls, etc.

```
pthread_mutex_t lock;  
bool finished[2];  
pthread_cond_t both_finished_cv;
```

```
void WaitForBothFinished() {  
    pthread_mutex_lock(&lock);  
    while (_____) {  
        pthread_cond_wait(&both_finished_cv, &lock);  
    }  
    pthread_mutex_unlock(&lock);  
}
```

```
void Finish(int index) {  
    pthread_mutex_lock(&lock);  
    finished[index] = true;  
    -----  
    pthread_mutex_unlock(&lock);  
}
```


wait for both finished

// MISSING: init calls, etc.

```
pthread_mutex_t lock;  
bool finished[2];  
pthread_cond_t both_finished_cv;
```

```
void WaitForBothFinished() {  
    pthread_mutex_lock(&lock);  
    while (-----) {  
        pthread_cond_wait(&both_finished_cv, &lock);  
    }  
    pthread_mutex_unlock(&lock);  
}
```

```
void Finish(int index) {  
    pthread_mutex_lock(&lock);  
    finished[index] = true;  
    -----  
    pthread_mutex_unlock(&lock);  
}
```

- A. `finished[0] && finished[1]`
- B. `finished[0] || finished[1]`
- C. `!finished[0] || !finished[1]`
- D. `finished[0] != finished[1]`
- E. something else

wait for both finished

// MISSING: init calls, etc.

```
pthread_mutex_t lock;  
bool finished[2];  
pthread_cond_t both_finished;
```

```
void WaitForBothFinished
```

```
{  
    pthread_mutex_lock(&lock);
```

```
    while (_____
```

```
        pthread_cond_wait(&both_finished_cv, &lock);
```

```
}
```

```
    pthread_mutex_unlock(&lock);
```

```
}
```

```
void Finish(int index) {
```

```
    pthread_mutex_lock(&lock);
```

```
    finished[index] = true;
```

```
    _____  
    pthread_mutex_unlock(&lock);
```

```
}
```

- A. pthread_cond_signal(&both_finished_cv)
- B. pthread_cond_broadcast(&both_finished_cv)
- C. if (finished[1-index])
 pthread_cond_signal(&both_finished_cv);
- D. if (finished[1-index])
 pthread_cond_broadcast(&both_finished_cv);
- E. something else

monitor exercise: barrier

suppose we want to implement a one-use barrier; fill in blanks:

```
struct BarrierInfo {  
    pthread_mutex_t lock;  
    int total_threads; // initially total # of threads  
    int number_reached; // initially 0  
    -----  
};  
  
void BarrierWait(BarrierInfo *b) {  
    pthread_mutex_lock(&b->lock);  
    ++b->number_reached;  
    if (b->number_reached == b->total_threads) {  
        -----  
    } else {  
        -----  
    }  
    pthread_mutex_unlock(&b->lock);  
}
```


extra exercises

producer/consumer signal?

```
pthread_mutex_t lock;  
pthread_cond_t data_ready;  
UnboundedQueue buffer;
```

```
Produce(item) {  
    pthread_mutex_lock(&lock);  
    buffer.enqueue(item);  
    /* GOOD CODE: pthread_cond_signal(&data_ready); */  
    /* BAD CODE: */  
    if (buffer.size() == 1)  
        pthread_cond_signal(&item);  
    pthread_mutex_unlock(&lock);  
}
```

```
Consume() {  
    pthread_mutex_lock(&lock);  
    while (buffer.empty()) {  
        pthread_cond_wait(&data_ready, &lock);  
    }  
    item = buffer.dequeue();  
    pthread_mutex_unlock(&lock);  
}
```

bad case (setup)

thread 0	1	2	3
Consume(): lock empty? wait on cv	Consume(): lock empty? wait on cv	Produce(): lock	Produce():

bad case

thread 0	1	2	3
Consume(): lock empty? wait on cv wait for lock gets lock dequeue	Consume(): lock empty? wait on cv	Produce(): lock enqueue size = 1? signal unlock	Produce(): wait for lock gets lock enqueue size \neq 1: don't signal unlock

monitor exercise: ConsumeTwo

suppose we want producer/consumer, but...

