CONCURRENCY:
QUEUE LOCKS AND CONDITION VARIABLES

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

- Project 4: xv6 Scheduler
  - Due Tuesday at 5:00pm (or midnight)
  - Test Cases available
  - Handin directories available

- Project 5 (xv6 Virtual Memory) available Tuesday
  - Partners strongly recommended

AGENDA / LEARNING OUTCOMES

Concurrency

- How to **block** instead of **spin-wait** while waiting for a lock?
- **When** should a waiting thread block vs. spin?
- How can threads enforce **ordering** across operations (**condition variables**)?
- How can **thread_join()** be implemented?
- How can we support **producer/consumer** apps?

RECAP

LOCK IMPLEMENTATION GOALS

Correctness

- *Mutual exclusion*
  Only one thread in critical section at a time
- *Progress* (deadlock-free)
  If several simultaneous requests, must allow one to proceed
- *Bounded* (starvation-free)
  Must eventually allow each waiting thread to enter

Fairness: Each thread given lock in same order as requested

Performance: CPU is not used unnecessarily

LOCK IMPLEMENTATION WITH XCHG

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    lock->flag = 0;
}

void acquire(lock_t *lock) {
    while(xchg(&lock->flag, 1) == 1) ;
    // spin-wait (do nothing)
}

void release(lock_t *lock) {
    lock->flag = 0;
}
```

FAIRNESS: TICKET LOCKS

Idea: reserve each thread's turn to use a lock.

Each thread spins until their turn.

Use new atomic primitive, fetch-and-add

```
int FetchAndAdd(int *ptr) {
   int old = *ptr;
   *ptr = old + 1;
   return old;
}
```

Acquire: Grab ticket; Spin while not thread's ticket != turn

Release: Advance to next turn

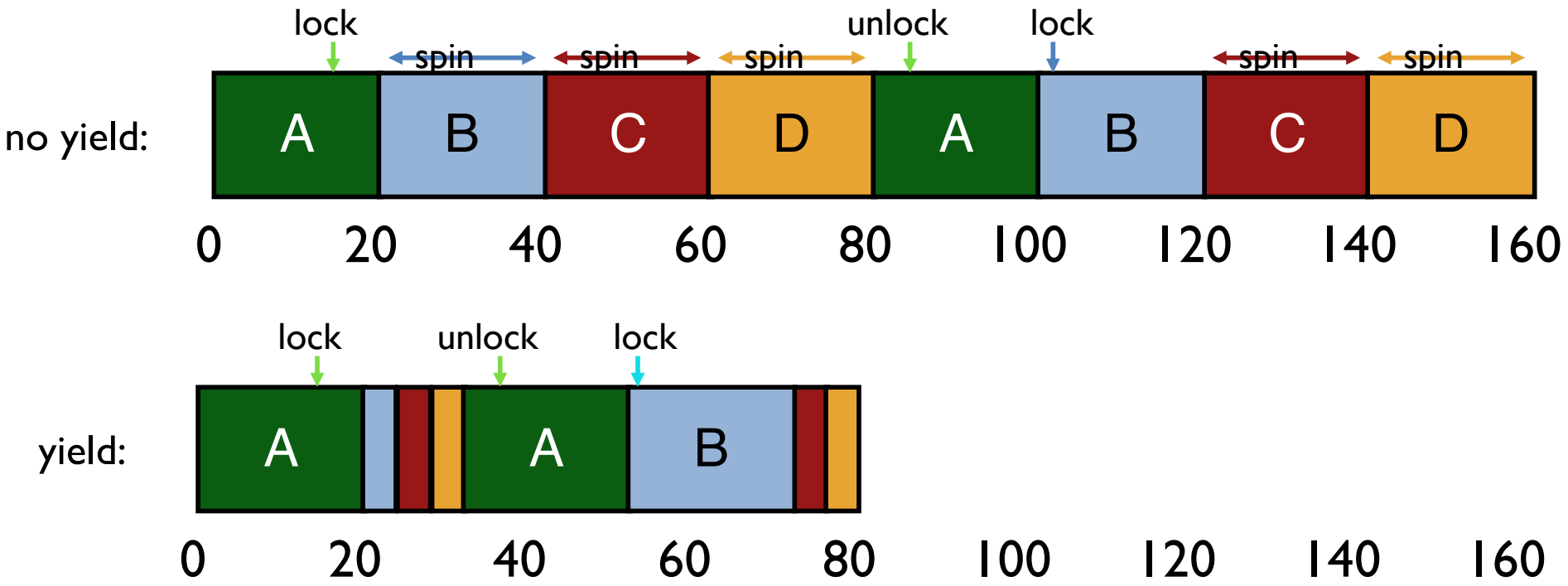
TICKET LOCK WITH YIELD

