Changelog

16 March 2021 (after lecture): correct solution to monitor ordering example

last time (1)

load/store reordering by compiler, processor

- default behavior: don't worry about other cores/threads
- threading library's sync. constructs: directives + instructions to prevent
- (so locks, pthread_join, etc. work as expected)

rules for accessing shared containers (C++)

false sharing

- processor caches work in *cache blocks*
- cores have to *take turns* modifying a cache block
- lots of overhead if two cores working on same cache block
  - even if no race condition (different parts)

last time (2)

implementing waiting locks (mutexes) with spinlocks

- spinlock protects list of waiting threads + “is mutex locked” boolean
- lock: add self to waiting list (if already locked)
- unlock: remove thread from waiting list (if any)
- required lock integration with scheduler

monitors = mutexes + condition variables + shared state

condition variable = list of waiting threads, with operations:

- Wait(CV, lock): add self to list, start waiting (unlocking lock while waiting)
- Broadcast(CV): wake up all waiting threads
- Signal(CV): wake up one waiting thread

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

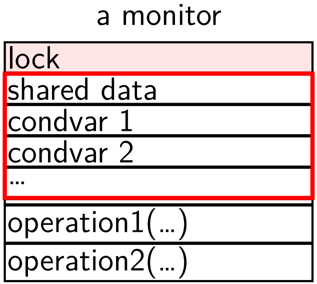
monitor = lock + 0 or more condition variables + shared data

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

monitor idea

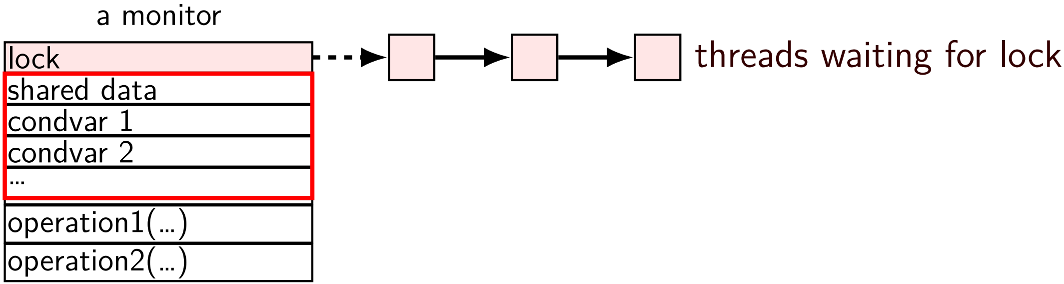
a monitor
lock
shared data
condvar 1
condvar 2
...
operation1(...)
operation2(...)

