CS 423
Operating System Design:
Queue Locks and Condition Variable
03/05

Ram Alagappan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

Logistics

Slight change in schedule: this friday MP2 walkthrough by our TA

We will release grades for MP1 as soon as possible (this weekend mostly)

AGENDA / LEARNING OUTCOMES

So far:

- Locks: how to use and how to implement
- HW atomic instructions to implement locks
- Spin if not able to acquire locks

Today:

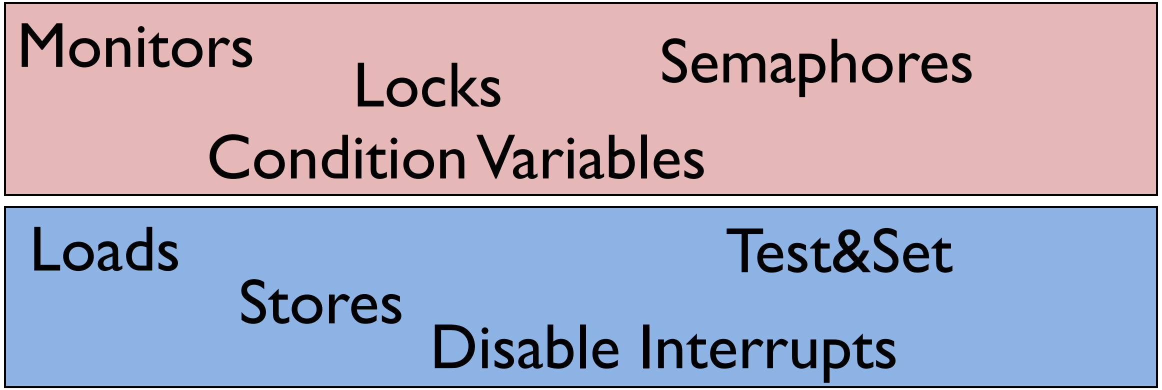
- Do something better than spinning
- Condition variables

RECAP

Synchronization

Build higher-level synchronization primitives in OS
Operations that ensure correct ordering of instructions across threads
Use help from hardware

Motivation: Build them once and get them right



Lock Implementation Goals

Correctness

- *Mutual exclusion (safety)*
  Only one thread in critical section at a time
- *Progress* (liveness)
  If several simultaneous requests, must allow one to proceed

Fairness: does each thread have a fair shot at acquiring? Does anybody starve?

Performance: CPU is not used unnecessarily (spinning)

Implementing Locks

Approaches

- Disable interrupts - not good: doesn’t work on multi processors, cannot perform IO, malicious thread can take CPU for arbitrarily long
- Load and stores of words - not good: we don’t build locks this way (more theoretical)
- Using atomic hardware instructions (e.g., test and set)

LOCK Implementation with TAS

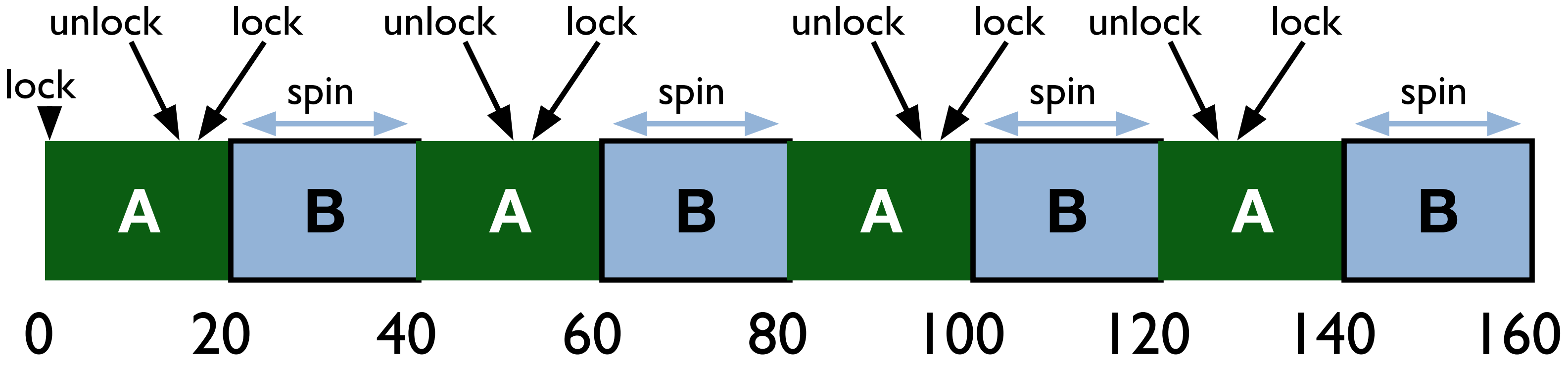
```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

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

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

Lock Using CAS