```
typedef struct __lock_t {
    int ticket;
    int turn;
}

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}
```

```
void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while (lock->turn != myturn)
        yield();
}

void release(lock_t *lock) {
    FAA(&lock->turn);
}
```

Remember: yield() voluntarily relinquishes CPU for remainder of timeslice, but process remains READY

YIELD INSTEAD OF SPIN



SPINLOCK PERFORMANCE

Waste of CPU cycles?

- Without yield: O(threads * **time_slice**)
- With yield: O(threads * **context_switch**)

Even with yield, spinning is slow with high thread contention

Next improvement: Block and put thread on waiting queue instead of spinning

QUEUE LOCKS

LOCK IMPLEMENTATION: BLOCK WHEN WAITING

Remove waiting threads from scheduler ready queue

Move to BLOCKED or WAITING state

(e.g., park() and unpark(threadID))

Scheduler runs any thread that is **ready**

Good separation of concerns between lock and scheduler

RUNNABLE: A, B, C, D

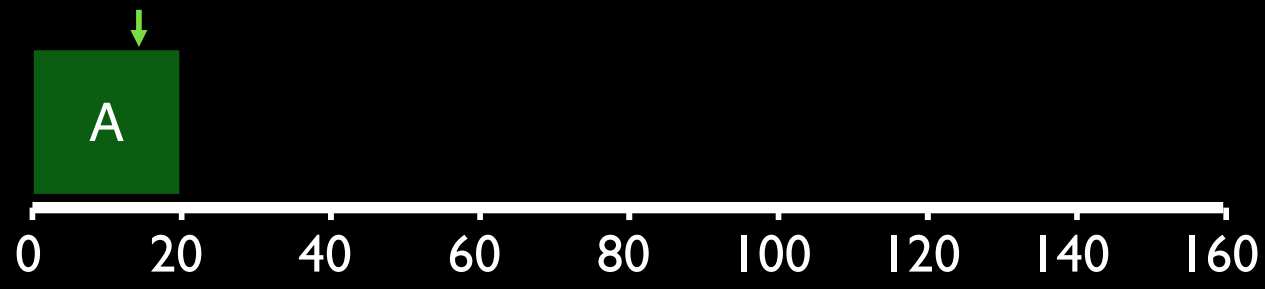
RUNNING: <empty>

WAITING: <empty>

RUNNABLE: B, C, D

RUNNING: A

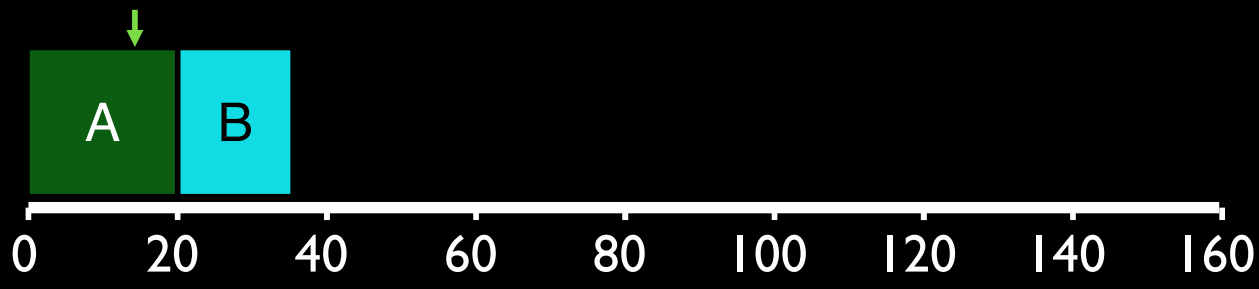
WAITING: <empty>



RUNNABLE: C, D, A

RUNNING: B

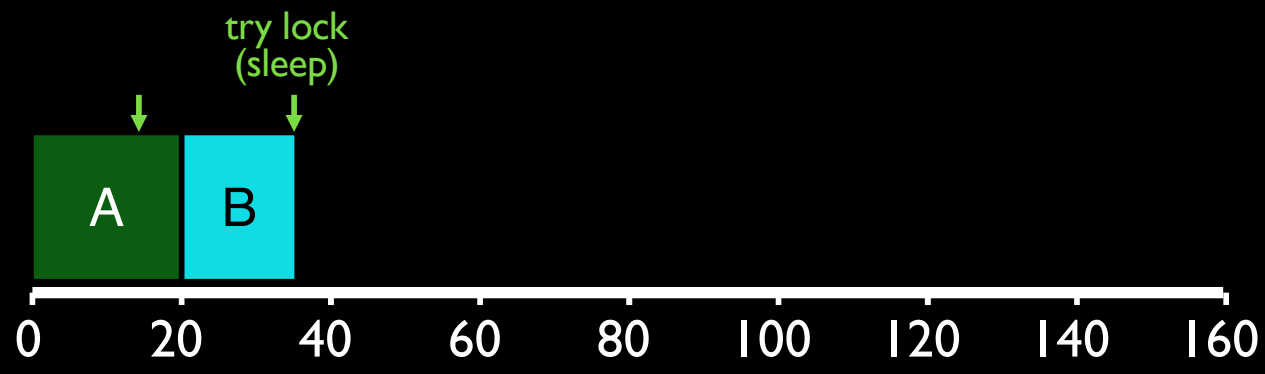
WAITING: <empty>



RUNNABLE: C, D, A

RUNNING:

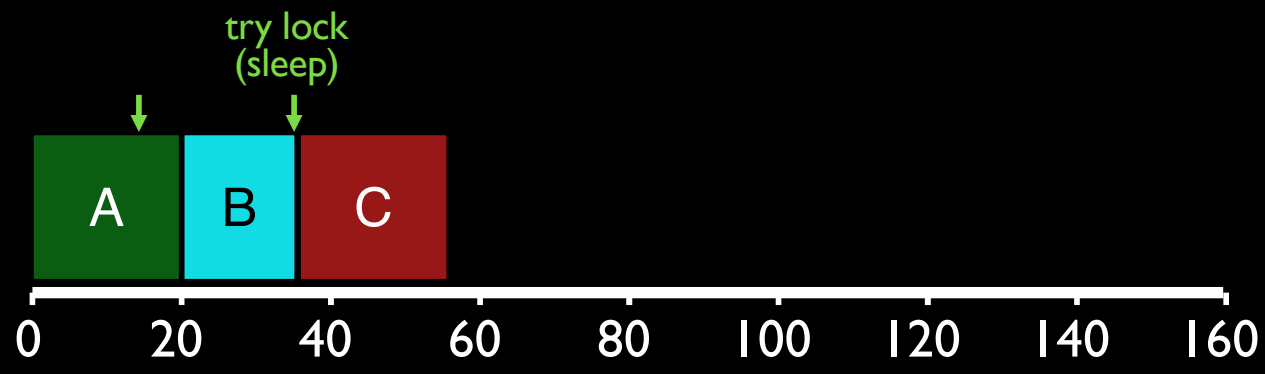
WAITING: B



RUNNABLE: D, A

RUNNING: C

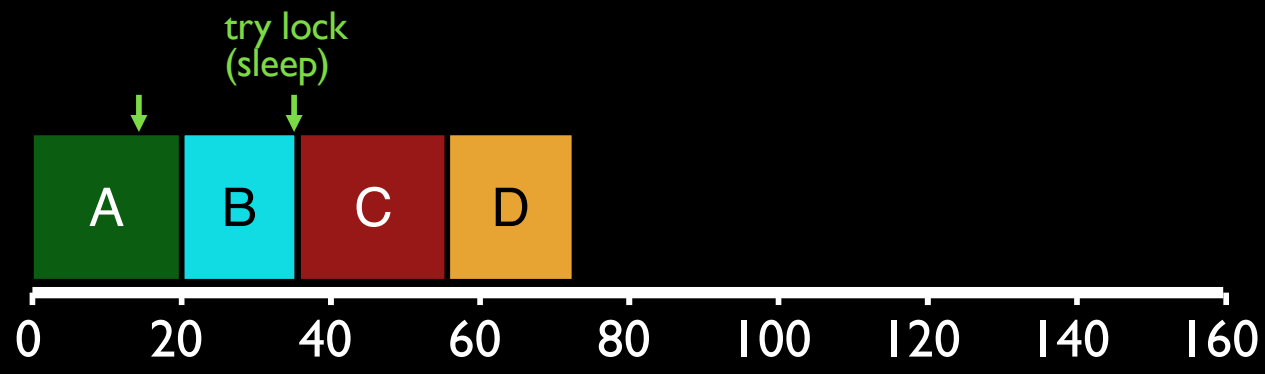
