last time

implementing locks without much busy-waiting

- spinlocks to prevent concurrent modification to list of waiting threads
- lock(): can't get now? set self not runnable + context switch
- unlock(): someone waiting? set them runnable

condition variables (CVs) $\sim$ list of waiting threads

- one list (CV) for each things threads might wait for
- Wait(cv, lock): unlock lock + start waiting; relock when done waiting
- Signal(cv)/Broadcast(cv): wake-up one/all waiting threads

monitor = lock + shared data + condition variables

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_fin

void WaitForBothFinished
  pthread_mutex_lock(&lo
  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_singal(&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);
}
```

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;
}
```

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);
    return item;
}
```

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	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
gets lock			
dequeue			
	still waiting		

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...

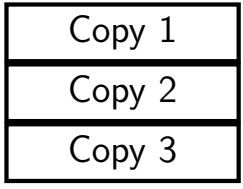
```
Semaphore free_copies = Semaphore(3);
void ReserveBook() {
    // wait for copy to be free
    free_copies.down();
    ... // ... then take reserved copy
}

void ReturnBook() {
    ... // return reserved copy
    free_copies.up();
    // ... then wakekup 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



free copies 3

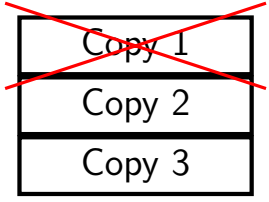
counting resources: reserving books

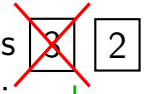
suppose tracking copies of same library book

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

up = give back book; down = take book

taken out



free copies 

after calling down to reserve

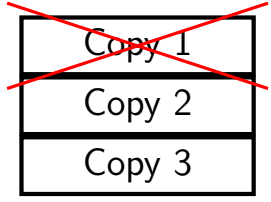
counting resources: reserving books

suppose tracking copies of same library book

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

up = give back book; down = take book

taken out



free copies 2

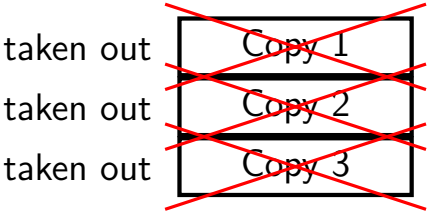
after calling down to reserve

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

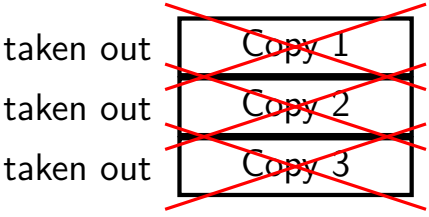
after calling down three times
to reserve all copies

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



free copies 0

reserve book

call *down* again

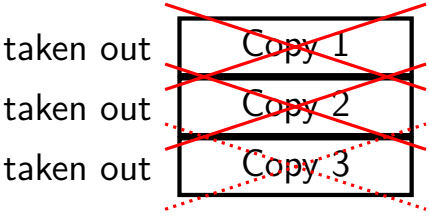
start waiting...

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



free copies 0

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);
```

semaphore exercise

```
int value;  sem_t empty, ready;  // with some initial values

void PutValue(int argument) {
    sem_wait(&empty);
    value = argument;
    sem_post(&ready);
}

int GetValue() {
    int result;
    ________________
    result = value;
    ________________
    return result;
}
```

What goes in the blanks?

A: `sem_post(&empty)` / `sem_wait(&ready)`
B: `sem_wait(&ready)` / `sem_post(&empty)`
C: `sem_post(&ready)` / `sem_wait(&empty)`
D: `sem_post(&ready)` / `sem_post(&empty)`
E: `sem_wait(&empty)` / `sem_post(&ready)`
F: something else

GetValue() waits for PutValue() to happen, retrieves value, then allows next PutValue().

PutValue() waits for prior GetValue(), places value, then allows next GetValue().

semaphore exercise [solution]

```
int value;
sem_t empty, ready;
void PutValue(int argument) {
    sem_wait(&empty);
    value = argument;
    sem_post(&ready);
}
int GetValue() {
    int result;
    sem_wait(&ready);
    result = value;
    sem_post(&empty);
    return result;
}
```

semaphore intuition

What do you need to wait for?

- critical section to be finished
- queue to be non-empty
- array to have space for new items

what can you count that will be 0 when you need to wait?

- # of threads that can start critical section now
- # of threads that can join another thread without waiting
- # of items in queue
- # of empty spaces in array

use up/down operations to maintain count

producer/consumer constraints

consumer waits for producer(s) if buffer is empty

producer waits for consumer(s) if buffer is full

any thread waits while a thread is manipulating the buffer

producer/consumer constraints

consumer waits for producer(s) if buffer is empty

producer waits for consumer(s) if buffer is full

any thread waits while a thread is manipulating the buffer

one semaphore per constraint:

```
sem_t full_slots;   // consumer waits if empty
sem_t empty_slots;  // producer waits if full
sem_t mutex;        // either waits if anyone changing buffer
FixedSizedQueue buffer;
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots);  // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots);  // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots);  // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);                                  re data
}

Consume() {
    sem_wait(&full_slots);  // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

Can we do
```
sem_wait(&mutex);
sem_wait(&empty_slots);
```
instead?