```
int CompareAndSwap(int *addr, int expected, int new) {
    int actual = *addr;
    if (actual == expected)
        *addr = new;
    return actual;
}
```

```
void acquire(lock_t *lock) {
    while(CompareAndSwap(&lock->flag, 0 , 1 ) == 1) ;
    // spin-wait (do nothing)
}
```

BASIC SPINLOCKS ARE UNFAIR



Scheduler is unaware of locks/unlocks!
B is unlucky - never is able to acquire lock

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

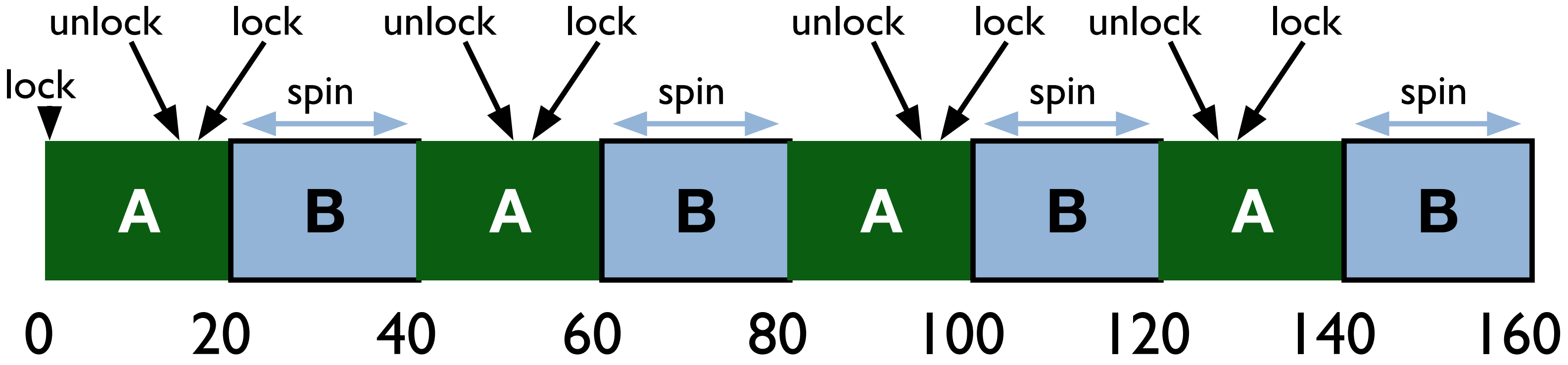
TICKET LOCK IMPLEMENTATION

```
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);
    // spin
    while (lock->turn != myturn);
}