WAITING: B



RUNNABLE: A, C

RUNNING: D

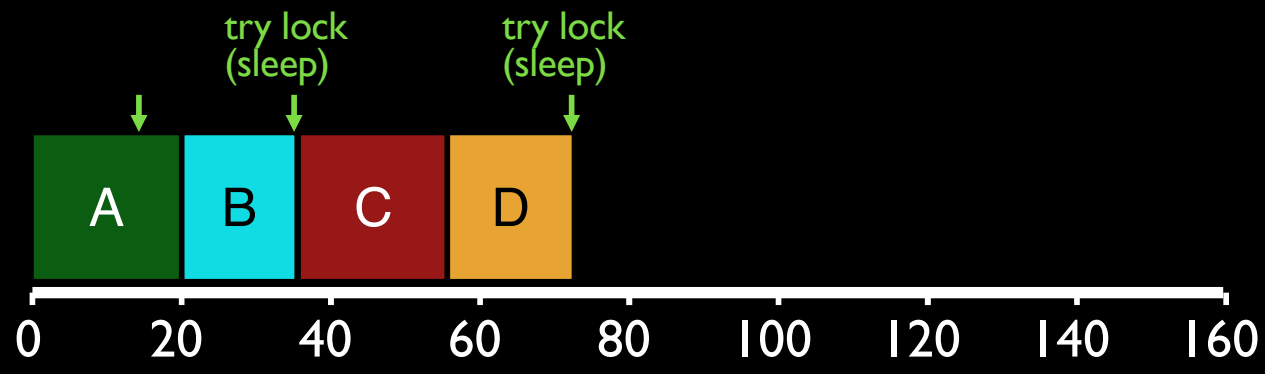
WAITING: B



RUNNABLE: A, C

RUNNING:

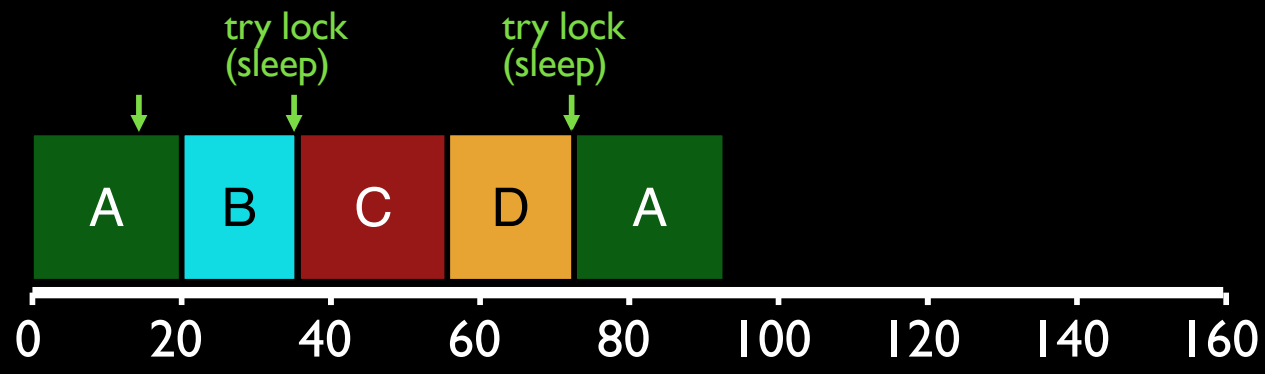
WAITING: B, D



RUNNABLE: C

RUNNING: A

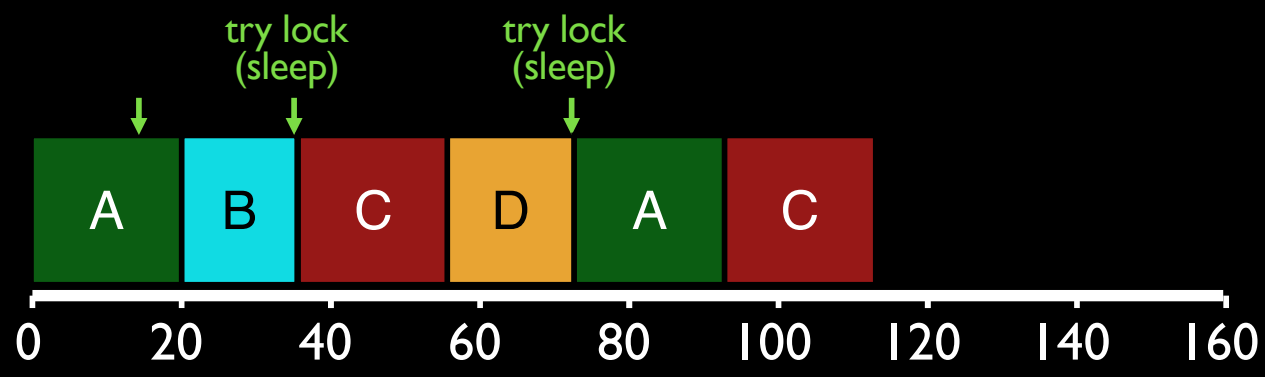
WAITING: B, D



RUNNABLE: A

RUNNING: C

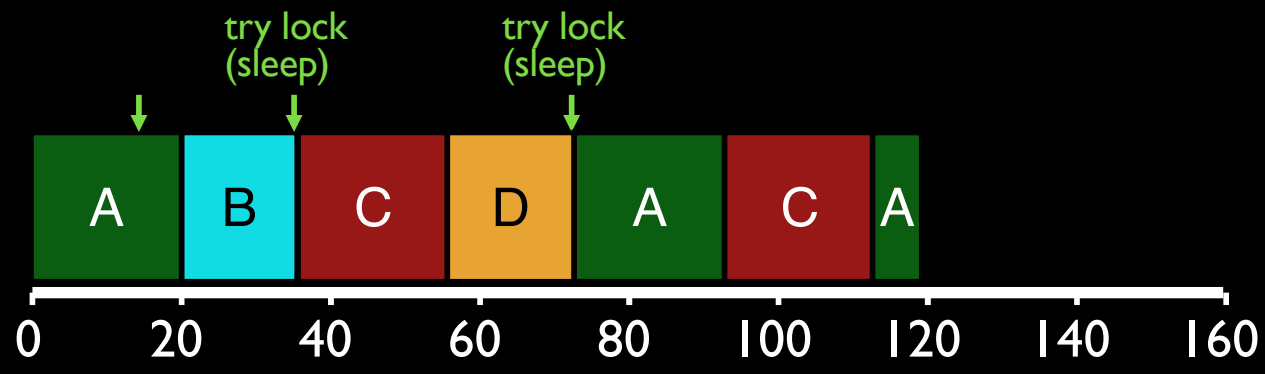
WAITING: B, D



RUNNABLE: C

RUNNING: A

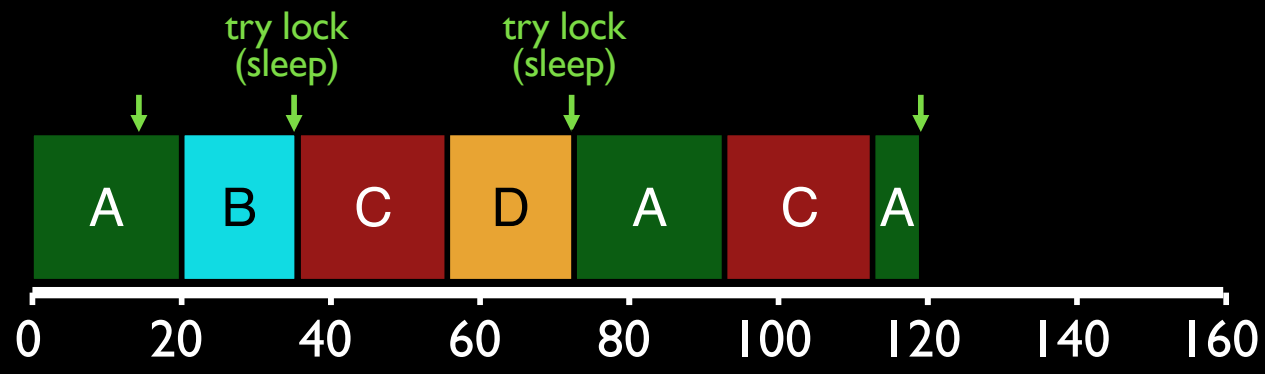
WAITING: B, D



RUNNABLE: B, C

RUNNING: A

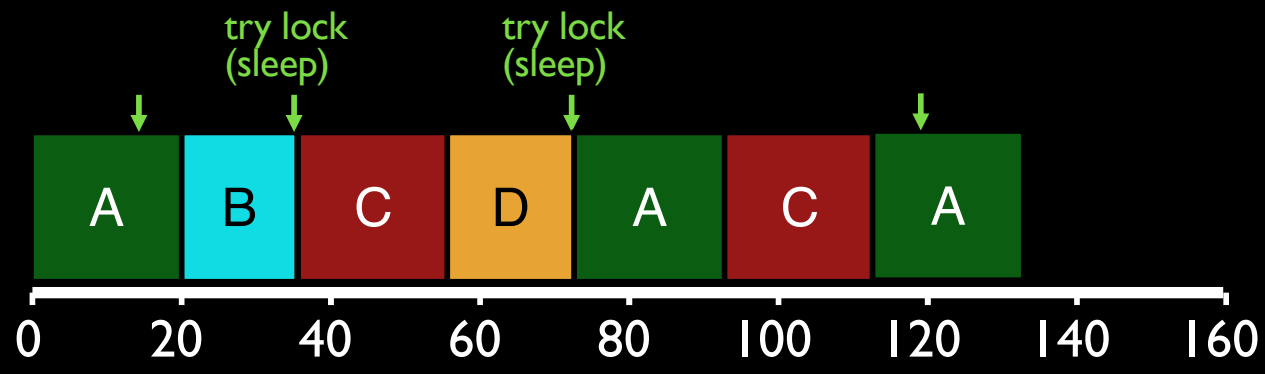
WAITING: D



RUNNABLE: B, C

RUNNING: A

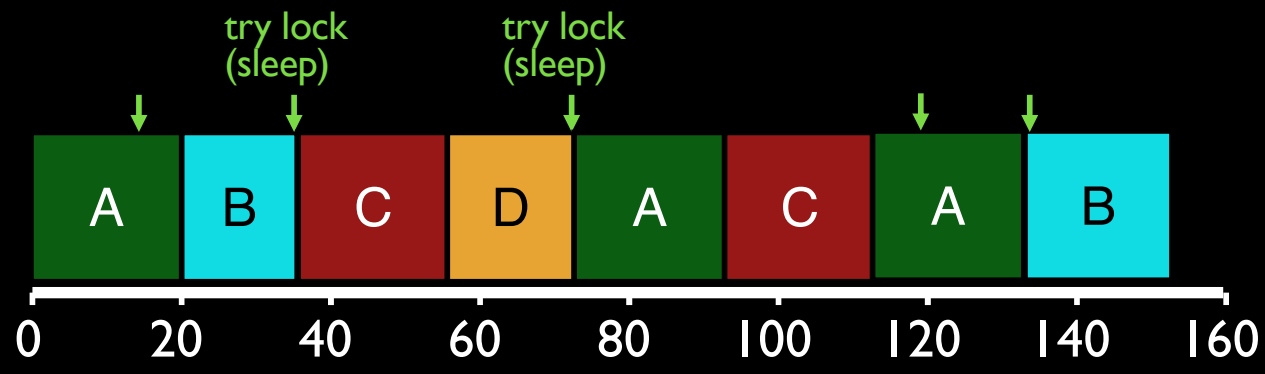
WAITING: D



RUNNABLE: C, A

RUNNING: B

WAITING: D



LOCK IMPLEMENTATION #1: BLOCK WHEN WAITING

