

# CONCURRENCY: QUEUE LOCKS, CONDITION VARIABLES

Shivaram Venkataraman  
CS 537, Spring 2019

# ADMINISTRIVIA

- Project 3 is out
- Project 2a grades are out

# AGENDA / LEARNING OUTCOMES

## Concurrency

How do we make locks more efficient?

How to support threads that need to conditionally execute?

# RECAP

# LOCK IMPLEMENTATION GOALS

## Correctness

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

Fairness: Each thread waits for same amount of time

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)  
}  
yield();
```

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

lock  
Text and set  
mutual exclusion

# 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

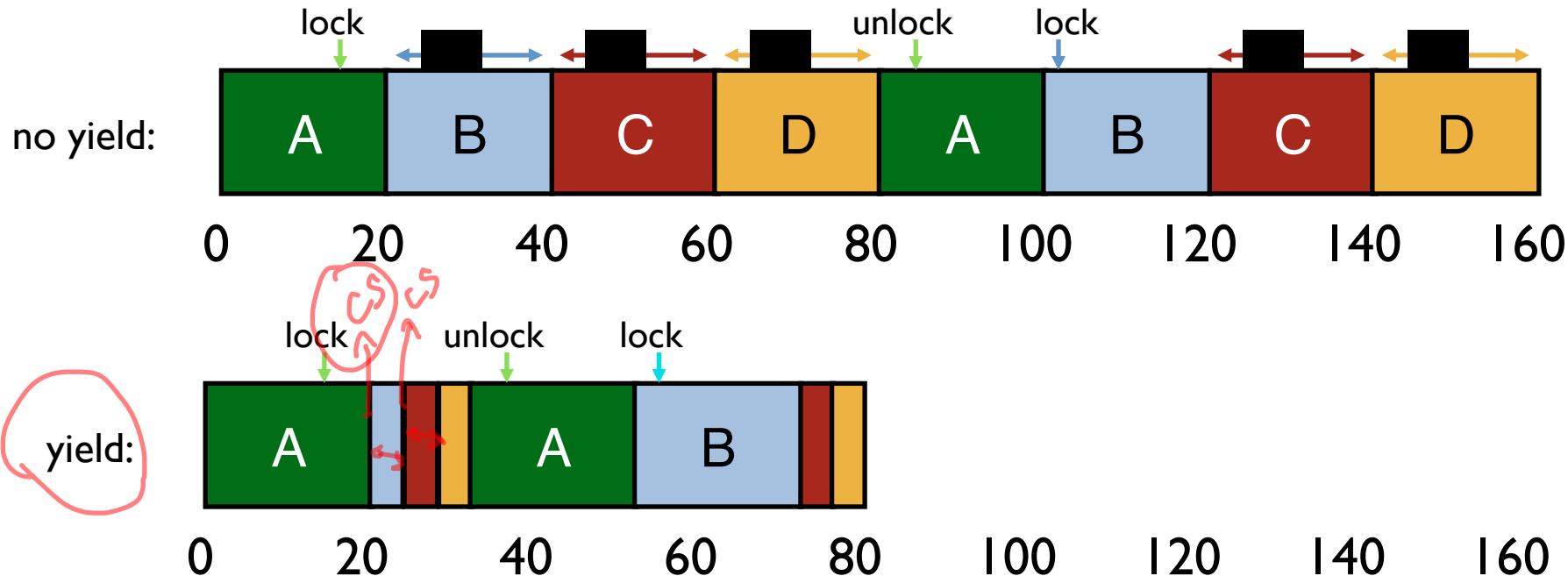
```
int FetchAndAdd(int *ptr) {  
    int old = *ptr;  
    *ptr = old + 1;  
    return old;  
}
```

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

Release: Advance to next turn

which thread gets the lock

# YIELD INSTEAD OF SPIN



<https://tinyurl.com/cs537-sp19-bunny5>

Assuming round robin scheduling, 10ms time slice  
Processes A, B, C, D, E, F, G, H, I, J in the system

Timeline

A: lock() ... compute ... unlock()  
B: lock() ... compute ... unlock()  
C: lock()

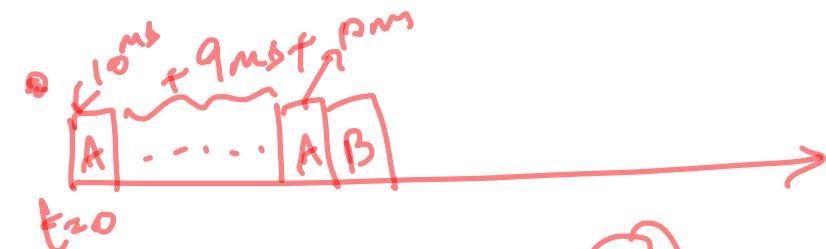