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots); // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);
}

Consume() {
    sem_wait(&full_slots);
    sem_wait(&mutex);
    item = buffer.dequeue()
    sem_post(&mutex);
    sem_post(&empty_slots);
    return item;
}
```

Can we do

```
sem_wait(&mutex);
sem_wait(&empty_slots);
```

instead?

No. Consumer waits on `sem_wait(&mutex)` so can't `sem_post(&empty_slots)` (result: producer waits forever problem called *deadlock*)

producer/consumer: cannot reorder mutex/empty

```
ProducerReordered() {
  // BROKEN: WRONG ORDER
  sem_wait(&mutex);
  sem_wait(&empty_slots);

  ...

  sem_post(&mutex);
```

```
Consumer() {
  sem_wait(&full_slots);

  // can't finish until
  // Producer's sem_post(&mutex):
  sem_wait(&mutex);

  ...

  // so this is not reached
  sem_post(&full_slots);
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots                                    s more data
}

Consume() {
    sem_wait(&full_slots                                    , reserve it
    sem_wait(&mutex);
    item = buffer.dequeu
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

Can we do

```
sem_post(&full_slots);
sem_post(&mutex);
```

instead?

Yes — post never waits

producer/consumer summary

producer: wait (down) empty_slots, post (up) full_slots

consumer: wait (down) full_slots, post (up) empty_slots

two producers or consumers?

- still works!

gate intuition/pattern

pattern to allow one thread at a time:

```
sem_t gate; // 0 = closed; 1 = open
ReleasingThread() {
    ... // finish what the other thread is waiting for
    while (another thread is waiting and can go) {
        sem_post(&gate)  // allow EXACTLY ONE thread
        ... // other bookkeeping
    }
    ...
}
WaitingThread() {
    ... // indicate that we're waiting
    sem_wait(&gate) // wait for gate to be open
    ... // indicate that we're not waiting
}
```

Anderson-Dahlin and semaphores

Anderson/Dahlin complains about semaphores

> "Our view is that programming with locks and condition variables is superior to programming with semaphores."

argument 1: clearer to have separate constructs for

> waiting for condition to be come true, and
> allowing only one thread to manipulate a thing at a time

arugment 2: tricky to verify thread calls up exactly once for every down

> alternatives allow one to be sloppier (in a sense)

backup slides

backup slides

binary semaphores

binary semaphores — semaphores that are only zero or one

as powerful as normal semaphores

- exercise: simulate counting semaphores with binary semaphores (more than one) and an integer

monitors with semaphores: locks

```
sem_t semaphore;  // initial value 1

Lock() {
    sem_wait(&semaphore);
}

Unlock() {
    sem_post(&semaphore);
}
```

monitors with semaphores: [broken] cvs

start with only wait/signal:

```
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
    lock.Unlock();
    sem_wait(&threads_to_wakeup);
    lock.Lock();
}
Signal() {
    sem_post(&threads_to_wakeup);
}
```

monitors with semaphores: [broken] cvs

start with only wait/signal:

```
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
    lock.Unlock();
    sem_wait(&threads_to_wakeup);
    lock.Lock();
}
Signal() {
    sem_post(&threads_to_wakeup);
}
```

problem: signal wakes up non-waiting threads (in the far future)

monitors with semaphores: cvs (better)

start with only wait/signal:

```
sem_t private_lock;  // initially 1
int num_waiters;
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
  sem_wait(&private_lock);
  ++num_waiters;
  sem_post(&private_lock);
  lock.Unlock();
  sem_wait(&threads_to_wakeup);
  lock.Lock();
}
```

```
Signal() {
  sem_wait(&private_lock);
  if (num_waiters > 0) {
    sem_post(&threads_to_wakeup);
    --num_waiters;
  }
  sem_post(&private_lock);
}
```

monitors with semaphores: broadcast

now allows broadcast:

```
sem_t private_lock;  // initially 1
int num_waiters;
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
  sem_wait(&private_lock);
  ++num_waiters;
  sem_post(&private_lock);
  lock.Unlock();
  sem_wait(&threads_to_wakeup);
  lock.Lock();
}
```

```
Broadcast() {
  sem_wait(&private_lock);
  while (num_waiters > 0) {
    sem_post(&threads_to_wakeup);
    --num_waiters;
  }
  sem_post(&private_lock);
}
```

building semaphore with monitors

```
pthread_mutex_t lock;
```

lock to protect shared state

building semaphore with monitors

```
pthread_mutex_t lock;
unsigned int count;
```

lock to protect shared state

shared state: semaphore tracks a count

building semaphore with monitors

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
```

lock to protect shared state

shared state: semaphore tracks a count

add cond var for each reason we wait

semaphore: wait for count to become positive (for down)

building semaphore with monitors

```
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);
}
```

lock to protect shared state

shared state: semaphore tracks a count

add cond var for each reason we wait

semaphore: wait for count to become positive (for down)

wait using condvar; broadcast/signal when condition changes

building semaphore with monitors

```
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;
    /* count must now be
       positive, and at most
       one thread can go per
       call to Up() */
    pthread_cond_signal(
        &count_is_positive_cv
    );
    pthread_mutex_unlock(&lock);
}
```

lock to protect shared state

shared state: semaphore tracks a count

add cond var for each reason we wait

semaphore: wait for count to become positive (for down)

wait using condvar; broadcast/signal when condition changes

semaphores with monitors: no condition

```
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;
    pthread_cond_signal(
        &count_is_positive_cv
    );
    pthread_mutex_unlock(&lock);
}
```

same as where we started...

semaphores with monitors: alt w/ signal

```
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;
    if (count > 0) {
        pthread_cond_signal(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

```
void up() {
    pthread_mutex_lock(&lock);
    count += 1;
    if (count == 1) {
        pthread_cond_signal(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

on signal/broadcast generally

whenever using signal need to ask
what if more than one thread is waiting?

need to explain why those threads will be signalled eventually

...even if next thread signalled doesn't run right away

another problem that would be avoided with Hoare scheduling

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