but change Consume() to ConsumeTwo() which returns a **pair of values**

and don't want two calls to ConsumeTwo() to wait...
with each getting one item

what should we change below?

```
pthread_mutex_t lock;  
pthread_cond_t data_ready;  
UnboundedQueue buffer;
```

```
Produce(item) {  
    pthread_mutex_lock(&lock);  
    buffer.enqueue(item);  
    pthread_cond_signal(&data_ready);  
    pthread_mutex_unlock(&lock);  
}
```

```
Consume() {  
    pthread_mutex_lock(&lock);  
    while (buffer.empty()) {  
        pthread_cond_wait(&data_ready, &lock);  
    }  
    item = buffer.dequeue();  
    pthread_mutex_unlock(&lock);  
    return item;  
}
```

monitor exercise: solution (1)

(one of many possible solutions)

Assuming ConsumeTwo **replaces** Consume:

```
Produce() {  
    pthread_mutex_lock(&lock);  
    buffer.enqueue(item);  
    if (buffer.size() > 1) { pthread_cond_signal(&data_ready); }  
    pthread_mutex_unlock(&lock);  
}  
ConsumeTwo() {  
    pthread_mutex_lock(&lock);  
    while (buffer.size() < 2) { pthread_cond_wait(&data_ready, &lock); }  
    item1 = buffer.dequeue(); item2 = buffer.dequeue();  
    pthread_mutex_unlock(&lock);  
    return Combine(item1, item2);  
}
```

monitor exercise: solution (2)

(one of many possible solutions)

Assuming ConsumeTwo is **in addition to** Consume (using two CVs):

```
Produce() {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&one_ready);
    if (buffer.size() > 1) { pthread_cond_signal(&two_ready); }
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.size() < 1) { pthread_cond_wait(&one_ready, &lock); }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}

ConsumeTwo() {
    pthread_mutex_lock(&lock);
    while (buffer.size() < 2) { pthread_cond_wait(&two_ready, &lock); }
    item1 = buffer.dequeue(); item2 = buffer.dequeue();
    pthread_mutex_unlock(&lock);
}
```

monitor exercise: slower solution

(one of many possible solutions)

Assuming ConsumeTwo is **in addition to** Consume (using one CV):

```
Produce() {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    // broadcast and not signal, b/c we might wakeup only ConsumeTwo() otherwise
    pthread_cond_broadcast(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.size() < 1) { pthread_cond_wait(&data_ready, &lock); }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}

ConsumeTwo() {
    pthread_mutex_lock(&lock);
    while (buffer.size() < 2) { pthread_cond_wait(&data_ready, &lock); }
    item1 = buffer.dequeue(); item2 = buffer.dequeue();
    pthread_mutex_unlock(&lock);
}
```

monitor exercise: ordering

suppose we want producer/consumer, but...

but want to ensure first call to Consume() **always** returns first

(no matter what ordering cond_signal/cond_broadcast use)

```
pthread_mutex_t lock;  
pthread_cond_t data_ready;  
UnboundedQueue buffer;
```

```
Produce(item) {  
    pthread_mutex_lock(&lock);  
    buffer.enqueue(item);  
    pthread_cond_signal(&data_ready);  
    pthread_mutex_unlock(&lock);  
}
```

```
Consume() {  
    pthread_mutex_lock(&lock);  
    while (buffer.empty()) {  
        pthread_cond_wait(&data_ready, &lock);  
    }  
    item = buffer.dequeue();  
    pthread_mutex_unlock(&lock);  
    return item;  
}
```

monitor ordering exercise: solution

(one of many possible solutions)

```
struct Waiter {  
    pthread_cond_t cv;  
    bool done;  
    T item;  
}  
Queue<Waiter*> waiters;
```

```
Produce(item) {  
    pthread_mutex_lock(&lock);  
    if (!waiters.empty()) {  
        Waiter *waiter = waiters.dequeue();  
        waiter->done = true;  
        waiter->item = item;  
        cond_signal(&waiter->cv);  
        ++num_pending;  
    } else {  
        buffer.enqueue(item);  
    }  
    pthread_mutex_unlock(&lock);  
}
```

```
Consume() {  
    pthread_mutex_lock(&lock);  
    if (buffer.empty()) {  
        Waiter waiter;  
        cond_init(&waiter.cv);  
        waiter.done = false;  
        waiters.enqueue(&waiter);  
        while (!waiter.done)  
            cond_wait(&waiter.cv, &lock);  
        item = waiter.item;  
    } else {  
        item = buffer.dequeue();  
    }  
    pthread_mutex_unlock(&lock);  
    return item;  
}
```