```
typedef struct {
   bool lock = false;
   queue_t q;
} LockT;
```

Track waiting processes on q

```
void acquire(LockT *l) {

     if (l->lock) {
          qadd(l->q, tid);
          park();      // blocked
   } else {
          l->lock = true;
   }
}

void release(LockT *l) {
   if (qempty(l->q)) l->lock=false;
   else unpark(qremove(l->q));
}
```

LOCK IMPLEMENTATION #2: BLOCK WHEN WAITING

```
typedef struct {
   bool lock = false;
   bool guard = false;
   queue_t q;
} LockT;
```

Add guard to lock

```
void acquire(LockT *l) {
   while (XCHG(&l->guard, true));
   if (l->lock) {
         qadd(l->q, tid);
         l->guard = false;
         park();     // blocked
   } else {
         l->lock = true;
         l->guard = false;
   }
}
```

```
void release(LockT *l) {
   while (XCHG(&l->guard, true));
   if (qempty(l->q)) l->lock=false;
   else unpark(qremove(l->q));
   l->guard = false;
}
```

LOCK IMPLEMENTATION: BLOCK WHEN WAITING

(a) Why is **guard** used?

(b) Why okay to **spin** on guard?

(c) In release(), why not set lock=false when unpark?

(d) Is there a race condition?

