sync-monitors

monitors/condition variables

locks for mutual exclusion

condition variables for waiting for event

- represents *list of waiting threads*
- operations: wait (for event); signal/broadcast (that event happened)

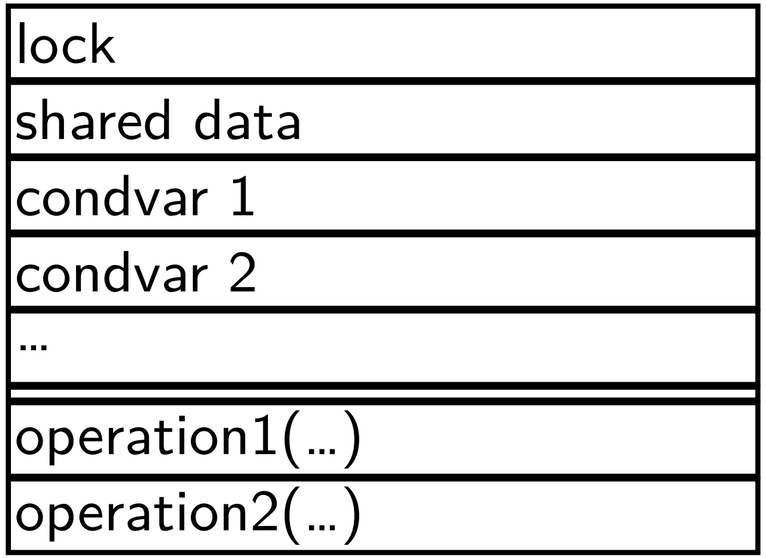
related data structures

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

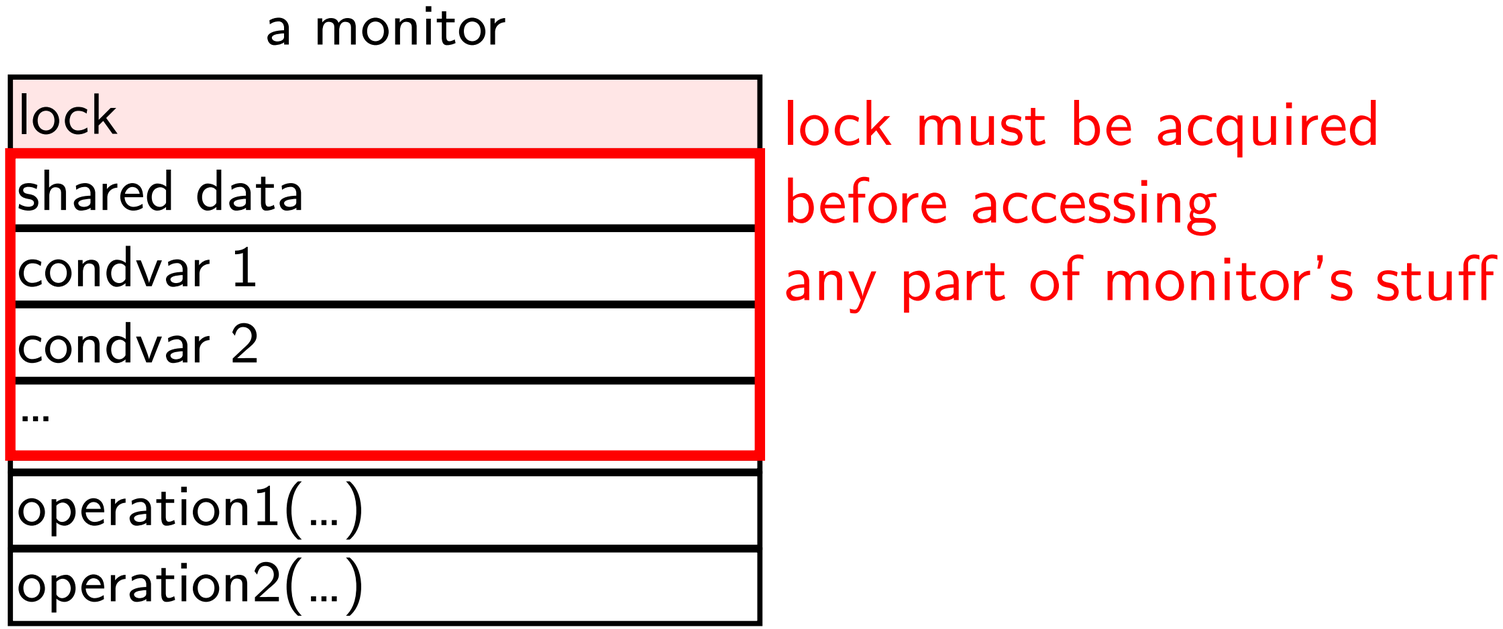
- Java: every object is a monitor (has instance variables, built-in lock, cond. var)
- pthreads: build your own: provides you locks + condition variables