monitor idea

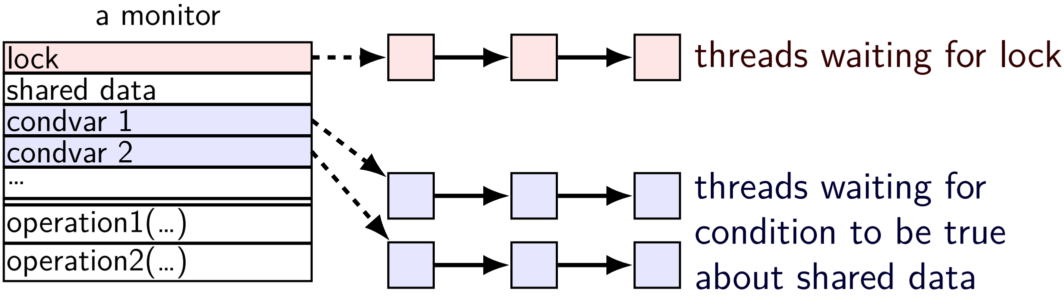


lock must be acquired
before accessing
any part of monitor's stuff

monitor idea



monitor idea



condvar operations

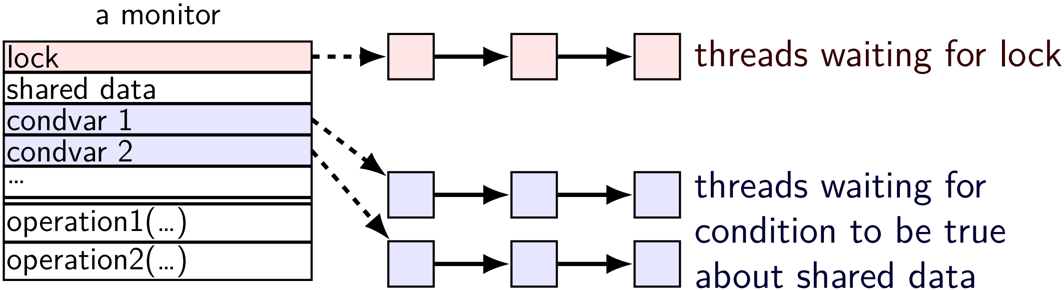
condvar operations:

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

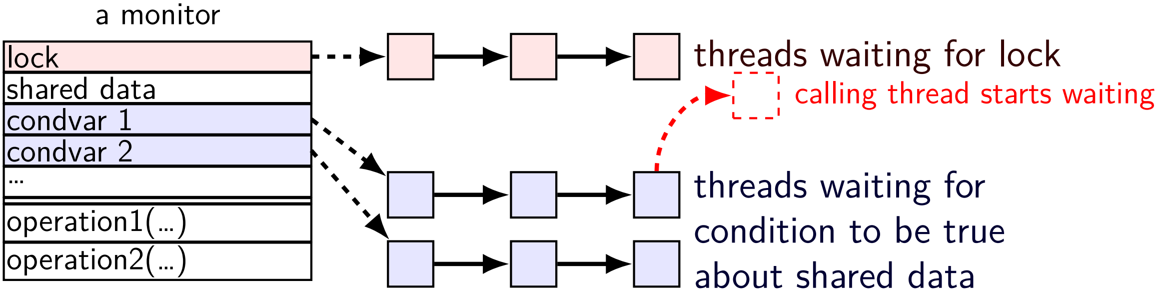
condvar operations:

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

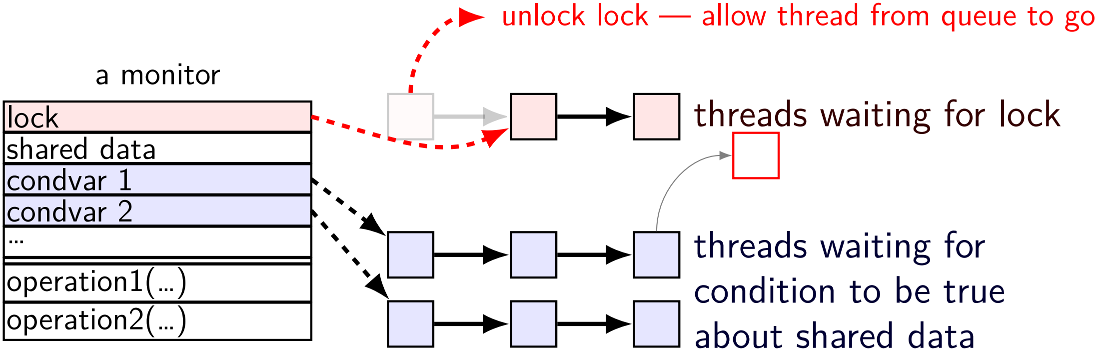
condvar operations:

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

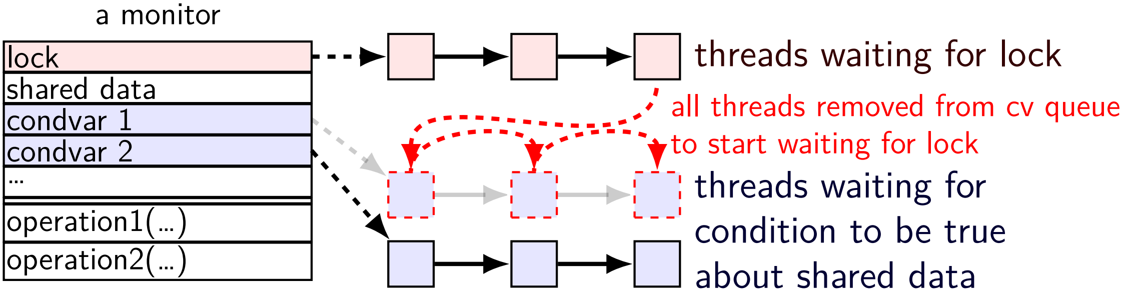
condvar operations:

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

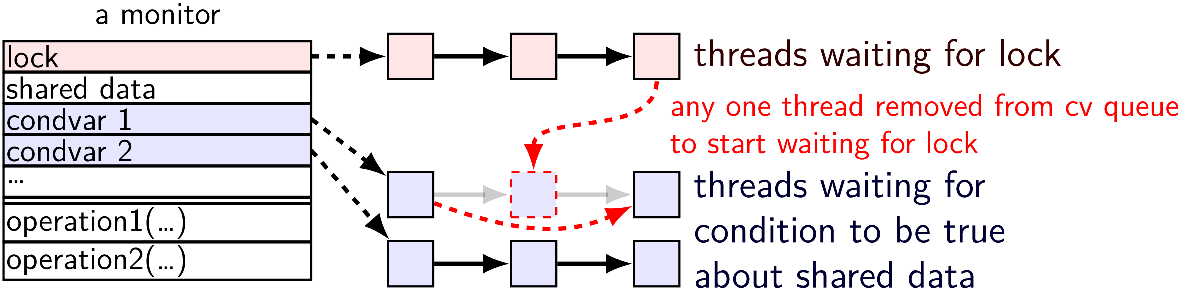
condvar operations:

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

```
// 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);
}

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

acquire lock before reading or writing `finished`

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);
}
```

check whether we need to wait at all

(why a loop? we'll explain later)

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)
`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?

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 simulatenously 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 can**not** 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);
}

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

wake one Consume thread
if any are waiting

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
Produce()	
...lock	
...enqueue	
...signal	
...unlock	
	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_rea
    pthread_mutex_unlock(&lock);
}

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

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

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

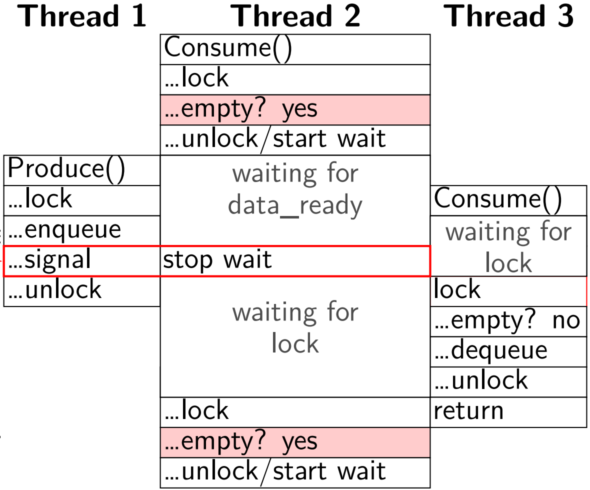
alternate design: signalled thread gets lock next

called "Hoare scheduling"

not done by pthreads, Java, ...

```
);
a_rea
ck);

);
    while (buffer.empty()) {
        pthread_cond_wait(&data_r
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```



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_mutex_unlock(&lock);
}

Consum
    pt
    wh
        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:

`pthread_cond_broadcast(&space_ready);`

(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 B) {
    pthread_cond_broadcast(&condvar_for_B);
    /* or signal, if only one thread cares */
}
if (set condition C) {
    pthread_cond_broadcast(&condvar_for_C);
    /* 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);
```

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);
}
```

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);
}
```

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;
}
```

generalizing locks: semaphores

semaphore has a non-negative integer **value** and two operations:

P() or **down** or **wait**:
wait for semaphore to become positive (> 0),
then decerement by 1

V() or **up** or **signal** or **post**:
increment semaphore by 1 (waking up thread if needed)

P, V from Dutch: *proberen* (test), *verhogen* (increment)

semaphores are kinda integers

semaphore like an integer, but...

cannot read/write directly

- down/up operaion only way to access (typically)
- exception: initialization

never negative — wait instead

- down operation wants to make negative? thread waits

reserving books

suppose tracking copies of library book...