# YIELD VS SPIN

Assuming round robin scheduling, 10ms time slice → Scheduler  
Processes A, B, C, D, E, F, G, H, I, J in the system

Timeline

A: lock() ... compute ... unlock()  
B: lock() ... compute ... unlock()  
C: lock()



If A's compute is 20ms long, starting at  $t = 0$ , when does B get lock with spin ?

110 ms

If B's compute is 30ms long, when does C get lock with spin ?

320 ms

If context switch time = 1ms, when does B get lock with yield ?

29 ms

# SPINLOCK PERFORMANCE

Waste of CPU cycles?

Without yield:  $O(\text{threads} * \text{time\_slice})$

With yield:  $O(\text{threads} * \text{context\_switch})$

Even with yield, spinning is slow with high thread contention

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

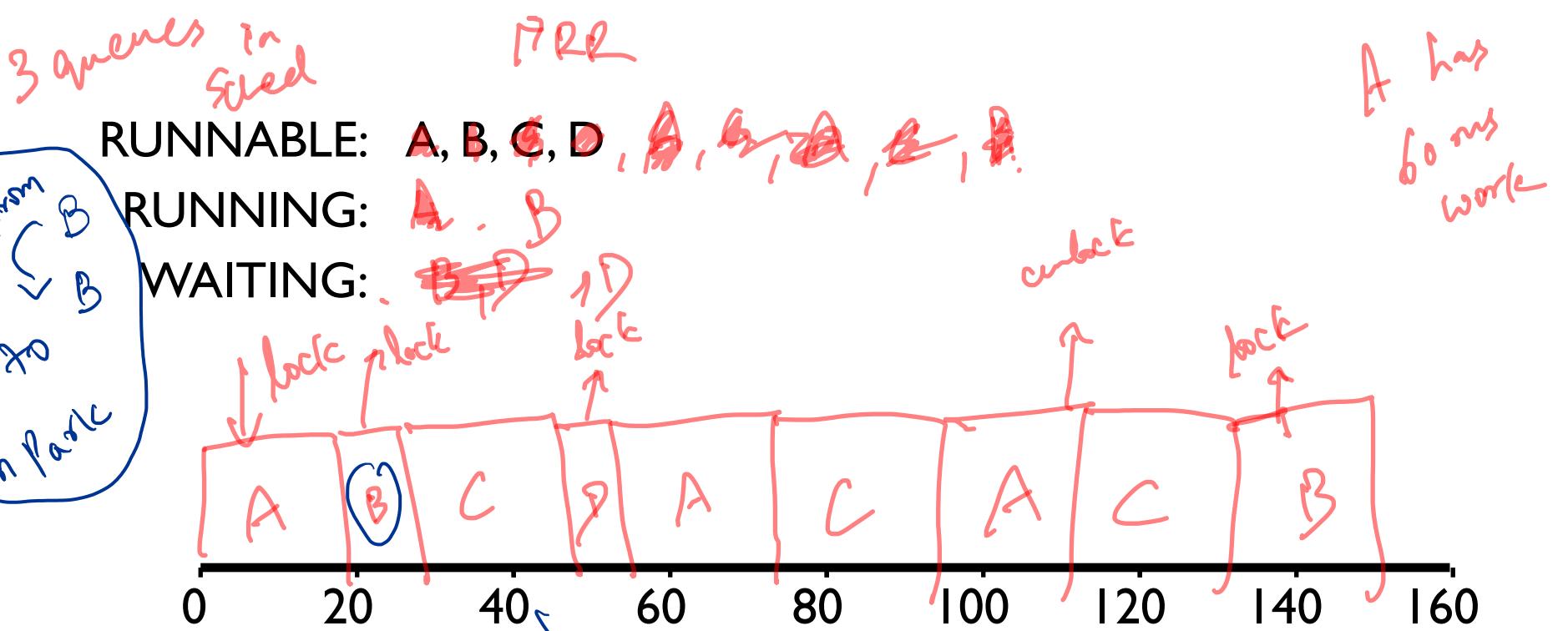
# QUEUE LOCKS

# LOCK IMPLEMENTATION: BLOCK WHEN WAITING

Remove waiting threads from scheduler ready queue

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

Scheduler runs any thread that is **ready**



ms

A, B, C  
Content  
for lock

C  
not extending

# LOCK IMPLEMENTATION: BLOCK WHEN WAITING

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

1. Grab lock on  $l \rightarrow \text{guard}$
2. If lock is held  
add Q  
park
3. Not held  
grab 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;  
    }  
}
```