monitor idea

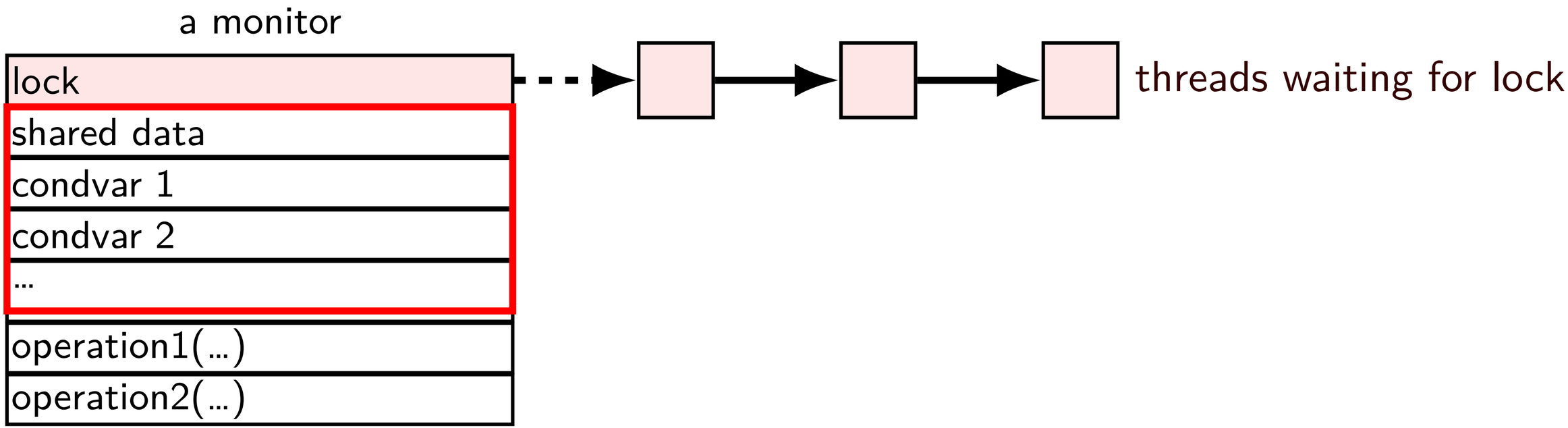
a monitor



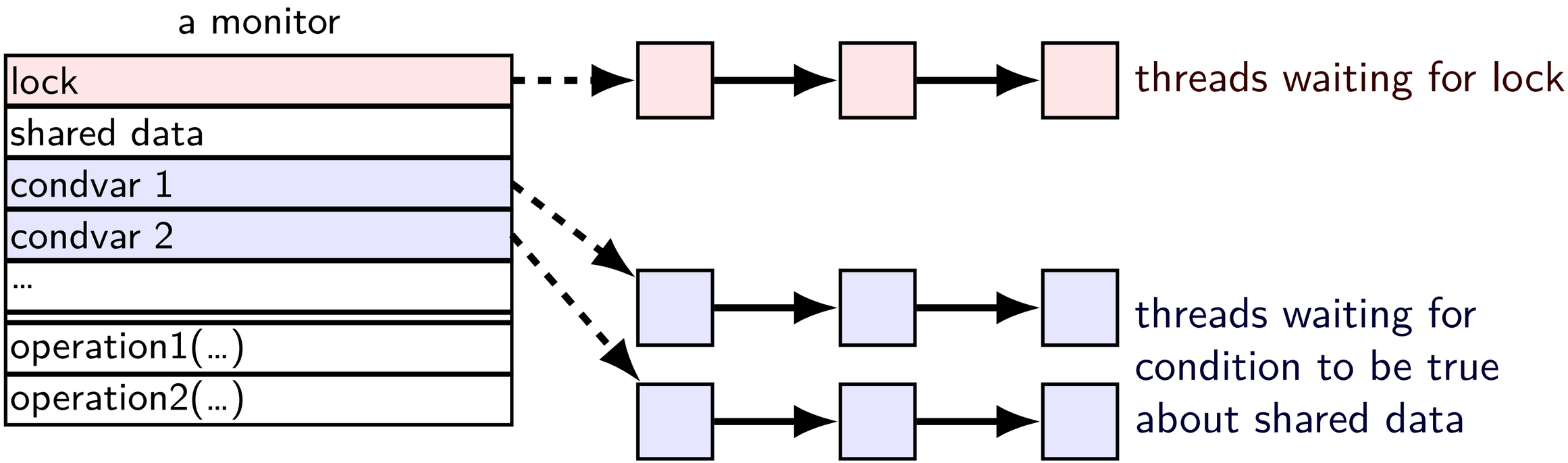
monitor idea



monitor idea



monitor idea



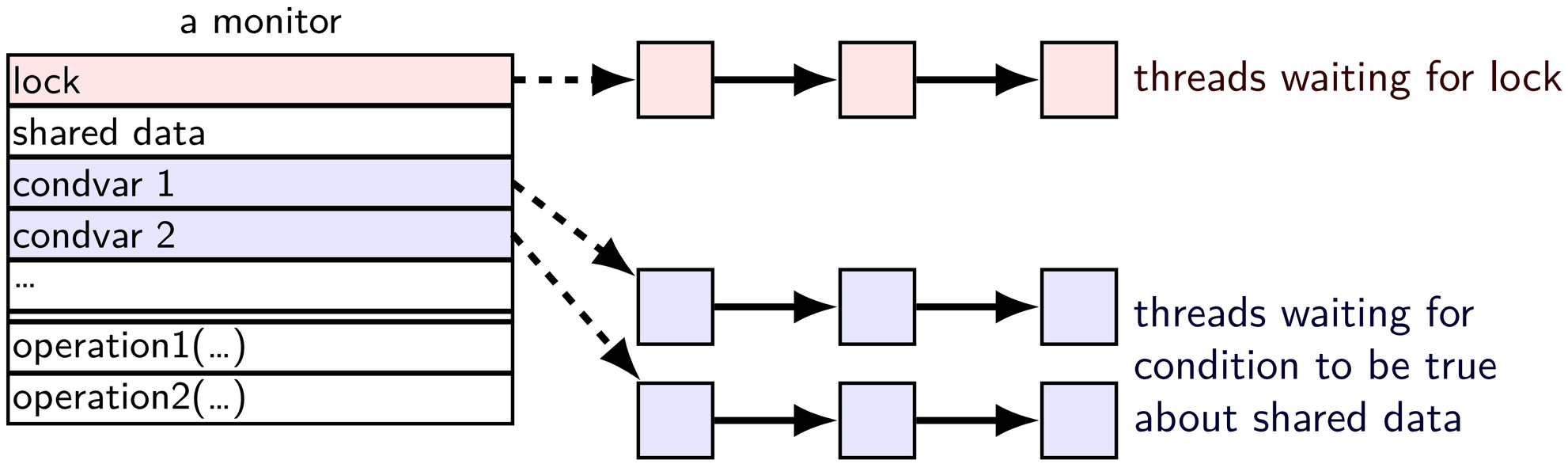
condvar operations

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



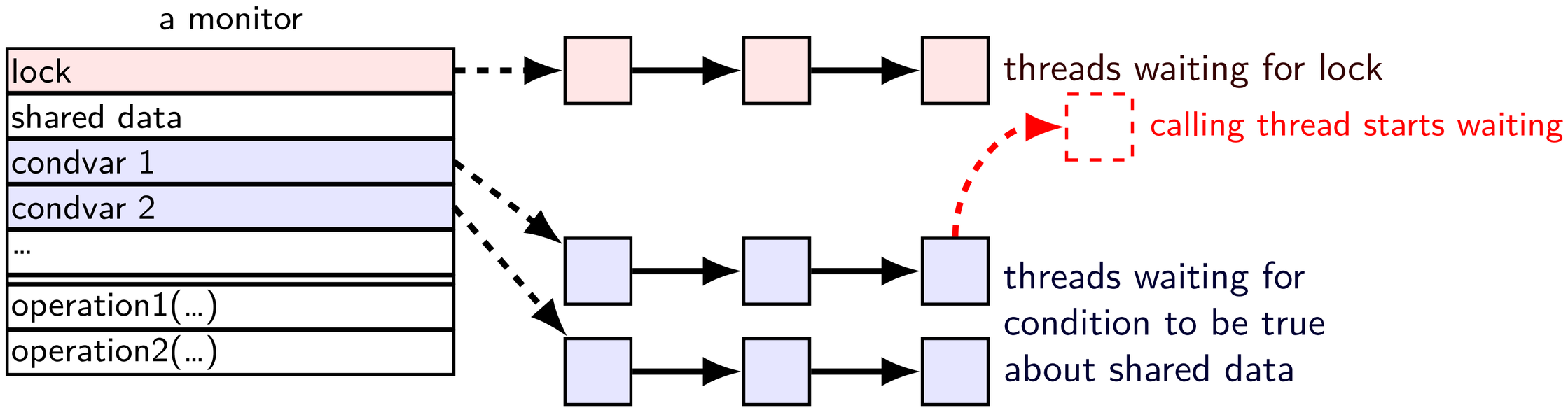
condvar operations

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



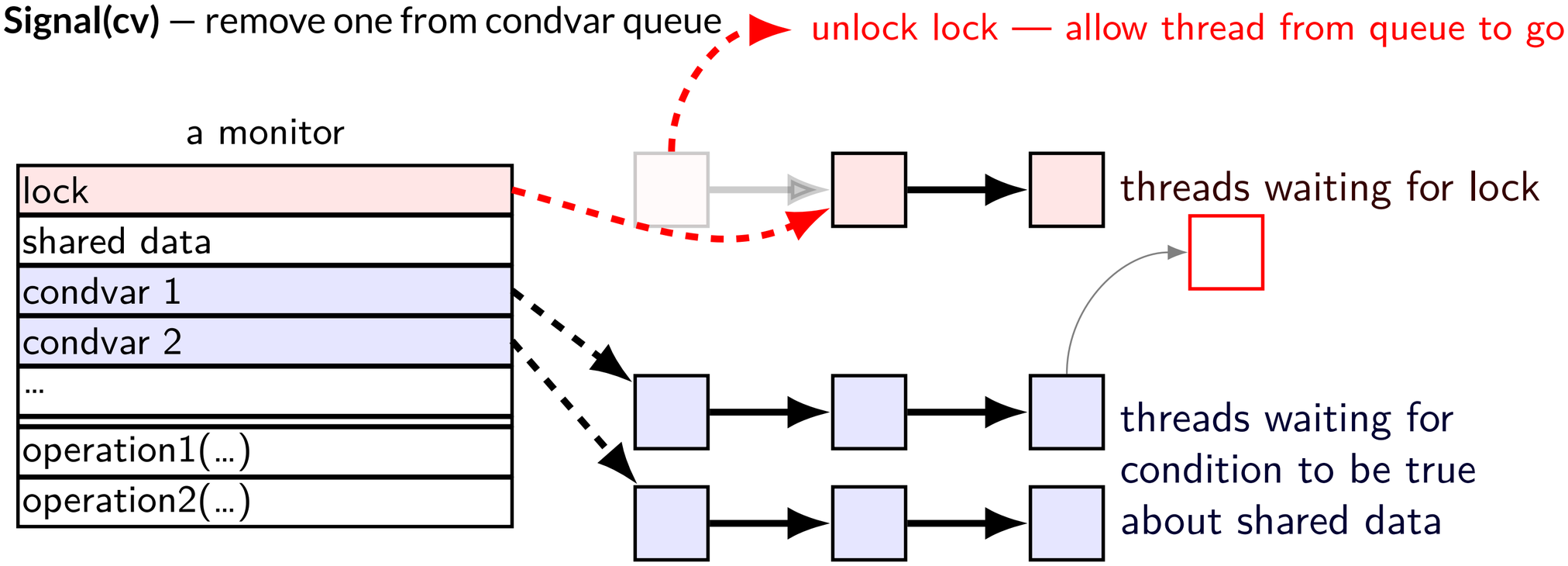
