sync-semaphores

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...

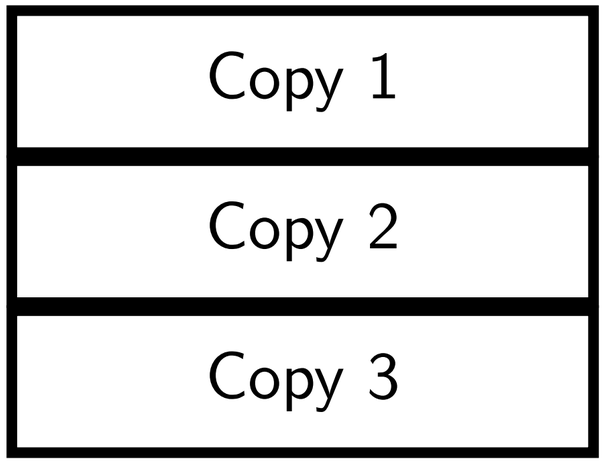
```
Semaphore free_copies = Semaphore(3);
void ReserveBook() {
    // wait for copy to be free
    free_copies.down();
    ... // ... then take reserved copy
}

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

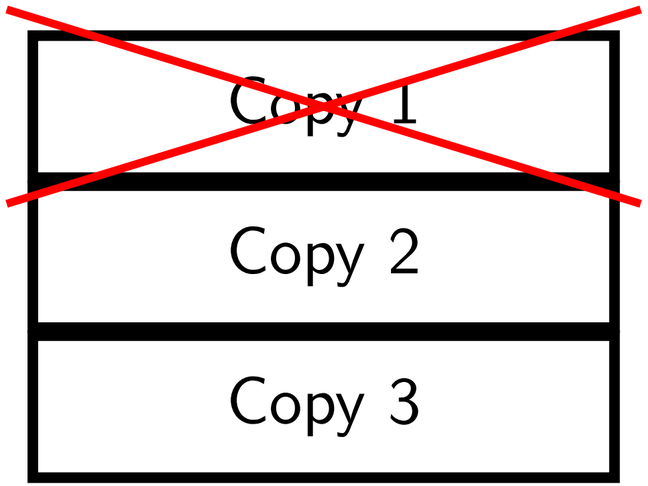
counting resources: reserving books

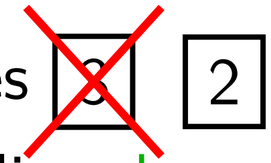
suppose tracking copies of same library book

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

up = give back book; down = take book

taken out



free copies  2

after calling down to reserve