```
void acquire(LockT *l) {
   while (XCHG(&l->guard, true));
   if (l->lock) {
         qadd(l->q, tid);
         l->guard = false;
         park();      // blocked
   } else {
         l->lock = true;
         l->guard = false;
   }
}

void release(LockT *l) {
   while (XCHG(&l->guard, true));
   if (qempty(l->q)) l->lock=false;
   else unpark(qremove(l->q));
   l->guard = false;
}
```

RACE CONDITION

Thread 1 (in lock)

```
if (l->lock) {
    qadd(l->q, tid);
    l->guard = false;

    park();     // block
```

Thread 2 (in unlock)

```
while (TAS(&l->guard, true));
if (qempty(l->q)) // false!!
else unpark(qremove(l->q));
l->guard = false;
```

Problem: Guard not held when call park()
Unlocking thread may unpark() before other park()

BLOCK WHEN WAITING: FINAL CORRECT LOCK

```
typedef struct {
   bool lock = false;
   bool guard = false;
   queue_t q;
} LockT;
```

setpark() fixes race condition

Park() does not block if unpark() occurred after setpark()

```
void acquire(LockT *l) {
   while (TAS(&l->guard, true));
   if (l->lock) {
         qadd(l->q, tid);
         setpark(); // notify of plan
         l->guard = false;
         park(); // unless unpark()
   } else {
         l->lock = true;
         l->guard = false;
   }
}
void release(LockT *l) {
   while (TAS(&l->guard, true));
   if (qempty(l->q)) l->lock=false;
   else unpark(qremove(l->q));
   l->guard = false;
}
```

REASONING ABOUT LOCKS

- When using locks, don't assume any implementation details are necessary for correctness of calling process
- Don't assume any particular ordering for which process acquires lock next
- Your application code must work correctly if any process acquires lock next

PERFORMANCE: SPIN-WAITING VS BLOCKING

Each approach is better under different circumstances

Uniprocessor

- Waiting process is scheduled → Process holding lock isn't
- Waiting process should always relinquish processor
- Associate queue of waiters with each lock (as in previous implementation)

Multiprocessor

- Waiting process is scheduled → Process holding lock might be
- Spin or block depends on how long, t, before lock is released
  - Lock released quickly → Spin-wait
  - Lock released slowly → Block
  - Quick and slow are relative to context-switch cost, C

WHEN TO SPIN-WAIT? WHEN TO BLOCK?

If know how long, **t,** before lock released, can determine optimal behavior

How much CPU time is wasted when spin-waiting?

t

How much wasted when block?

c

What is the best action when t < C?

When t > C? spin-wait

block

Problem:

Requires knowledge of future;
too much overhead to do any special prediction

TWO-PHASE WAITING

Theory: Bound worst-case performance;
ratio of actual / optimal

When would worst-possible performance occur?

Spin for very long time t >> C
Ratio: t/C (unbounded)

Algorithm: Spin-wait for time C then block --> Factor of 2 of optimal

Two cases:

- t < C: optimal approach spin-waits for t; we also spin-wait t
- t > C: optimal blocks immediately (cost of C); we pay spin C then block (cost of 2 C);
  2C / C → 2-competitive algorithm

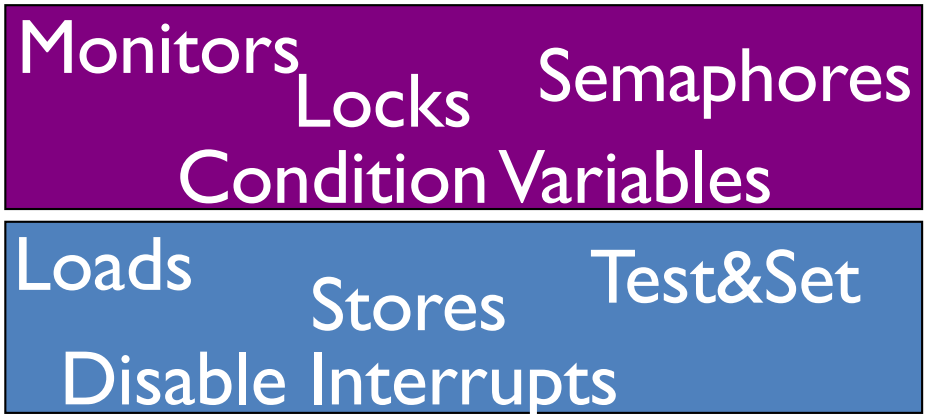
Example of competitive analysis

IMPLEMENTING SYNCHRONIZATION

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right



CONDITION VARIABLES

CONCURRENCY OBJECTIVES

Mutual exclusion (e.g., A and B don't run at same time)

- solved with *locks*

Ordering (e.g., B runs after A does something)

- solved with *condition variables* and *semaphores*

ORDERING EXAMPLE: JOIN