condvar operations

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



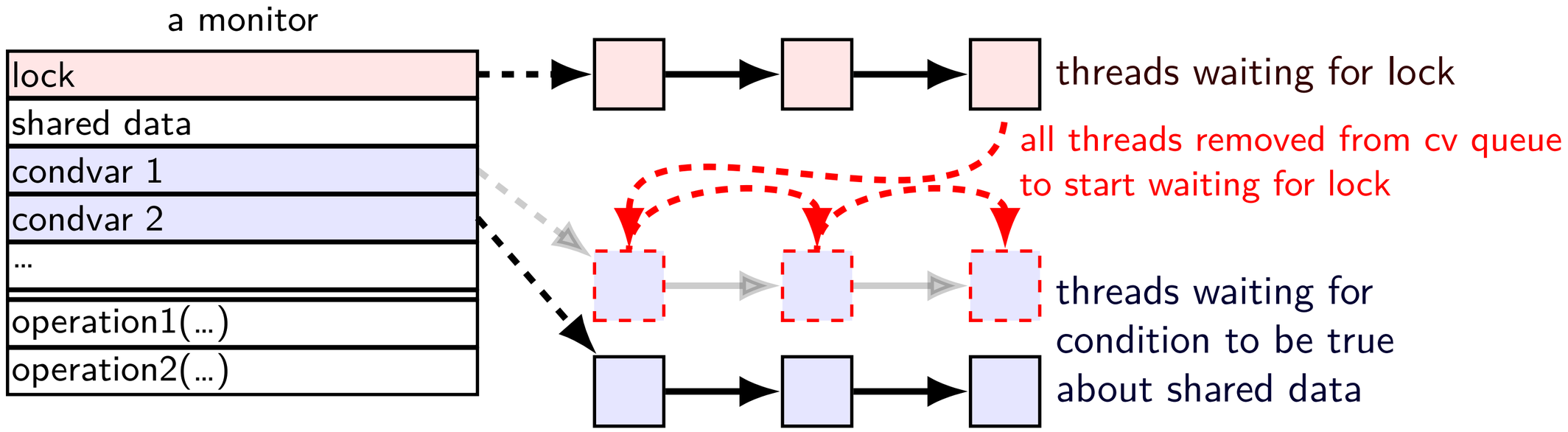
condvar operations

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



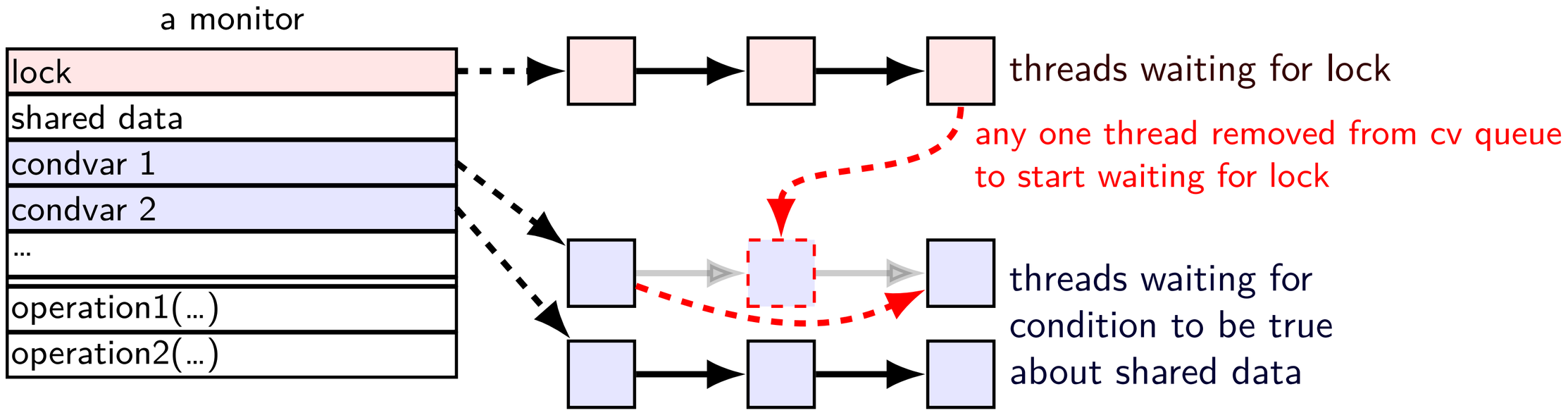
condvar operations

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

```
1  // MISSING: init calls, etc.
2  pthread_mutex_t lock;
3  bool finished;   // data, only accessed with after acquiring lock
4  pthread_cond_t finished_cv;  // to wait for 'finished' to be true
5
6  void WaitForFinished() {
7    pthread_mutex_lock(&lock);
8    while (!finished) {
9      pthread_cond_wait(&finished_cv, &lock);
10   }
11   pthread_mutex_unlock(&lock);
12 }
13
14 void Finish() {
15   pthread_mutex_lock(&lock);
16   finished = true;
17   pthread_cond_broadcast(&finished_cv);
18   pthread_mutex_unlock(&lock);
19 }
20
```