```
Semaphore free_copies = Semaphore(3);
void ReserveBook() {
    // wait for copy to be free
    free_copies.down();
    ... // ... then take reserved copy
}

void ReturnBook() {
    ... // return reserved copy
    free_copies.up();
    // ... then wakeup waiting thread
}
```

counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book

Copy 1
Copy 2
Copy 3

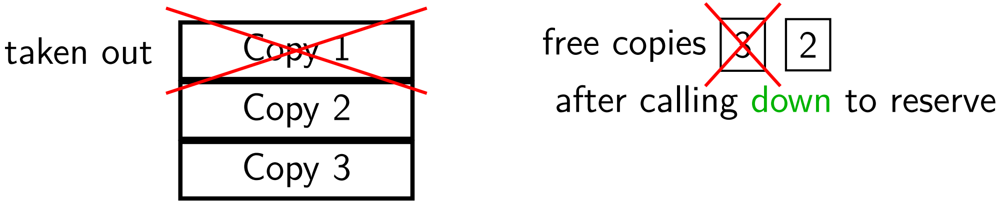
free copies 3

counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book

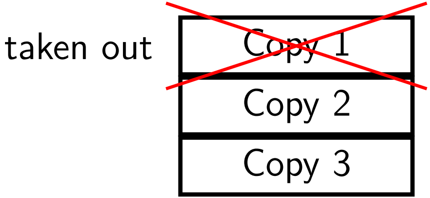


counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book

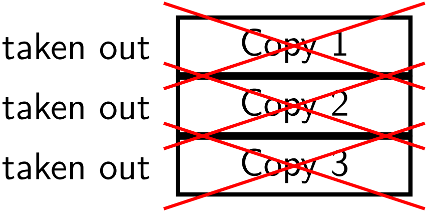


free copies 2

after calling down to reserve

counting resources: reserving books

suppose tracking copies of same library book
non-negative integer count = # how many books used?
up = give back book; down = take book

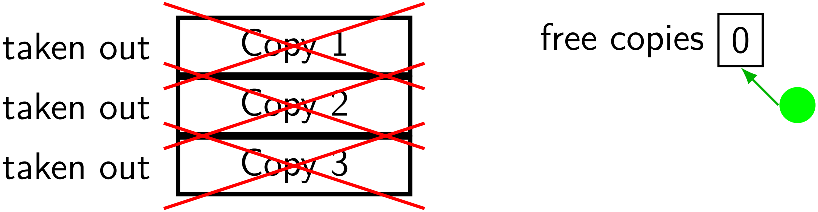


free copies 0

after calling down three times to reserve all copies

counting resources: reserving books

suppose tracking copies of same library book
non-negative integer count = # how many books used?
up = give back book; down = take book



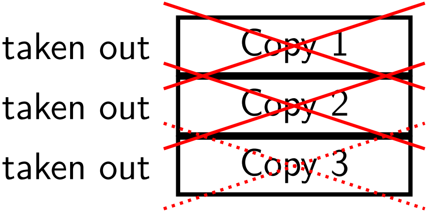
reserve book
call *down* again
start waiting...

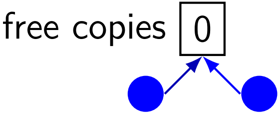
counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book





return book

call *up*

release waiter

reserve book

call *down*

~~waiting~~

done waiting

implementing mutexes with semaphores

```
struct Mutex {
    Semaphore s; /* with inital value 1 */
    /* value = 1 --> mutex if free */
    /* value = 0 --> mutex is busy */
}

MutexLock(Mutex *m) {
    m->s.down();
}

MutexUnlock(Mutex *m) {
    m->s.up();
}
```

implementing join with semaphores

```
struct Thread {
    ...
    Semaphore finish_semaphore; /* with initial value 0 */
    /* value = 0: either thread not finished OR already joined */
    /* value = 1: thread finished AND not joined */
};
thread_join(Thread *t) {
    t->finish_semaphore->down();
}

/* assume called when thread finishes */
thread_exit(Thread *t) {
    t->finish_semaphore->up();
    /* tricky part: deallocating struct Thread safely? */
}
```

POSIX semaphores