```
pthread_t p1, p2;
Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");

// join waits for specified thread to finish
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);

printf("main: done\n [balance: %d]\n [should: %d]\n",
    balance, max*2);
return 0;
```

how to implement join()?

CONDITION VARIABLES

Condition Variable: queue of waiting threads

B waits for a signal on CV before running

- wait(CV, …)

A sends signal to CV when time for ***B*** to run

- signal(CV, …)

CONDITION VARIABLES

wait(cond_t *cv, mutex_t *lock)

- assumes the specified lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

signal(cond_t *cv)

- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing

JOIN IMPLEMENTATION: ATTEMPT 1

Parent:

```
void thread_join() {
        Mutex_lock(&m);       // x
        Cond_wait(&c, &m);    // y
        Mutex_unlock(&m);     // z
}
```

Child:

```
void thread_exit() {
        Cond_signal(&c);      // a
}
```

Example schedule:

Works!?

Parent:	x	y		z
Child:			a	

JOIN IMPLEMENTATION: ATTEMPT 1

Parent:

```
void thread_join() {
        Mutex_lock(&m);       // x
        Cond_wait(&c, &m);    // y
        Mutex_unlock(&m);     // z
}
```

Child:

```
void thread_exit() {
        Cond_signal(&c);      // a
}
```

Parent waits forever!

Example broken schedule:

Parent:		x	y
Child:	a		

CV RULE OF THUMB 1

Keep state in addition to CV's!

CV's are used to signal threads when state changes

If state is already as needed, thread doesn't wait for a signal!

JOIN IMPLEMENTATION: ATTEMPT 2

Int done = 0; // shared between parent and child

Parent:

```
void thread_join() {
        Mutex_lock(&m);          // w
        if (done == 0)           // x
            Cond_wait(&c, &m); // y
        Mutex_unlock(&m);        // z
}
```

Child:

```
void thread_exit() {
        done = 1;          // a
        Cond_signal(&c); // b
}
```

Fixes previous broken ordering:

Parent: w x y z

Child: a b

JOIN IMPLEMENTATION: ATTEMPT 2

Parent:

```
void thread_join() {
        Mutex_lock(&m);          // w
        if (done == 0)           // x
            Cond_wait(&c, &m); // y
        Mutex_unlock(&m);        // z
}
```

Child:

```
void thread_exit() {
        done = 1;          // a
        Cond_signal(&c); // b
}
```

Parent waits forever!

Can you construct ordering that does not work?

Parent: w x y

Child: a b

JOIN IMPLEMENTATION: CORRECT

Parent:

```
void thread_join() {
        Mutex_lock(&m);          // w
        if (done == 0)           // x
            Cond_wait(&c, &m); // y
        Mutex_unlock(&m);        // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);      // a
        done = 1;            // b
        Cond_signal(&c);     // c
        Mutex_unlock(&m);    // d
}
```

Parent: w x y z

Child: ? a b c

Use mutex to ensure no race between interacting with state and wait/signal

Essential that mutex is released within cond wait()

CV RULE OF THUMB 2

Modify state with mutex held (in threads calling wait and signal)

Mutex is required to ensure state does not change between testing of state and waiting on CV

PRODUCER/CONSUMER PROBLEM

EXAMPLE: UNIX PIPES

A pipe may have many writers and readers

Internally, there is a finite-sized, circular buffer

Writers add data to the buffer

- Writers have to wait if buffer is full

Readers remove data from the buffer

- Readers have to wait if buffer is empty

PRODUCER/CONSUMER PROBLEM

Producers generate data (like pipe writers)

Consumers grab data and process it (like pipe readers)

Producer/consumer problems are frequent in systems (e.g. web servers)

General strategy use condition variables to:

- make producers wait when buffers are full
- make consumers wait when buffers are empty

PRODUCE/CONSUMER EXAMPLE

Start with easy case:

- 1 producer thread
- 1 consumer thread
- 1 shared buffer to fill/consume (max = 1)

Numfull = number of buffers currently filled

max = 1

numfull = 0

Thread 1 state:

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Assume do_fill() increments numfull

Thread 2 state:

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

Assume do_get() decrements numfull

numfull = 0

Thread 1 state: RUNNABLE

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Thread 2 state: RUNNING

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

numfull = 0

Thread 1 state: RUNNABLE

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Thread 2 state: RUNNING

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

numfull = 0

Thread 1 state: RUNNABLE

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Thread 2 state: RUNNING

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

numfull = 0

Thread 1 state: RUNNABLE

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Thread 2 state: RUNNING

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

numfull = 0

Thread 1 state: RUNNABLE

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Consumer releases mutex in cond_wait

Thread 2 state: BLOCKED on CV

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

numfull = 0

Thread 1 state: RUNNING

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Thread 2 state: BLOCKED on CV

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

Will producer be stuck waiting for mutex_lock()?

numfull = 0

Thread 1 state: RUNNING

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

No, because cond_wait released lock

Thread 2 state: BLOCKED on CV

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

numfull = 0

Thread 1 state: RUNNING

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Numful != max

Thread 2 state: BLOCKED on CV

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

numfull = 1

Thread 1 state: RUNNING

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

What happens to consumer?

Thread 2 state: BLOCKED on CV

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

numfull = 1

Thread 1 state: RUNNING

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Thread 2 state: BLOCKED on MUTEX

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

Consumer must reacquire mutex to return from cond_wait

numfull = 1

Thread 1 state: RUNNING

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Thread 2 state: RUNNABLE

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

numfull = 1

Thread 1 state: RUNNING

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Thread 2 state: RUNNABLE

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
```