pthread cv usage — lock first

```
1  // MISSING: init calls, etc.
2  pthread_mutex_t lock;
3  bool finished;   // data, only accessed with after acquiring lock
4  pthread_cond_t finished_cv;  // to wait for 'finished' to be true
5
6  void WaitForFinished() {
7    pthread_mutex_lock(&lock);
8    while (!finished) {
9      pthread_cond_wait(&finished_cv, &lock);
10   }
11   pthread_mutex_unlock(&lock);
12 }
13
14 void Finish() {
15   pthread_mutex_lock(&lock);
16   finished = true;
17   pthread_cond_broa
18   pthread_mutex_unl
19 }
20
```

acquire lock before reading or write `finished`

pthread cv usage — check waiting?

```
1  // MISSING: init calls, etc.
2  pthread_mutex_t lock;
3  bool finished;   // data, only accessed with after acquiring lock
4  pthread_cond_t finished_cv;  // to wait for 'finished' to be true
5
6  void WaitForFinished() {
7    pthread_mutex_lock(&lock);
8    while (!finished) {
9      pthread_cond_wait(&finished_cv, &lock);
10   }
11   pthread_mutex_unlock(&lock);
12 }
13
14 void Finish() {
15   pthread_mutex_lock(&lo
16   finished = true;
17   pthread_cond_broadcast
18   pthread_mutex_unlock(&
19 }
20
```

check whether we need to wait at all

yes, must be a loop — we'll explain later

pthread cv usage — do waiting

```
1  // MISSING: init calls, etc.
2  pthread_mutex_t lock;
3  bool finished;   // data, only accessed with after acquiring lock
4  pthread_cond_t finished_cv;  // to wait for 'finished' to be true
5
6  void WaitForFinished() {
7    pthread_mutex_lock(&lock);
8    while (!finished) {
9      pthread_cond_wait(&finished_cv, &lock);
10   }
11   pthread
12 }
13
14 void Fini
15   pthread
16   finishe
17   pthread
18   pthread
19 }
20
```

know we need to wait

(`finished` cannot have changed since we checked because of lock)

so wait releasing lock

important that we release lock, so `finished` can change

pthread cv usage — broadcast

```
1  // MISSING: init calls, etc.
2  pthread_mutex_t lock;
3  bool finished;   // data, only accessed with after acquiring lock
4  pthread_cond_t finished_cv;  // to wait for 'finished' to be true
5
6  void WaitForFinished() {
7    pthread_mutex_lock(&lock);
8    while (!finished) {
9      pthread_cond_wait(&finished_cv, &lock);
10   }
11   pthread_mutex_unlock(&lock);
12 }
13
14 void Finish() {
15   pthread_mutex_lock(&lock);
16   finished = true;
17   pthread_cond_broadcast(&finished_cv);
18   pthread_mutex_unlock(&lock);
19 }
20
```

all waiters to proceed once we release 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)	
`mutex_unlock(&lock)`	

WaitForFinish timeline 2

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

why the loop

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

we only `broadcast` if `finished` is true

so why check `finished` afterwards?

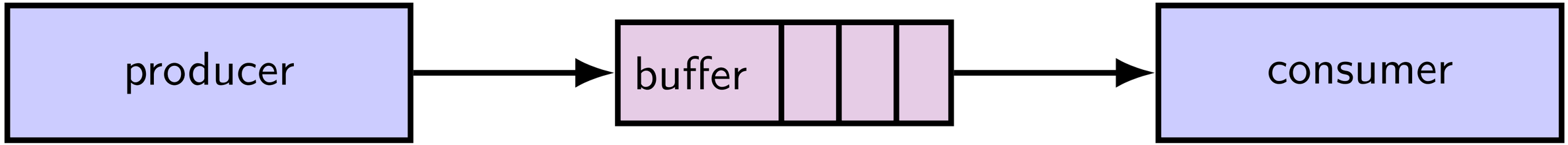
pthread_cond_wait manual page:

> "*Spurious wakeups* … may occur."

spurious wakeup = `wait` returns even though nothing happened

producer/consumer idea

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 $\rightarrow$ compiler $\rightarrow$ assembler $\rightarrow$ linker

unbounded buffer producer/consumer

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  Queue buffer;
4  Produce(item) {
5      pthread_mutex_lock(&lock);
6      buffer.enqueue(item);
7      pthread_cond_signal(&data_ready);
8      pthread_mutex_unlock(&lock);
9  }
10 Consume() {
11     pthread_mutex_lock(&lock);
12     while (buffer.empty()) {
13         pthread_cond_wait(&data_ready, &lock);
14     }
15     item = buffer.dequeue();
16     pthread_mutex_unlock(&lock);
17     return item;
18 }
19
```

unbounded p/c — lock

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  Queue buffer;
4  Produce(item) {
5      pthread_mutex_lock(&lock);
6      buffer.enqueue(item);
7      pthread_cond_signal(&data_ready);
8      pthread_mutex_unlock(&lock);
9  }
10 Consume() {
11     pthread_mutex_lock(&lock);
12     while (buffer.empty()) {
13         pthread_cond_wait(&data_ready, &lock);
14     }
15     item = buffer.dequeue();
16     pthread_mutex_unlock(&lock);
17     return item;
18 }
19
```

rule: never touch `buffer` without acquiring lock

otherwise: what if two threads simulatenously en/dequeue?