```
#include <semaphore.h>
...
sem_t my_semaphore;
int process_shared = /* 1 if sharing between processes */;
sem_init(&my_semaphore, process_shared, initial_value);
...
sem_wait(&my_semaphore);  /* down */
sem_post(&my_semaphore);  /* up */
...
sem_destroy(&my_semaphore);
```

backup slides

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.

semaphores/CV

```
int num_waiting = 0;
bool finished = false;
sem_t mutex; // initially 1
sem_t gate;  // initially 0
void WaitForFinished() {
    sem_wait(&mutex);
    if (finished) {
        sem_post(&mutex);
    } else {
        num_waiting += 1;
        sem_post(&mutex);
        sem_wait(&gate);
    }
}

void Finish() {
    sem_wait(&mutex);
    finished = true;
    while (num_waiting > 0) {
        num_waiting -= 1;
        sem_post(&gate);
    }
}
```

```
bool finished = false;
pthread_mutex_t mutex;
pthread_cond_t cv;


void WaitForFinished() {
    pthread_mutex_lock(&mutex);
    while (!finished) {
        pthread_cond_wait(&cv, &mutex);
    }
    pthread_mutex_unlock(&mutex);
}


void Finish() {
    pthread_mutex_lock(&mutex);
    finished = true;
    pthread_cond_broadcast(&cv);
    pthread_mutex_unlock(&mutex);
}
```

semaphores/CV

```
int num_waiting = 0;
bool finished = false;
sem_t mutex; // initially 1
sem_t gate;  // initially 0
void WaitForFinished() {
    sem_wait(&mutex);
    if (finished) {
        sem_post(&mutex);
    } else {
        num_waiting += 1;
        sem_post(&mutex);
        sem_wait(&gate);
    }
}

void Finish() {
    sem_wait(&mutex);
    finished = true;
    while (num_waiting > 0) {
        num_waiting -= 1;
        sem_post(&gate);
    }
}
```

```
bool finished = false;
pthread_mutex_t mutex;
pthread_cond_t cv;

void WaitForFinished() {
    pthread_mutex_lock(&mutex);
    while (!finished) {
        pthread_cond_wait(&cv, &mutex);
    }
    pthread_mutex_unlock(&mutex);
}

void Finish() {
    pthread_mutex_lock(&mutex);
    finished = true;
    pthread_cond_broadcast(&cv);
    pthread_mutex_unlock(&mutex);
}
```

monitors with semaphores: chosen order

if we want to make sure threads woken up in order

```
ThreadSafeQueue<sem_t> waiters;
Wait(Lock lock) {
  sem_t private_semaphore;
  ... /* init semaphore
         with count 0 */
  waiters.Enqueue(&semaphore);
  lock.Unlock();
  sem_post(private_semaphore);
  lock.Lock();
}
```

```
Signal() {
  sem_t *next = waiters.DequeueOrNull();
  if (next != NULL) {
    sem_post(next);
  }
}
```

monitors with semaphores: chosen order

if we want to make sure threads woken up in order

```
ThreadSafeQueue<sem_t> waiters;
Wait(Lock lock) {
  sem_t private_semaphore;
  ... /* init semaphore
         with count 0 */
  waiters.Enqueue(&semaphore);
  lock.Unlock();
  sem_post(private_semaphore);
  lock.Lock();
}
```

```
Signal() {
  sem_t *next = waiters.DequeueOrNull();
  if (next != NULL) {
    sem_post(next);
  }
}
```

(but now implement queue with semaphores...)

rwlock exercise (1)

suppose there are multiple waiting writers

which one gets waken up first?

- whichever gets signal'd or gets lock first

could instead keep in order they started waiting

exercise: what extra information should we track?

- hint: we might need an array

```
mutex_t lock; cond_t ok_to_read_cv, ok_to_write_cv;
int readers, writers, waiting_writers;
```

rwlock exercise solution?

list of waiting writes?