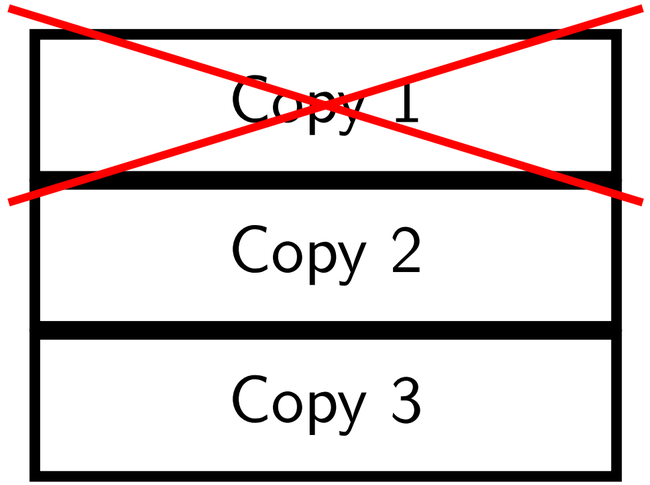
counting resources: reserving books

suppose tracking copies of same library book

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

up = give back book; down = take book

taken out

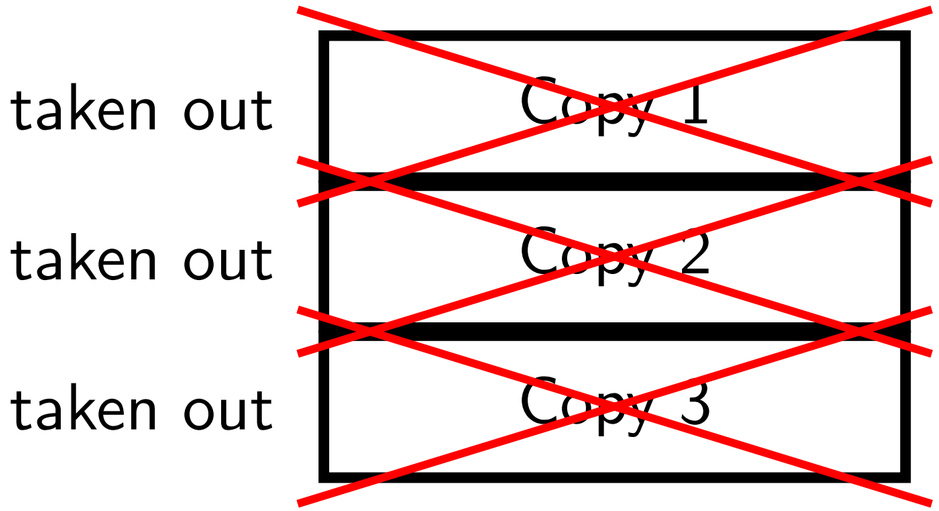


free copies 2

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

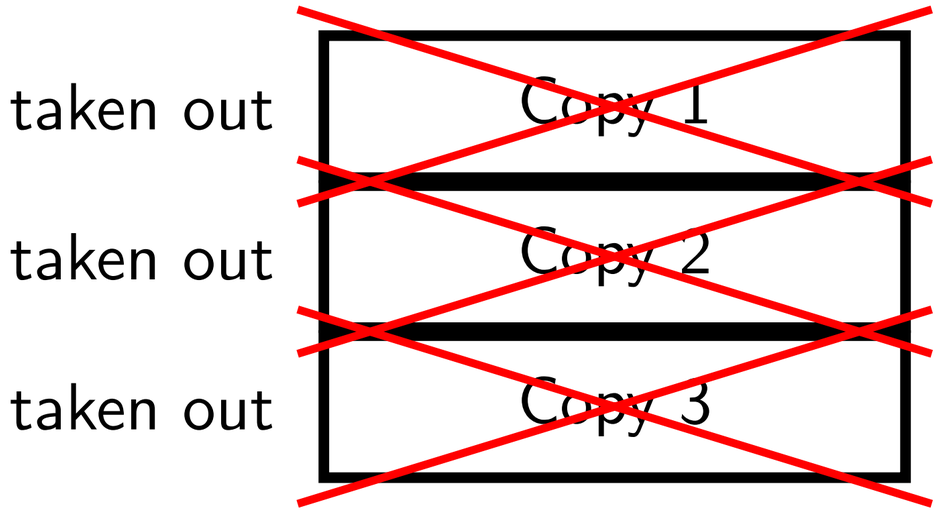


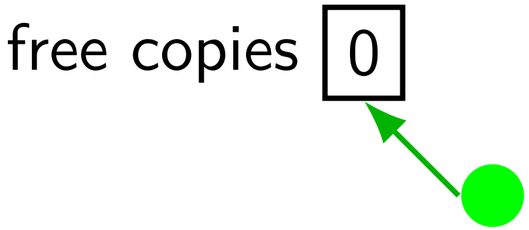
free copies 0

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

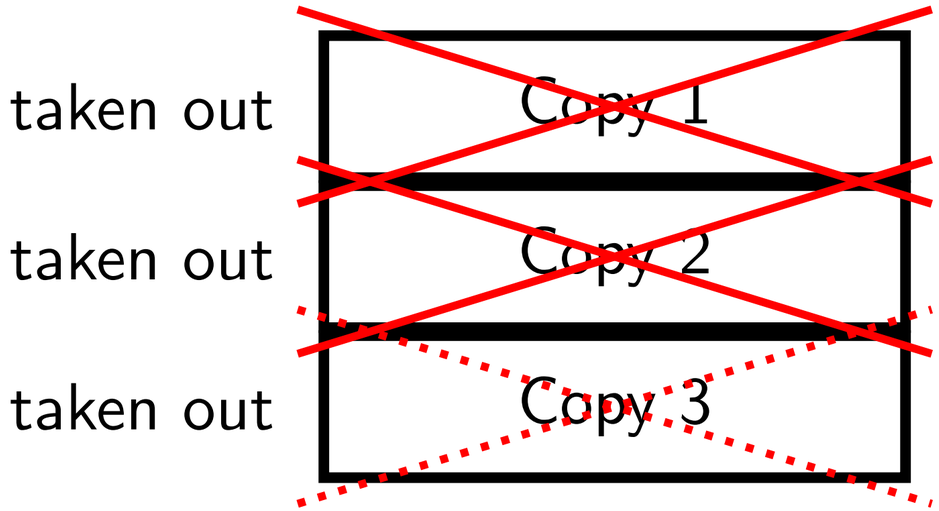


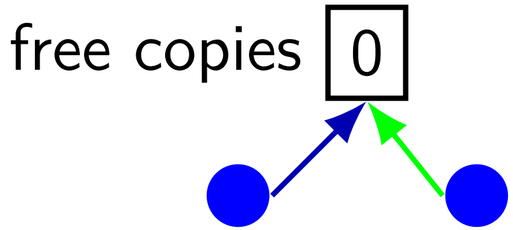


reserve book
call *down* again
start waiting...

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);
```