unbounded p/c — dequeue if empty

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  Queue buffer;
4  Produce(item) {
5      pthread_mutex_lock(&lock);
6      buffer.enqueue(item);
7      pthread_cond_signal(&data_ready);
8      pthread_mutex_unlock(&lock);
9  }
10 Consume() {
11     pthread_mutex_lock(&lock);
12     while (buffer.empty()) {
13         pthread_cond_wait(&data_ready, &lock);
14     }
15     item = buffer.dequeue();
16     pthread_mutex_unlock(&lock);
17     return item;
18 }
19
```

check if empty, then dequeue

(note: have lock — don't need to worry about someone else dequeuing)

unbounded p/c — wake

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  Queue buffer;
4  Produce(item) {
5      pthread_mutex_lock(&lock);
6      buffer.enqueue(item);
7      pthread_cond_signal(&data_ready);
8      pthread_mutex_unlock(&lock);
9  }
10 Consume() {
11     pthread_mutex_lock(&lock);
12     while (buffer.empty()) {
13         pthread_cond_wait(&data_ready, &lock);
14     }
15     item = buffer.dequeue();
16     pthread_mutex_unlock(&lock);
17     return item;
18 }
19
```

check if empty (waiting if needed), then dequeue

(note: have lock — don't need to worry about someone else dequeuing)

unbounded p/c — wake

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  Queue buffer;
4  Produce(item) {
5      pthread_mutex_lock(&lock);
6      buffer.enqueue(item);
7      pthread_cond_signal(&data_ready);
8      pthread_mutex_unlock(&lock);
9  }
10 Consume() {
11     pthread_mutex_lock(&lock);
12     while (buffer.empty()) {
13         pthread_cond_wait(&data_ready, &lock);
14     }
15     item = buffer.dequeue();
16     pthread_mutex_unlock(&lock);
17     return item;
18 }
19
```

after enqueuing
wake one Consume thread (if any are waiting)

unbounded p/c — waiting

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  Queue buffer;
4  Produce(item) {
5      pthread_mutex_lock(&lock);
6      buffer.enqueue(item);
7      pthread_cond_signal(&data_ready);
8      pthread_mutex_unlock(&lock);
9  }
10 Consume() {
11     pthread_mutex_lock(&lock);
12     while (buffer.empty()) {
13         pthread_cond_wait(&data_ready, &lock);
14     }
15     item = buffer.dequeue();
16     pthread_mutex_unlock(&lock);
17     return item;
18 }
19
```

unbounded p/c — waiting

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  Queue buffer;
4  Produce(item) {
5      pthread_mutex_lock(&lock);
6      buffer.enqueue(item);
7      pthread_cond_signal(&data_ready);
8      pthread_mutex_unlock(&lock);
9  }
10 Consume() {
11     pthread_mutex_lock(&lock);
12     while (buffer.empty()) {
13         pthread_cond_wait(&data_ready, &lock);
14     }
15     item = buffer.dequeue();
16     pthread_mutex_unlock(&lock);
17     return item;
18 }
19
```

while loop could run 0 times

Thread 1	Thread 2
Produce() … lock … enqueue … signal … unlock	
	Consume? … lock … empty? no … dequeue … unlock return

unbounded p/c — waiting

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  Queue buffer;
4  Produce(item) {
5      pthread_mutex_lock(&lock);
6      buffer.enqueue(item);
7      pthread_cond_signal(&data_ready);
8      pthread_mutex_unlock(&lock);
9  }
10 Consume() {
11     pthread_mutex_lock(&lock);
12     while (buffer.empty()) {
13         pthread_cond_wait(&data_ready, &lock);
14     }
15     item = buffer.dequeue();
16     pthread_mutex_unlock(&lock);
17     return item;
18 }
19
```

while loop could run 1 times

Thread 1	Thread 2
	Consume?
	… lock
	… empty? yes
	… unlock/start wait
Produce()	
… lock	
… enqueue	
… signal	stop wait
… unlock	lock
	… empty? no
	… dequeue
	… unlock
	return

unbounded p/c — waiting

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  Queue buffer;
4  Produce(item) {
5      pthread_mutex_lock(&lock);
6      buffer.enqueue(item);
7      pthread_cond_signal(&data_ready);
8      pthread_mutex_unlock(&lock);
9  }
10 Consume() {
11     pthread_mutex_lock(&lock);
12     while (buffer.empty()) {
13         pthread_cond_wait(&data_ready, &lock);
14     }
15     item = buffer.dequeue();
16     pthread_mutex_unlock(&lock);
17     return item;
18 }
19
```

while loop could run 2+ times

Thread 1	Thread 2	Thread 3
	Consume? … lock … empty? yes … unlock/start wait	
Produce() … lock … enqueue … signal … unlock (waiting for lock)	 stop wait lock	
		Consume? … lock … empty? no … dequeue … unlock return
	… empty? no … dequeue … unlock return	

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 — full code

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  pthread_cond_t space_ready;
4  BoundedQueue buffer;
5  Produce(item) {
6      pthread_mutex_lock(&lock);
7      while (buffer.full()) {
8          pthread_cond_wait(&space_ready, &lock);
9      }
10     buffer.enqueue(item);
11     pthread_cond_signal(&data_ready);
12     pthread_mutex_unlock(&lock);
13 }
14 Consume() {
15     pthread_mutex_lock(&lock);
16     while (buffer.empty()) {
17         pthread_cond_wait(&data_ready, &lock);
18     }
19     item = buffer.dequeue();
20     pthread_cond_signal(&space_ready);
21     pthread_mutex_unlock(&lock);
22     return item;
23 }
24
```