Who acquires lock next?

No assumptions for correctness! Could be either thread!
Example gives lock to producer

numfull = 1

Thread 1 state: RUNNING

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Thread 2 state: RUNNABLE

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

numfull = 1

Thread 1 state: RUNNING

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Thread 2 state: RUNNABLE

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

What important thing happens during cond_wait()?

numfull = 1

Thread 1 state: BLOCKED on CV

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

Producer releases mutex

Thread 2 state: RUNNING, Acquires lock

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

numfull = 1

Thread 1 state: BLOCKED on CV

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill();
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

And so on…

Thread 2 state: RUNNING, Acquires lock

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        do_work(tmp);
    }
}
```

WHAT ABOUT 2 CONSUMERS?

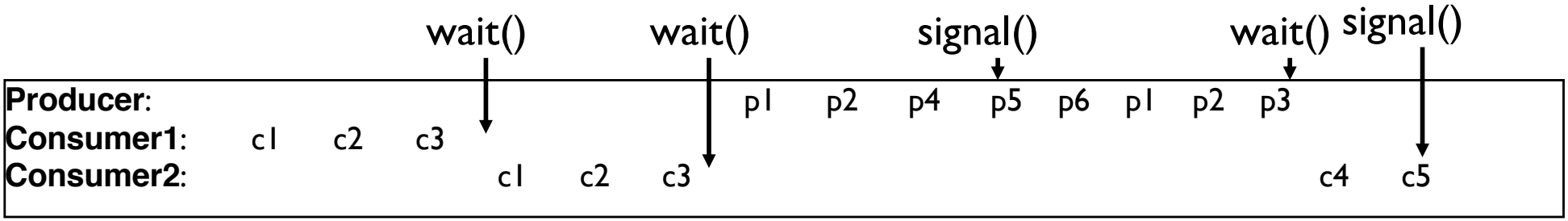
Can you find a problematic timeline with 2 consumers (still 1 producer)?

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m); // p1
        if(numfull == max) //p2
            Cond_wait(&cond, &m); //p3
        do_fill(); // p4
        Cond_signal(&cond); //p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);  // c1
        if(numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        do_work(tmp); // c7
    }
}
```

Want consumer2 signal() to wake producer since numbufs = 0, but could wake consumer1

Cannot assume which waiting thread will be worken!



HOW TO WAKE THE RIGHT THREAD?

Wake all the threads!?

WAKING ALL WAITING THREADS

wait(cond_t *cv, mutex_t *lock)

- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

signal(cond_t *cv)

- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing

broadcast(cond_t *cv)

- wake **all** waiting threads (if >= 1 thread is waiting)
- if there are no waiting thread, just return, doing nothing

HOW TO WAKE THE RIGHT THREAD?

Wake all the threads!?
Better solution (usually): use separate condition variables

PRODUCER/CONSUMER: TWO CVS

```
void *producer(void *arg) {
    While (1) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill();  // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

Is this correct? Can you find a bad schedule?

PRODUCER/CONSUMER: TWO CVS

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill();  // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs lock, gets data and sets numfull=0.
4. When consumer2 wakes from cond_wait with lock, reads bad data

PRODUCER/CONSUMER: TWO CVS

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill();  // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        if (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

Cannot assume which threads will acquire lock next

When wake from cond_wait(), must recheck state to ensure state is indeed true

(i.e., no other thread changed state between cond_signal() returning from cond_wait())

PRODUCER/CONSUMER: TWO CVS AND WHILE

```
void *producer(void *arg) {
    while (1) {
        Mutex_lock(&m); // p1
        while (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill();  // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        while (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

Lock of mutex: No concurrent access to shared state

Every time lock is re-acquired, assumptions are reevaluated (while loop instead of if)

Progress: A consumer will get to run after every do_fill()

Progress: A producer will get to run after every do_get()

GOOD RULE OF THUMB 3

Whenever a lock is acquired, recheck assumptions about state!

Another thread could grab lock in between signal and wakeup from wait

Some implementations have "spurious wakeups"

- May wake multiple waiting threads at signal or at any time
- May treat signal() as broadcast()
- Good way to stress test your code: change signals to broadcasts

SUMMARY: RULES OF THUMB FOR CVS

1. Keep state in addition to CV's

2. Always change state and do wait/signal with lock held

3. Whenever thread wakes from waiting, recheck state

IMPLEMENTING SYNCHRONIZATION

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right