Solaris

1. Grab guard
2. Empty & release lock

3. unpark head of Q

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

Thread  
Ids

# LOCK IMPLEMENTATION: BLOCK WHEN WAITING

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

To ensure queue thread safe

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

Very short critical section

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

lock is still  
in passed from  
one thread to  
next

```
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;  
}
```

(d) Is there a race condition?

T1

# RACE CONDITION

**Thread 1** (in lock)

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

park();

// block

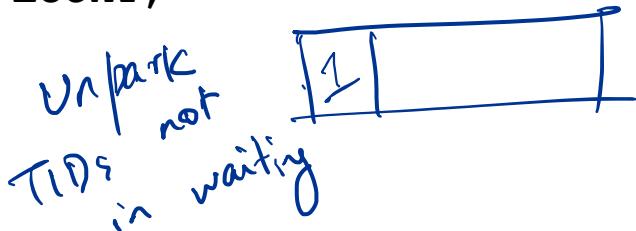
**Thread 2**

(in unlock)

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

# BLOCK WHEN WAITING: FINAL CORRECT LOCK

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



**setpark()** fixes race condition

```
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;  
}
```

<https://tinyurl.com/cs537-sp19-bunny6>



# YIELD VS BLOCKING

Assuming round robin scheduling, 10ms time slice  
Processes A, B, C, D, E, F, G, H, I, J in the system

Context switch c = 1 ms

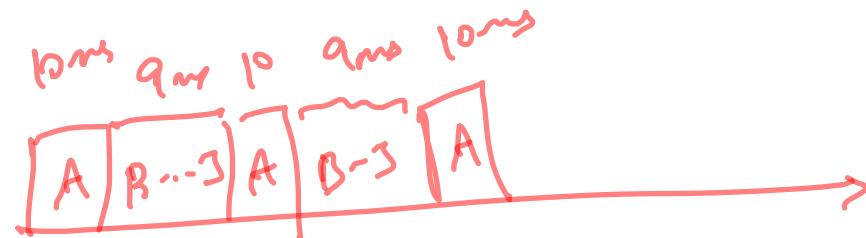
Timeline

A: lock() ... compute ... unlock()

B: lock() ... compute ... unlock()

C: lock() ...

...

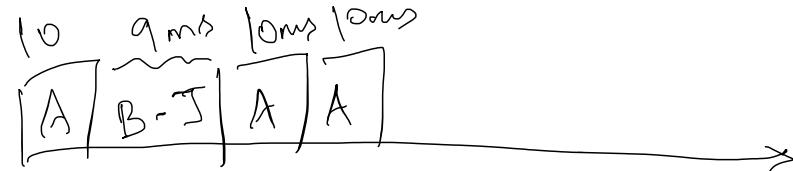


If A's compute is 30ms long, starting at t = 0, when does B get lock with yield ?

$$10 + 9 + 10 + 9 + 10 = 48 \text{ ms}$$

If A's compute is 30ms long, starting at t = 0, when does B get lock with blocking ?