bounded buffer p/c— added

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  pthread_cond_t space_ready;
4  BoundedQueue buffer;
5  Produce(item) {
6      pthread_mutex_lock(&lock);
7      while (buffer.full()) {
8          pthread_cond_wait(&space_ready, &lock);
9      }
10     buffer.enqueue(item);
11     pthread_cond_signal(&data_ready);
12     pthread_mutex_unlock(&lock);
13 }
14 Consume() {
15     pthread_mutex_lock(&lock);
16     while (buffer.empty()) {
17         pthread_cond_wait(&data_ready, &lock);
18     }
19     item = buffer.dequeue();
20     pthread_cond_signal(&space_ready);
21     pthread_mutex_unlock(&lock);
22     return item;
23 }
24
```

bounded buffer p/c— added

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  pthread_cond_t space_ready;
4  BoundedQueue buffer;
5  Produce(item) {
6      pthread_mutex_lock(&lock);
7      while (buffer.full()) {
8          pthread_cond_wait(&space_ready, &lock);
9      }
10     buffer.enqueue(item);
11     pthread_cond_signal(&data_ready);
12     pthread_mutex_unlock(&lock);
13 }
14 Consume() {
15     pthread_mutex_lock(&lock);
16     while (buffer.empty()) {
17         pthread_cond_wait(&data_ready, &lock);
18     }
19     item = buffer.dequeue();
20     pthread_cond_signal(&space_ready);
21     pthread_mutex_unlock(&lock);
22     return item;
23 }
24
```

add new condition variable
for new reason to wait

wait on that condition
while reason to wait is true

signal that condition
when one thread can stop waiting

bounded buffer p/c — signal/broadcast

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  pthread_cond_t space_ready;
4  BoundedQueue buffer;
5  Produce(item) {
6      pthread_mutex_lock(&lock);
7      while (buffer.full()) {
8          pthread_cond_wait(&space_ready, &lock);
9      }
10     buffer.enqueue(item);
11     pthread_cond_signal(&data_ready);
12     pthread_mutex_unlock(&lock);
13 }
14 Consume() {
15     pthread_mutex_lock(&lock);
16     while (buffer.empty()) {
17         pthread_cond_wait(&data_ready, &lock);
18     }
19     item = buffer.dequeue();
20     pthread_cond_signal(&space_ready);
21     pthread_mutex_unlock(&lock);
22     return item;
23 }
24
```

bounded buffer p/c — signal/broadcast

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  pthread_cond_t sp
4  BoundedQueue buff
5  Produce(item) {
6      pthread_mutex
7      while (buffer
8          pthread_c
9      }
10     buffer.enqueu
11     pthread_cond_signal(&data_ready);
12     pthread_mutex_unlock(&lock);
13 }
14 Consume() {
15     pthread_mutex_lock(&lock);
16     while (buffer.empty()) {
17         pthread_cond_wait(&data_ready, &lock);
18     }
19     item = buffer.dequeue();
20     pthread_cond_signal(&space_ready);
21     pthread_mutex_unlock(&lock);
22     return item;
23 }
24
```

`signal` better than `broadcast` in this case

only one waiting thread can do something useful

correct (but slow) to `broadcast` instead

bounded buffer p/c — signal/broadcast

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  pthread_cond_t space_ready;
4  BoundedQueue buffer;
5  Produce(item) {
6      pthread_mutex_lock(&lock);
7      while (buffer.full()) {
8          pthread_cond_wait(&space_ready, &lock);
9      }
10     buffer.enqueue(item);
11     pthread_cond_signal(&data_ready);
12     pthread_mutex_unlock(&lock);
13 }
14 Consume() {
15     pthread_mutex_lock(&lock);
16     while (buffer.empty()) {
17         pthread_cond_wait(&data_ready, &lock);
18     }
19     item = buffer.dequeue();
20     pthread_cond_signal(&space_ready);
21     pthread_mutex_unlock(&lock);
22     return item;
23 }
24
```

bounded buffer p/c — signal/broadcast

```
1  pthread_mutex_t lock;
2  pthread_cond_t data_ready;
3  pthread_cond_t space_ready;
4  BoundedQueue buffer;
5  Produce(item) {
6      pthread_mutex_lock(&lock);
7      while (buffer.full()) {
8          pthread_cond_wait(&space_ready, &lock);
9      }
10     buffer.enqueue(item);
11     pthread_cond_signal(&data_ready);
12     pthread_mutex_unlock(&lock);
13 }
14 Consume() {
15     pthread_mutex_lock(&lock);
16     while (buffer.empty()) {
17         pthread_cond_wait(&data_ready, &lock);
18     }
19     item = buffer.dequeue();
20     pthread_cond_signal(&space_ready);
21     pthread_mutex_unlock(&lock);
22     return item;
23 }
24
```

correct but slow to use just one condition variable:

- `data_ready`, `space_ready` $\Rightarrow$ `ready`
- always use `broadcast`
  - let each thread figure out if it was supposed to be woken up

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*

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 (0)

```
1  // MISSING: init calls, etc.
2  pthread_mutex_t lock;
3  bool finished[2];
4  pthread_cond_t both_finished_cv;
5
6  void WaitForBothFinished() {
7    pthread_mutex_lock(&lock);
8    while (_______________________________) {
9      pthread_cond_wait(&both_finished_cv, &lock);
10   }
11   pthread_mutex_unlock(&lock);
12 }
13
14 void Finish(int index) {
15   pthread_mutex_lock(&lock);
16   finished[index] = true;
17   ___________________________________
18   pthread_mutex_unlock(&lock);
19 }
20
```

wait for both finished (1)