void release(lock_t *lock) {
    FAA(&lock->turn);
    // can you do (&lock->turn)++?
}
```

END RECAP

HOW DOES TICKET LOCK PREVENT B FROM STARVING?



What will B's turn number be (when it first tries to acquire lock)?
When will B get the lock (roughly)?

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

Fairness: does each thread have a fair shot at acquiring? Does anybody starve?

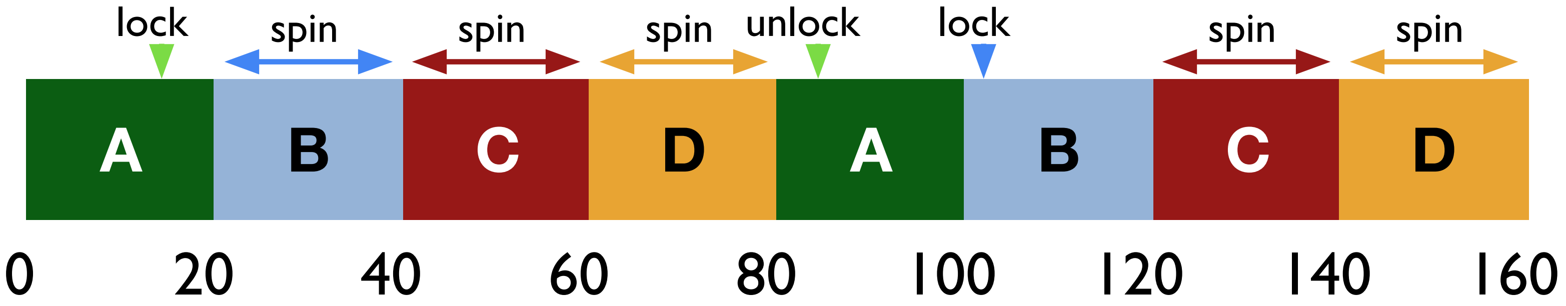
Performance: CPU is not used unnecessarily (spinning)

Spinlock Performance

Slow when…

- one CPU
- locks held a long time
- disadvantage: spinning is wasteful

CPU SCHEDULER IS IGNORANT



CPU scheduler may run **B, C, D** instead of **A**
even though **B, C, D** are waiting for **A**

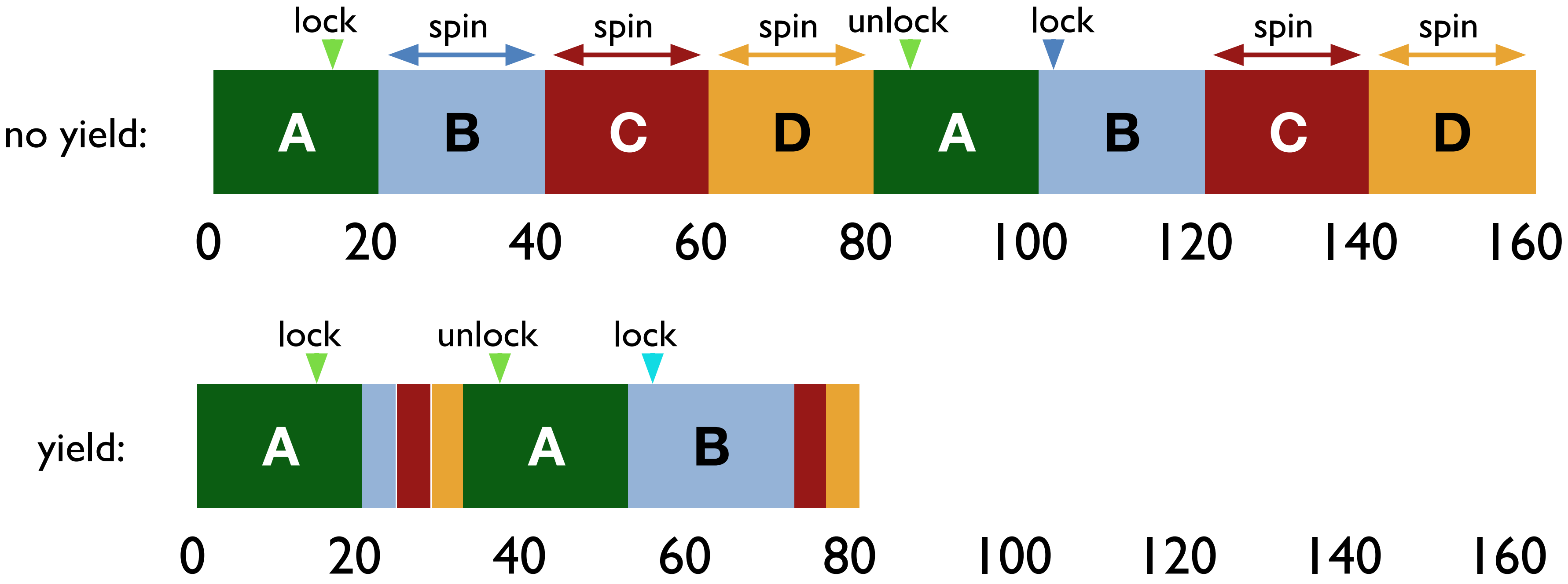
TICKET LOCK WITH YIELD