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;
}
```

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

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

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

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

```
1  sem_init(&full_slots, ..., 0);
2  sem_init(&empty_slots, ..., BUFFER_CAPACITY);
3  sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
4  buffer.set_size(BUFFER_CAPACITY);
5  ...
6  Produce(item) {
7      sem_wait(&empty_slots);
8      sem_wait(&mutex);
9      buffer.enqueue(item);
10     sem_post(&mutex);
11     // tell consumers there is more data
12     sem_post(&full_slots);
13 }
14
15 Consume() {
16     // wait until queued item, reserve it
17     sem_wait(&full_slots);
18     sem_wait(&mutex);
19     item = buffer.dequeue();
20     sem_post(&mutex);
21     // let producer reuse item slot
22     sem_post(&empty_slots);
23     return item;
24 }
25
```

producer/consumer psuedocode — semaphores

```
1  sem_init(&full_slots, ..., 0);
2  sem_init(&empty_slots, ..., BUFFER_CAPACITY);
3  sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
4  buffer.set_size(BUFFER_CAPACITY);
5  ...
6  Produce(item) {
7      sem_wait(&empty_slots);
8      sem_wait(&mutex);
9      buffer.enqueue(item);
10     sem_post(&mutex);
11     // tell consumers there is more data
12     sem_post(&full_slots);
13 }
14
15 Consume() {
16     // wait until queued item, reserve it
17     sem_wait(&full_slots);
18     sem_wait(&mutex);
19     item = buffer.dequeue();
20     sem_post(&mutex);
21     // let producer reuse item slot
22     sem_post(&empty_slots);
23     return item;
24 }
25
```

`full_slots` $\leq$ # items on queue

`empty_slots` $\leq$ # free slots on queue

producer/consumer psuedocode — exercise 1

```
1  sem_init(&full_slots, ..., 0);
2  sem_init(&empty_slots, ..., BUFFER_CAPACITY);
3  sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
4  buffer.set_size(BUFFER_CAPACITY);
5  ...
6  Produce(item) {
7      sem_wait(&empty_slots);
8      sem_wait(&mutex);
9      buffer.enqueue(item);
10     sem_post(&mutex);
11     // tell consumers there is more data
12     sem_post(&full_slots);
13 }
14
15 Consume() {
16     // wait until queued item, reserve it
17     sem_wait(&full_slots);
18     sem_wait(&mutex);
19     item = buffer.dequeue();
20     sem_post(&mutex);
21     // let producer reuse item slot
22     sem_post(&empty_slots);
23     return item;
24 }
25
```

exercise: when is `full_slots` value + `empty_slots` value != queue size?

producer/consumer pseudocode — exercise 2

```
1  sem_init(&full_slots, ..., 0);
2  sem_init(&empty_slots, ..., BUFFER_CAPACITY);
3  sem_init(&mutex, ..., 1 /* # thread that can
4  buffer.set_size(BUFFER_CAPACITY);
5  ...
6  Produce(item) {
7      sem_wait(&empty_slots);
8      sem_wait(&mutex);
9      buffer.enqueue(item);
10     sem_post(&mutex);
11     // tell consumers there is more data
12     sem_post(&full_slots);
13 }
14
15 Consume() {
16     // wait until queued item, reserve it
17     sem_wait(&full_slots);
18     sem_wait(&mutex);
19     item = buffer.dequeue();
20     sem_post(&mutex);
21     // let producer reuse item slot
22     sem_post(&empty_slots);
23     return item;
24 }
25
```

can we do

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

instead in Produce()?

producer/consumer pseudocode — exercise 2

