





# 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;
}
```

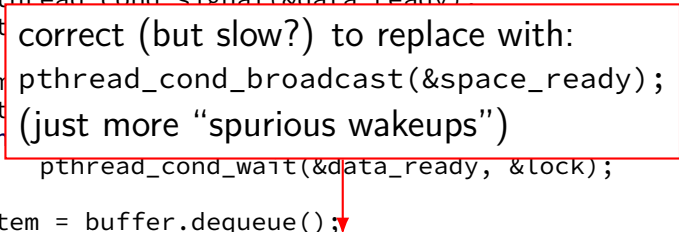
# bounded buffer producer/consumer

```
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);
}
Consume() {
    pthread_cond_broadcast(&space_ready);
    pthread_cond_wait(&data_ready, &lock);
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

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



# bounded buffer producer/consumer

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

correct but slow to replace  
data\_ready and space\_ready  
with 'combined' condvar ready  
and use broadcast  
(just more "spurious wakeups")

# monitor pattern

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

# monitors rules of thumb

never touch shared data without holding the lock

keep lock held for **entire operation**:

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

create condvar for every kind of scenario waited for

always write **loop** calling cond\_wait to wait for condition X

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



# monitors rules of thumb

never touch shared data without holding the lock

keep lock held for **entire operation**:

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

create condvar for every kind of scenario waited for

always write **loop** calling cond\_wait to wait for condition X

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

correct but slow to...

broadcast when just signal would work

broadcast or signal when nothing changed

use one condvar for multiple conditions

# mutex/cond var init/destroy

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

# wait for both finished

```
// MISSING: init calls, etc.
```

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

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

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

# wait for both finished

*// MISSING: init calls, etc.*

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

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

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

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

# wait for both finished

*// MISSING: init calls, etc.*

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

```
void WaitForBothFinished
```

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

```
    while ( _____ )
```

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

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

```
}
```

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

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

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

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

```
}
```

- A. pthread\_cond\_signal(&both\_finished\_cv)
- B. pthread\_cond\_broadcast(&both\_finished\_cv)
- C. if (finished[1-index])  
 pthread\_cond\_signal(&both\_finished\_cv);
- D. if (finished[1-index])  
 pthread\_cond\_broadcast(&both\_finished\_cv);
- E. something else

# 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 decrement 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 operation 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
---

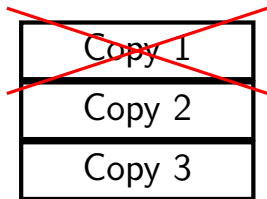
# counting resources: reserving books

suppose tracking copies of same library book

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

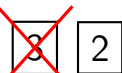
up = give back book; down = take book

taken out



Copy 1
Copy 2
Copy 3

free copies



<del>3</del>	2
--------------	---

after calling down to reserve

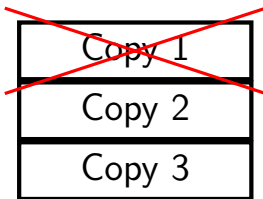
# counting resources: reserving books

suppose tracking copies of same library book

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

up = give back book; down = take book

taken out



Copy 1
Copy 2
Copy 3

free copies 2

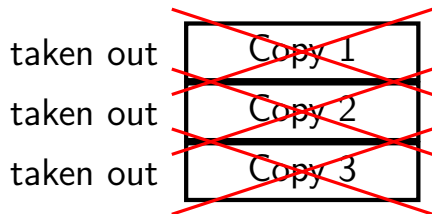
after calling down to reserve

# 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

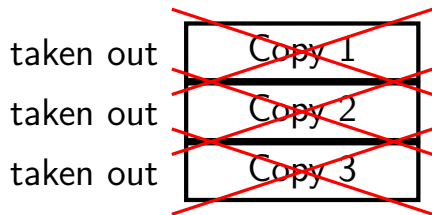
after calling down three times  
to reserve all copies

# 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



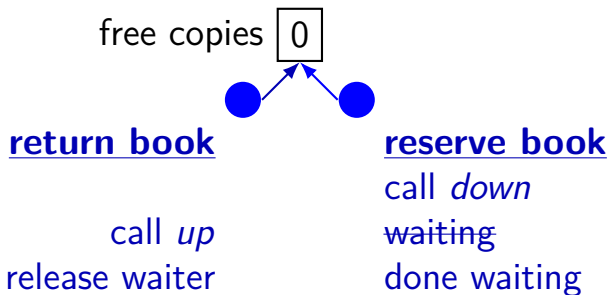
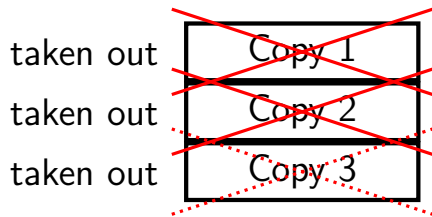
reserve book  
call *down* again  
start waiting...