```
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);
}
```

yield() voluntarily relinquishes the CPU for remainder of time slice, but process remains READY

Yield Instead of Spin



YIELD VS SPIN

Waste of CPU cycles?

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

While yield is better than spinning, with high contention, it can also be bad

Problem: the thread is still in READY state (and so can be scheduled on CPU)

Next improvement: Block and put thread on waiting queue

Lock Implementation: Block when Waiting

Remove waiting threads from scheduler READY queue

Move to BLOCKED state

Scheduler runs any thread that is READY

Support in Solaris OS: park(), unpark()

Block when Waiting

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

```
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;
}
```

Block when Waiting

(a) Why is **guard** used? What does it protect?

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

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

```
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;
}
```

Block when Waiting

(d) What if order of guard=false and park() is changed?

(e) Is there a case where a thread may indefinitely sleep?

```
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)

Thread 2 (in unlock)

FINAL correct LOCK

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

```
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;
}
```

setpark() fixes race condition

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

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?

How much wasted when blocking?

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

When $t>C$?

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 does worst-possible performance occur?

Spin for very long time t >> C

Ratio: t/C (unbounded)

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

Two cases:

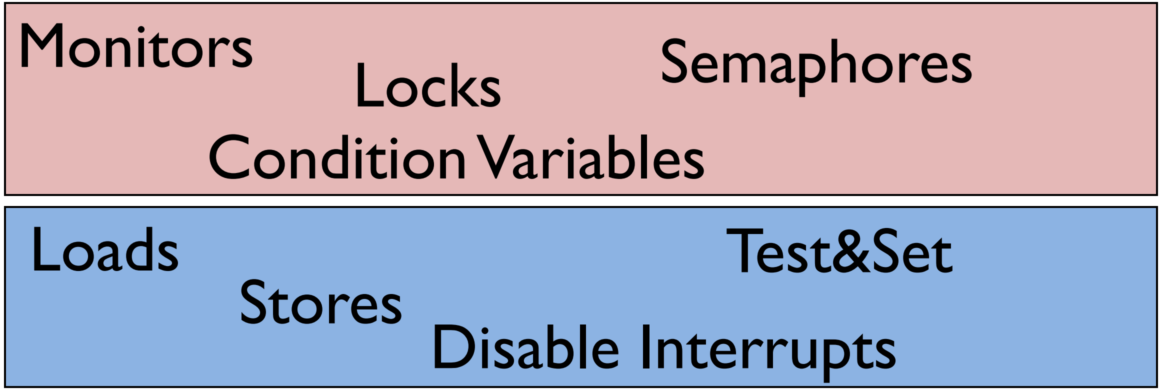
- t < C: optimal spin-waits for t; we spin-wait t too
- t > C: optimal blocks immediately (cost of C);
  - we pay spin C then block (cost of 2 C);
  - 2C / C □ 2-competitive algorithm

Linux Futex() is an example of a two-phase waiting lock…

Synchronization

Build higher-level synchronization primitives in OS
Operations that ensure correct ordering of instructions across threads
Use help from hardware

Motivation: Build them once and get them right



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 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, …)

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 >= 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() {
        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:

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
}
```

Fixes previous broken schedule

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
}
```

An example broken schedule:

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

CV Rule of Thumb 2

Modify/check state with mutex held

Mutex is required to ensure state doesn’t change between checking the state and waiting on CV

