# CONCURRENCY: CONDITION VARIABLES

Shivaram Venkataraman

CS 537, Spring 2023

# RECAP

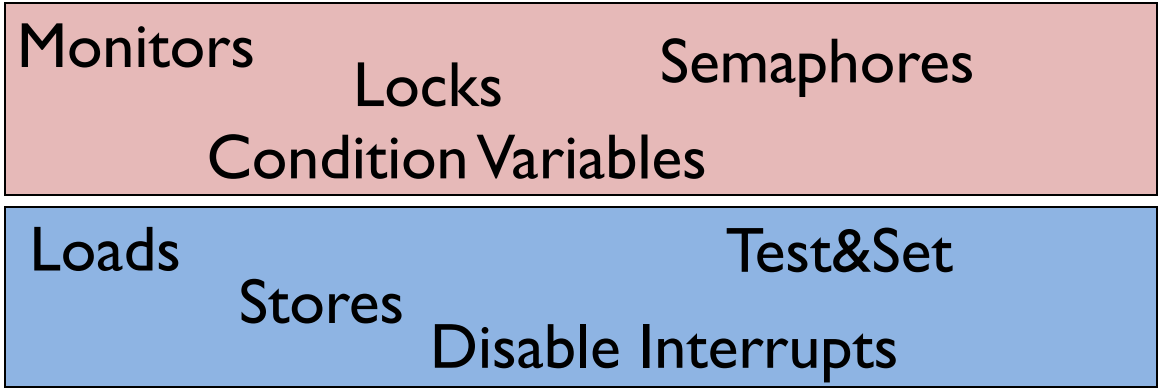
# SYNCHRONIZATION

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



# 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*

# LOCK IMPLEMENTATION WITH XCHG

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

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

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

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

```
int xchg(int *addr, int newval)
```

# 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);
}
```

# 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

# BLOCK WHEN WAITING: 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;
}
```

# 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 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

# 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:

| | | | | | |
|---|---|---|---|---|---|
| Parent: | | | | x | y |
| Child: | a | b | c | | |

# 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
}
```

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 | |

# 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

# QUIZ 16 https://tinyurl.com/cs537-sp23-quiz16

Possible outputs

Fix the code?

```
int *p = NULL; // global
void child(void *arg) {
    printf("%d\n", *p);
}

int main(int argc, char *argv[]) {
    thread_t p1;
    int x = 3;
    thread_create(&p1, child, NULL);
    p = &x;

    return 0;

}
```

# PRODUCER/CONSUMER PROBLEM

# EXAMPLE: UNIX PIPES

A pipe may have many writers and readers

Internally, there is a finite-sized buffer

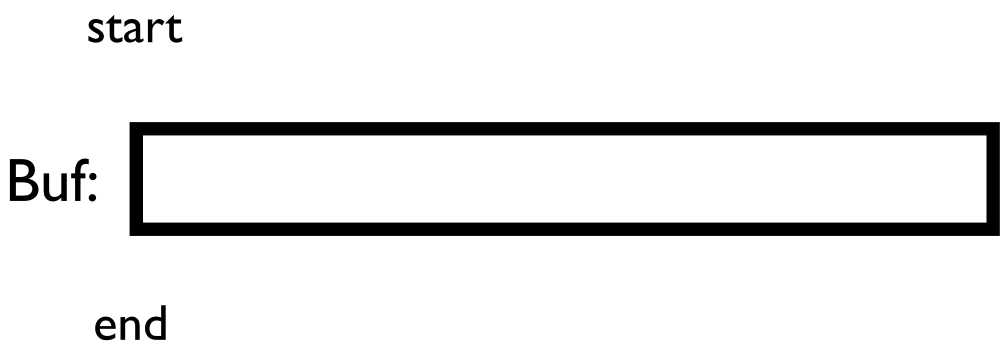
Writers add data to the buffer

- 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



# 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 (max = 1)

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
    }
}
```

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | wait() | | | wait() | | | | signal() | | | | wait() | | | signal() |
| **Producer**: | | | | | | | p1 | p2 | p4 | p5 | p6 | p1 | p2 | p3 | | | |
| **Consumer1**: | c1 | c2 | c3 | | | | | | | | | | | | | | |
| **Consumer2**: | | | | c1 | c2 | c3 | | | | | | | | | c2 | c4 | c5 |

# 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) {
    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);
    }
}
```

# 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);
    }
}
```

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) {
    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)

# 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

Next class: Semaphores