```
1  // MISSING: init calls, etc.
2  pthread_mutex_t lock;
3  bool finished[2];
4  pthread_cond_t both_finished_cv;
5
6  void WaitForBothFinished() {
7    pthread_mutex_lock(&lock);
8    while (____________________________) {
9      pthread_cond_wait(&both_finished_cv, &lock);
10   }
11   pthread_mutex_unlock(&lock);
12 }
13
14 void Finish(int index) {
15   pthread_mutex_lock(&lock);
16   finished[index] = true;
17   ___________________________________
18   pthread_mutex_unlock(&lock);
19 }
20
```

A. `finished[0] && finished[1]`

B. `finished[0] || finished[1]`

C. `!finished[0] || !finished[1]`

D. `finsihed[0] != finished[1]`

E. something else

wait for both finished (2)

```
1  // MISSING: init calls,
2  pthread_mutex_t lock;
3  bool finished[2];
4  pthread_cond_t both_fin
5
6  void WaitForBothFinishe
7    pthread_mutex_lock(&l
8    while (_____________
9      pthread_cond_wait(&
10   }
11   pthread_mutex_unlock(
12 }
13
14 void Finish(int index) {
15   pthread_mutex_lock(&lock);
16   finished[index] = true;
17   ___________________________________
18   pthread_mutex_unlock(&lock);
19 }
20
```

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: one-use 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);
}
```

monitor exercise: one-use barrier

```
struct BarrierInfo {
    pthread_mutex_t lock;
    int total_threads;  // initially total # of threads
    int number_reached; // initially 0
    pthread_cond_t cv;
};

void BarrierWait(BarrierInfo *b) {
    pthread_mutex_lock(&b->lock);
    ++b->number_reached;
    if (b->number_reached == b->total_threads) {
        pthread_cond_broadcast(&b->cv);
    } else {
        while (b->number_reached < b->total_threads)
            pthread_cond_wait(&b->cv, &b->lock);
    }
    pthread_mutex_unlock(&b->lock);
}
```

Backup slides

why spurious wakeups?

makes implementing condition variables simpler

can be hard to avoid loop in more complicated scenarios

e.g. signal() saying okay to remove item from queue
what if another thread sneaks in and does it first?

- maybe signal() could be redesigned to prevent this somehow?
- … but that's harder to implement

BROKEN: producer/consumer signal

exercise: example why signal here is BROKEN? hint: two consume()+two produce()

```
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()) {X\\tikzmark{empty}X
        pthread_cond_wait(&data_ready, &lock);
    }X\\tikzmark{after loop}X
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

BROKEN: producer/consumer signal

exercise: example why signal here is BROKEN? hint: two consume()+two produce()

```
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()) {X\\tikzmark{empty}X
        pthread_cond_wait(&data_ready, &lock);
    }X\\tikzmark{after loop}X
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

bad case (setup)

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

againframe(PCSignalBadSetup)

bad case

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

againframe(PCSignalBad)

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);
  return Combine(item1, item2);
}
```

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);
  return Combine(item1, item2);
}
```

exercise: WaitForBoth

exercise: wait for both finished

```
pthread_mutex_t lock; pthread_cond_t cv;
bool FirstFinished = false; bool SecondFinished = false;

void FinishFirst() {
    pthread_mutex_lock(&lock);
    FirstFinished = true;
    ____________________  // (1)
    pthread_mutex_unlock(&lock);
}

void FinishSecond() {
    pthread_mutex_lock(&lock);
    SecondFinished = true;
    ____________________ // (1)
    pthread_mutex_unlock(&lock);
}

void WaitForBothFinished() {
    pthread_mutex_lock(&lock);
    ___ ( ___________________________ ) { // (2)
        pthread_cond_wait(&lock, &cv);
    }
    pthread_mutex_unlock(&lock);
}
```

Fill in the blanks.

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)
:::: {.columns}

```
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;
}
```

::::

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t @4space_ready4@;
BoundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(@4&space_ready4@, &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(@4&space_ready4@);]{.fragment fragment-index=2 .custom .myem-only}]{.fragment
fragment-index=3 .custom .myem-only}X\\tikzmark{signal}X
```

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t @4space_ready4@;
BoundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(@4&space_ready4@, &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(@4&space_ready4@);]{.fragment fragment-index=2 .custom .myem-only}]{.fragment
fragment-index=3 .custom .myem-only}X\\tikzmark{signal}X
```

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t @4space_ready4@;
BoundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(@4&space_ready4@, &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(@4&space_ready4@);]{.fragment fragment-index=2 .custom .myem-only}]{.fragment
fragment-index=3 .custom .myem-only}X\\tikzmark{signal}X
```

potential fixes

unconditionally signal

- each consume allows one produce to go
- rely on condition variable knowing if no one is waiting

broadcast if buffer changed from full to not-full

- every thread waiting because it was full could go buffer it becomes full again

explicitly count number of waiting producers — buffer not full and waiter

how could I have avoided this?

question: who might be waiting when condition changes

almost always multiple threads!

if not broadcasting, *explain why each waiting thread gets to go*

my implicit non-explanation: queue will be full again first

- not actually true: can keep consuming before producers go

alternate view: consuming causes what threads to go?

- not just when the buffer was full
- since if I empty the buffer by consuming…