```
1  sem_init(&full_slots, ..., 0);
2  sem_init(&empty_slots, ..., BUFFER_CAPACITY);
3  sem_init(&mutex, ..., 1 /* # thread that can
4  buffer.set_size(BUFFER_CAPACITY);
5  ...
6  Produce(item) {
7      sem_wait(&empty_slots);
8      sem_wait(&mutex);
9      buffer.enqueue(item);
10     sem_post(&mutex);
11     // tell consumers there is more data
12     sem_post(&full_slots);
13 }
14
15 Consume() {
16     // wait until queued item, reserve it
17     sem_wait(&full_slots);
18     sem_wait(&mutex);
19     item = buffer.dequeue();
20     sem_post(&mutex);
21     // let producer reuse item slot
22     sem_post(&empty_slots);
23     return item;
24 }
25
```

can we do

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

instead in Produce()?

No Consumer waits on `sem_wait(&mutex)`

so can't `sem_wait(&empty_slots)`

(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 psuedocode — exercise 3

```
1  sem_init(&full_slots, ..., 0);
2  sem_init(&empty_slots, ..., BUFFER_CAPACITY);
3  sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
4  buffer.set_size(BUFFER_CAPACITY);
5  ...
6  Produce(item) {
7      sem_wait(&empty_slots);
8      sem_wait(&mutex);
9      buffer.enqueue(item);
10     sem_post(&mutex);
11     // tell consumers there is more data
12     sem_post(&full_slots);
13 }
14
15 Consume() {
16     // wait until queued item, reserve it
17     sem_wait(&full_slots);
18     sem_wait(&mutex);
19     item = buffer.dequeue();
20     sem_post(&mutex);
21     // let producer reuse item slot
22     sem_post(&empty_slots);
23     return item;
24 }
25
```

can we do

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

instead in Produce()?

producer/consumer psuedocode — exercise 3

```
1  sem_init(&full_slots, ..., 0);
2  sem_init(&empty_slots, ..., BUFFER_CAPACITY);
3  sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
4  buffer.set_size(BUFFER_CAPACITY);
5  ...
6  Produce(item) {
7      sem_wait(&empty_slots);
8      sem_wait(&mutex);
9      buffer.enqueue(item);
10     sem_post(&mutex);
11     // tell consumers there is more data
12     sem_post(&full_slots);
13 }
14
15 Consume() {
16     // wait until queued item, reserve it
17     sem_wait(&full_slots);
18     sem_wait(&mutex);
19     item = buffer.dequeue();
20     sem_post(&mutex);
21     // let producer reuse item slot
22     sem_post(&empty_slots);
23     return item;
24 }
25
```

can we do

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

instead in Produce()?

Yes

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!

Backup slides

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)

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);
}
```

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

- 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;
```

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;
```

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;
```

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)

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)

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

counting semaphores with binary semaphores (1)

\tiny{ via Hemmendinger, “Comments on ‘A correect and unrestrictive implementation of general semaphores’ ” (1989); Barz, “Implementing semaphores by binary semaphores” (1983)}

```
// assuming initialValue > 0
BinarySemaphore mutex(1);
int value = initialValue ;
BinarySemaphore gate(1 /* if initialValue >= 1 */);
    /* gate = # threads that can Down() now */
```

```
void Down() {
  gate.Down();
  // wait, if needed
  mutex.Down();
  value -= 1;
  if (value > 0) {
    gate.Up();
    // because next down should finish
    // now (but not marked to before)
  }
  mutex.Up();
}
```

```
void Up() {
  mutex.Down();
  value += 1;
  if (value == 1) {
    gate.Up();
    // because down should finish now
    // but could not before
  }
  mutex.Up();
}
```

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
}
```

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;
```

```
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;
```

```
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;
```

```
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);
  }
}
```

(but now implement queue with semaphores…)