# 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



# implementing mutexes with semaphores

```
struct Mutex {  
    Semaphore s; /* with initial 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 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);  
}
```

Can we do  
    sem\_wait(&mutex);  
    sem\_wait(&empty\_slots);  
instead? *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);  
}
```

Can we do  
    sem\_wait(&mutex);  
    sem\_wait(&empty\_slots); *data*  
instead?

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

**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);  
}
```

```
Consume() {  
    sem_wait(&full_slots);  
    sem_wait(&mutex);  
    item = buffer.dequeue();  
    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

more data

reserve it

## 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!

# transactions

transaction: set of operations that occurs atomically

idea: something higher-level handles locking, etc.:

```
BeginTransaction();  
int FromOldBalance = GetBalance(FromAccount);  
int ToOldBalance = GetBalance(ToAccount);  
SetBalance(FromAccount, FromOldBalance - 100);  
SetBalance(ToAccount, FromOldBalance + 100);  
EndTransaction();
```

idea: library/database/etc. makes “transaction” happens all at once

# consistency / durability

“happens all at once” = could mean:

locking to make sure no other operations interfere (consistency)

making sure on crash, no partial transaction seen (durability)

(some systems provide both, some provide only one)

we'll just talk about implementing consistency

# implementing consistency: simple

simplest idea: only one run transaction at a time



# implementing consistency: locking

everytime something read/written: acquire associated lock

on end transaction: release lock

if deadlock: undo everything, go back to BeginTransaction(), retry

how to undo?

one idea: keep list of writes instead of writing

apply writes only at EndTransaction()

# implementing consistency: locking

everytime something read/written: acquire associated lock

on end transaction: release lock

if deadlock: **undo everything**, go back to BeginTransaction(), retry

how to undo?

one idea: keep list of writes instead of writing

apply writes only at EndTransaction()

# implementing consistency: optimistic

on read: copy value read

on write: record value to be written, but don't write yet

on end transaction:

- acquire locks on everything

- make sure values read haven't been changed since read

if they have changed, just retry transaction

**backup slides**

## 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);  
}
```

# unbounded buffer producer/consumer

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

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

# unbounded buffer producer/consumer

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

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

rule: never touch buffer  
without acquiring lock

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

# unbounded buffer producer/consumer

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

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

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

check if empty  
if so, dequeue

okay because have lock  
other threads cannot dequeue here



# unbounded buffer producer/consumer

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

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

wake one Consume thread  
*if any are waiting*

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

# unbounded buffer producer/consumer

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

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

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

## Thread 1

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

## Thread 2

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

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

# unbounded buffer producer/consumer

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

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

Thread 1

Thread 2

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

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

# unbounded buffer producer/consumer

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

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

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

Thread 1

Produce()

...lock  
...enqueue  
...signal  
...unlock

Thread 2

Consume()

...lock  
...empty? yes  
...unlock/start wait

Thread 3

Consume()

waiting for lock  
lock  
...empty? no  
...dequeue  
...unlock  
return

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

# unbounded buffer producer/consumer

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

in pthreads: signalled thread not  
gaurenteed to hold lock next

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

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

Thread 1

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

Thread 2

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

Thread 3

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

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

# 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: 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;  
}
```

# 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(&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;  
}
```



## 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  enqueue size = 1? signal unlock	Produce(): wait for lock  gets lock enqueue size $\neq$ 1: don't signal unlock
wait for lock			
gets lock dequeue	still 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 become true, and

allowing only one thread to manipulate a thing at a time

argument 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);
}
```

# 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

# 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

via Hemmendinger, "Comments on 'A correct 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
}
```



## exercise: forwarding paths (2)

cycle # 0 1 2 3 4 5 6 7 8

**addq** %r8, %r9

**subq** %r8, %r9

**ret** (goes to andq)

**andq** %r10, %r9

in subq, %r8 is \_\_\_\_\_ addq.

in subq, %r9 is \_\_\_\_\_ addq.

in andq, %r9 is \_\_\_\_\_ subq.

in andq, %r9 is \_\_\_\_\_ addq.

A: not forwarded from

B-D: forwarded to decode from {execute,memory,writeback} stage of