39 ms



# SPIN-WAITING VS BLOCKING

Each approach is better under different circumstances

Uniprocessor → | worse

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

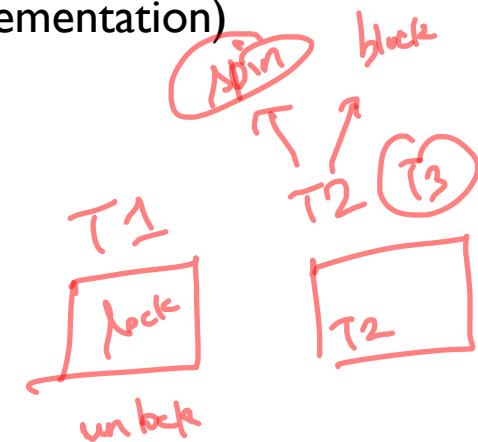
Waiting process is scheduled → Process holding lock might be

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,  $C$



# 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*

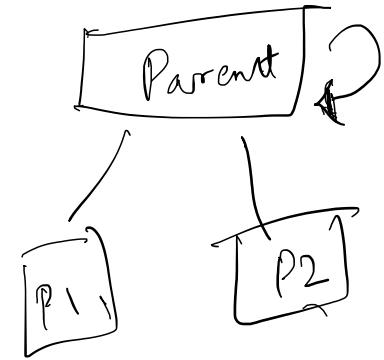
Thread A

↳ Thread B

↳ Thread C

# ORDERING EXAMPLE: JOIN

```
pthread_t p1, p2;  
Pthread_create(&p1, NULL, mythread, "A");  
Pthread_create(&p2, NULL, mythread, "B");  
// join waits for the threads 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, ...)`

wait → add itself to  
queue of waiting  
threads

signal → wakeup threads  
in queue

# CONDITION VARIABLES

**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  $\geq 1$  thread is waiting)
- if there is no waiting thread, just return, doing nothing

T1  
: lock.  
wait(cv, lock)  
---

T2  
signal(cv)

# JOIN IMPLEMENTATION: ATTEMPT 1

m, c

Parent:

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

Child:

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

Example schedule:

Parent:

x  
  y

z

Child:

a  
  b  
  c

# 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() {  
    Mutex_lock(&m);      // a  
    Cond_signal(&c);     // b  
    Mutex_unlock(&m);   // c  
}
```

Example broken schedule:

Parent:

x

y

because  
will call

Child:

a

b

c

→ stuck  
no body

signal

# 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

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  
}
```

mutex-  
lock()

mutex-  
unlock

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: w x

y

Child:

a  
~~b~~  
b

lock again (l)

# 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

# PRODUCER/CONSUMER PROBLEM

# EXAMPLE: UNIX PIPES

A pipe may have many writers and readers

Internally, there is a finite-sized 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

# EXAMPLE: UNIX PIPES

start

Buf:



end

# EXAMPLE: UNIX PIPES

Implementation:

- reads/writes to buffer require locking
- when buffers are full, writers must wait
- when buffers are empty, readers must wait

# 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 there is nothing to consume

# PRODUCE/CONSUMER EXAMPLE

Start with easy case:

- 1 producer thread
- 1 consumer thread
- 1 shared buffer to fill/consume ( $\text{max} = 1$ )

$\text{Numfull}$  = number of buffers currently filled

# numfull

Thread 1 state:

```
void *producer(void *arg) {  
    for (int i=0; i<loops; i++) {  
        Mutex_lock(&m);  
        if(numfull == max)  
            Cond_wait(&cond, &m);  
        do_fill(i);  
        Cond_signal(&cond);  
        Mutex_unlock(&m);  
    }  
}
```

Thread 2 state:

```
void *consumer(void *arg) {  
    while(1) {  
        Mutex_lock(&m);  
        if(numfull == 0)  
            Cond_wait(&cond, &m);  
        int tmp = do_get();  
        Cond_signal(&cond);  
        Mutex_unlock(&m);  
        printf("%d\n", tmp);  
    }  
}
```

# WHAT ABOUT 2 CONSUMERS?

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

```

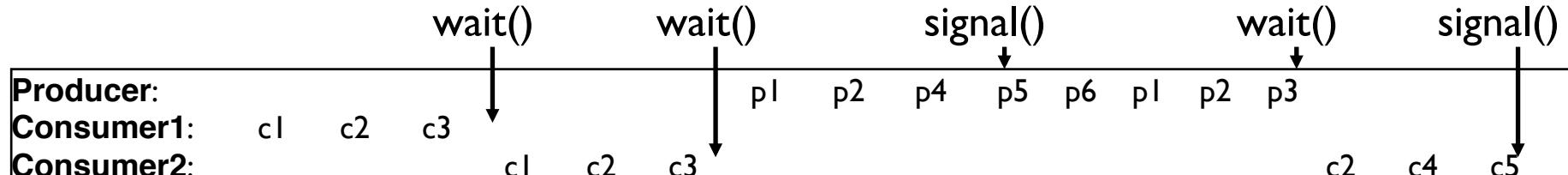
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        if(numfull == max) //p2
            Cond_wait(&cond, &m); //p3
        do_fill(i); // p4
        Cond_signal(&cond); //p5
        Mutex_unlock(&m); //p6
    }
}

```

```

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
        printf("%d\n", tmp); // c7
    }
}

```



# HOW TO WAKE THE RIGHT THREAD?

Wake all the threads!?

Better solution (usually): use two condition variables

# PRODUCER/CONSUMER: TWO CVS

```
void *producer(void *arg) {                                void *consumer(void *arg) {  
    for (int i = 0; i < loops; i++) {  
        Mutex_lock(&m); // p1  
        if (numfull == max) // p2  
            Cond_wait(&empty, &m); // p3  
        do_fill(i); // p4  
        Cond_signal(&fill); // p5  
        Mutex_unlock(&m); //p6  
    }  
}  
  
}
```

# PRODUCER/CONSUMER: TWO CVS

```
void *producer(void *arg) {                                void *consumer(void *arg) {  
    for (int i = 0; i < loops; i++) {  
        Mutex_lock(&m); // p1  
        if (numfull == max) // p2  
            Cond_wait(&empty, &m); // p3  
        do_fill(i); // p4  
        Cond_signal(&fill); // p5  
        Mutex_unlock(&m); //p6  
    }  
}  
  
}  
}
```

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs entry, sets numfull=0.
4. consumer2 then reads bad data.

# PRODUCER/CONSUMER: TWO CVS AND WHILE

```
void *producer(void *arg) {                                void *consumer(void *arg) {  
    for (int i = 0; i < loops; i++) {  
        Mutex_lock(&m); // p1  
        while (numfull == max) // p2  
            Cond_wait(&empty, &m); // p3  
        do_fill(i); // p4  
        Cond_signal(&fill); // p5  
        Mutex_unlock(&m); // p6  
    }  
}  
  
}
```

No concurrent access to shared state

Every time lock is acquired, assumptions are reevaluated

A consumer will get to run after every do\_fill()

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

Note that some libraries also have “spurious wakeups” (may wake multiple waiting threads at signal or at any time)

# SUMMARY: RULES OF THUMB FOR CVS

1. Keep state in addition to CV's

2. Always do wait/signal with lock held

3. Whenever thread wakes from waiting, recheck state

# NEXT STEPS

Project 3: Out now!

Next class: Semaphores

# 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  $\geq 1$  thread is waiting)
- if there is no waiting thread, just return, doing nothing

**broadcast(cond\_t \*cv)**

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

# 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?

How much wasted when block?

What is the best action when  $t < C$ ?

When  $t > C$ ?