```
struct WaitingWriter {
    cond_t cv;
    bool ready;
};
Queue<WaitingWriter*> waiting_writers;

WriteLock(...) {
  ...
  if (need to wait) {
    WaitingWriter self;
    self.ready = false;
    ...
    while(!self.ready) {
        pthread_cond_wait(&self.cv, &lock);
    }
  }
  ...
}
```

rwlock exercise solution?

dedicated writing thread with queue

(DoWrite∼Produce; WritingThread∼Consume)

```
ThreadSafeQueue<WritingTask*> waiting_writes;
WritingThread() {
    while (true) {
        WritingTask* task = waiting_writer.Dequeue();
        WriteLock();
        DoWriteTask(task);
        task.done = true;
        cond_broadcast(&task.cv);
    }
}
DoWrite(task) {
    // instead of WriteLock(); DoWriteTask(...); WriteUnlock()
    WritingTask task = ...;
    waiting_writes.Enqueue(&task);
    while (!task.done) { cond_wait(&task.cv); }
}
```

building semaphore with monitors (version B)

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
void down() {
    pthread_mutex_lock(&lock);
    while (!(count > 0)) {
        pthread_cond_wait(
            &count_is_positive_cv,
            &lock);
    }
    count -= 1;
    pthread_mutex_unlock(&lock);
}
```

```
void up() {
    pthread_mutex_lock(&lock);
    count += 1;
    /* condition *just* became true */
    if (count == 1) {
        pthread_cond_broadcast(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

before: signal every time

can check if condition just became true instead?

building semaphore with monitors (version B)

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
void down() {
    pthread_mutex_lock(&lock);
    while (!(count > 0)) {
        pthread_cond_wait(
            &count_is_positive_cv,
            &lock);
    }
    count -= 1;
    pthread_mutex_unlock(&lock);
}
```

```
void up() {
    pthread_mutex_lock(&lock);
    count += 1;
    /* condition *just* became true */
    if (count == 1) {
        pthread_cond_broadcast(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

before: signal every time

can check if condition just became true instead?

but do we really need to broadcast?

exercise: why broadcast?

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
void down() {
    pthread_mutex_lock(&lock);
    while (!(count > 0)) {
        pthread_cond_wait(
            &count_is_positive_cv,
            &lock);
    }
    count -= 1;
    pthread_mutex_unlock(&lock);
}
```

```
void up() {
    pthread_mutex_lock(&lock);
    count += 1;
    if (count == 1) { /* became > 0 */
        pthread_cond_broadcast(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

exercise: why can't this be `pthread_cond_signal`?

hint: think of two threads calling down + two calling up?

brute force: only so many orders they can get the lock in

broadcast problem

Thread 1	Thread 2	Thread 3	Thread 4
Down()			
lock			
count == 0? yes			
unlock/wait			
	Down()		
	lock		
	count == 0? yes		
	unlock/wait		
		Up()	
		lock	
		count += 1 (now 1)	Up()
stop waiting on CV		signal	wait for lock
wait for lock		unlock	wait for lock
wait for lock			lock
wait for lock			count += 1 (now 2)
wait for lock			count != 1: don't signal
lock			unlock
count == 0? no			
count -= 1 (becomes 1)			
unlock			
	still waiting???		

broadcast problem

Thread 1	Thread 2	Thread 3	Thread 4
Down()			
lock			
count == 0? yes			
unlock/wait			
	Down()		
	lock		
	count == 0? yes		
	unlock/wait		
		Up()	
		lock	
		count += 1 (now 1)	Up()
stop waiting on CV		signal	wait for lock
wait for lock		unlock	wait for lock
wait for lock			lock
wait for lock			count += 1 (now 2)
wait for lock			count != 1: don't signal
lock			unlock
count == 0? no			
count -= 1 (becomes 1)			
unlock			
	still waiting???		

broadcast problem

Thread 1	Thread 2	Thread 3	Thread 4
Down()			
lock			
count == 0? yes			
unlock/wait			
	Down()		
	lock		
	count == 0? yes		
	unlock/wait		
		Up()	
		lock	
		count += 1 (now 1)	Up()
stop waiting on CV		signal	wait for lock
wait for lock		unlock	wait for lock
wait for lock			lock
wait for lock			count += 1 (now 2)
wait for lock			count != 1: don't signal
lock			unlock
count == 0? no			
count -= 1 (becomes 1)			
unlock			
	still waiting???		

Mesa-style monitors
signalling doesn't
"hand off" lock