PRODUCER/CONSUMER PROBLEM

EXAMPLE: UNIX PIPES

Implementation:

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

EXAMPLE: UNIX PIPES

A pipe may have many writers and readers

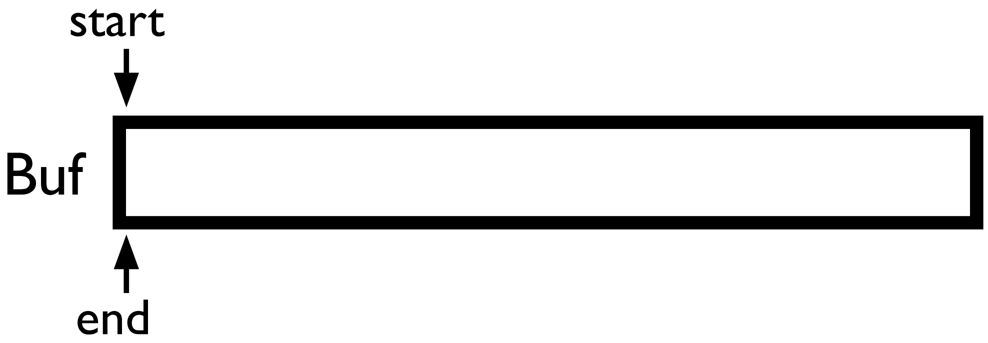
Internally, there is a finite-sized buffer

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



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 (max = 1)

Numfull = number of slots currently filled

Numfull = 0 initially

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);
    }
}
```

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?

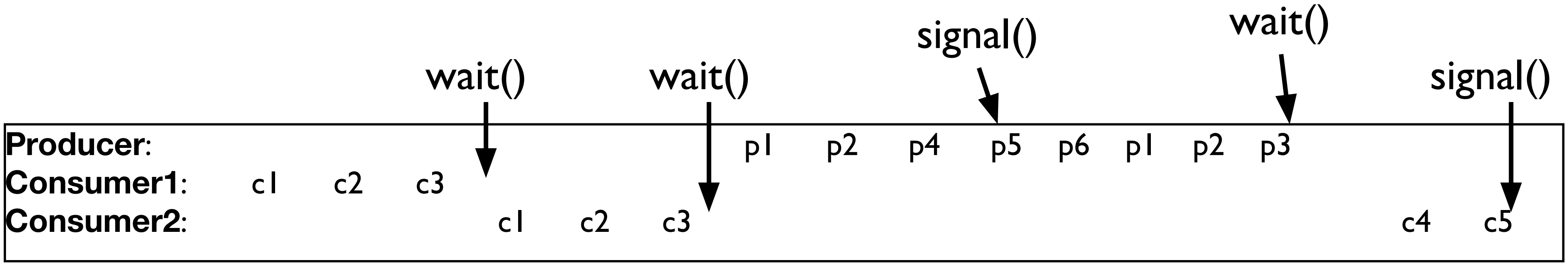
Can you find a problematic timeline with 2 consumers (still 1 producer)?

```c
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
    }
}
```

```c
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
    }
}
```

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



HOW TO WAKE THE RIGHT THREAD?

Wake all the threads!? (Broadcast)

Better solution (usually): use two condition variables

Producer/Consumer: Two CVs

```
void *producer(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
    }
}
```

```
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);
    }
}
```

Solves the previous problem…
But 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 entry, sets numfull=0.
4. consumer1 then reads bad data.

Producer/Consumer: Two CVs and WHILE

```
void *producer(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
    }
}
```

```
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);
    }
}
```

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)

Good stress test: change your signal to broadcast and see if your code still works

HOARE VS MESA SEMANTICS

- Mesa (used widely)
  - Signal puts waiter on ready list
  - Signaler keeps lock and processor
  - Not necessarily the waiter runs next
- Hoare (almost no one uses)
  - Signal gives processor and lock to waiter
  - Waiter runs when woken up by signaler
  - When waiter finishes, processor/lock given back to signaler

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