Welcome!

# CONCURRENCY: CONDITION VARIABLES

Shivaram Venkataraman

CS 537, Spring 2023

Midterm 1 : later this week

Project 4:

after Spring break : next Project.

# RECAP

# SYNCHRONIZATION

Build higher-level synchronization primitives in OS

Operations that ensure correct ordering of instructions across threads

Use help from hardware

Apps

Motivation: Build them once and get them right

Abstractions

Monitors Locks Semaphores

Condition Variables

Hardware

Loads Stores Test&Set

Disable Interrupts

# CONCURRENCY OBJECTIVES

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

mov
add
mov

- 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 = ~~??;~~ 0;
}

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

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

0 unlocked

1 locked

if thread calls this and returns → thread has lock

this threads lets go of this lock

Spin locks

CPU scheduler

↳ T1, T2

T2, T1

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

→ *addr = newval

→ return old val

Try to set flag to 1

↳ check if it was 1

# TICKET LOCK IMPLEMENTATION

Fairness

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

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

→ thread grabs a ticket

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

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

Fetch And Add

0

1

T1 → myturn = 0 → gets lock

T2 → myturn = 1 → when turn = 1

# SPINLOCK PERFORMANCE

while ( xchg(. . . . ) ) {
// do nothing
}

Waste of CPU cycles?

Without yield: O(threads * **time_slice**)

With yield: O(threads * **context_switch**)

↳ threads can yield their scheduling quanta.

Even with yield, spinning is slow with high thread contention

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

new system calls → telling scheduler that I am blocked on this lock.

# BLOCK WHEN WAITING: FINAL CORRECT LOCK

Solaris

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

unlocked

threads waiting for this lock

q

T1 T2 T3

blocked blocked

expensive

```
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 lock

Add TID to waiting queue

only returns when you get lock.

grab the lock

moves thread Ready gives it lock.

# SPIN-WAITING VS BLOCKING

lock_t;

spin lock

blocking lock

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

Blocking is useful if lock is held for a long time

# 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 → waiting till lock is released

How much wasted when blocking?

C → time taken for context switch

What is the best action when t<C? → measure

spin locks

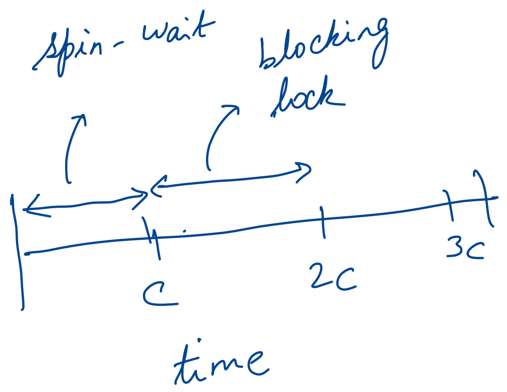
When $\boxed{t}$>C?

blocking locks

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: Actual time = t

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

$$\frac{\text{time taken}}{\text{optimal}} = \frac{2C}{C} = 2$$

# 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;
```

wait until two threads finish

how to implement join()?

# CONDITION VARIABLES

Condition Variable: queue of waiting threads

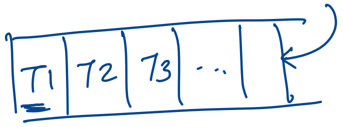
***B*** waits for a signal on CV before running

- wait(CV, …)

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

- signal(CV, …)

CV




waiting to be run

Signal()

A

B

Ordering between A and B

signal(CV)

wait() → block until some condition in A is true

# 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

API for condition variables

**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: wait until child exits

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

wait

lock reacquire

Child:

I'm done

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

Example schedule:

waiting

signal

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | wait | |
| Parent: | | | | x | y | | Parent thread stuck! |
| Child: | a | b | c | | | | |
| | | 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!

State here is
int done

# 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
}
```

~~wait~~

| | | | | | | |
|---|---|---|---|---|---|---|
| Parent: | | | w | x | ~~y~~ | z |
| Child: | a | b | | | | |

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
}
```

done = 0      wait

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

signal

parent thread stuck

# JOIN IMPLEMENTATION: CORRECT

Parent:

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

releases lock

Child:

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Parent: | w | x | y | grab lock | | | z |
| Child: | | | releases lock | 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

① Child thread executes first
→ crash

② print 3. Parent p = &x
Child printf( );

③ Process exit before child executes

## Fix the code?

thread_join(&p1);

move p = &x before thread creation

```
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;
};
```

← NULL

Process exits → terminates all threads

# 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

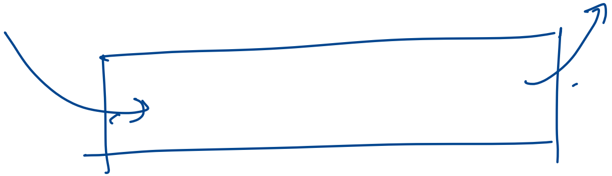
Readers remove data from the buffer

- Readers have to wait if buffer is empty

cat test.txt | grep foo

Producer/Writer

Consumer



fixed size

= 10 entries

40 KB

# EXAMPLE: UNIX PIPES

start

Buf:

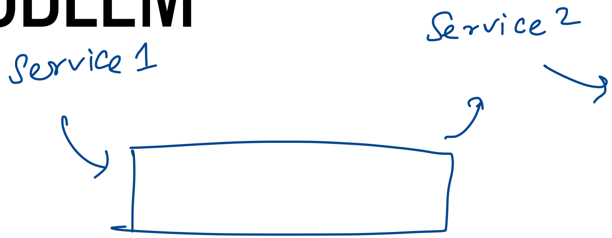
end

# 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)

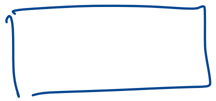
General strategy use condition variables to:

make producers wait when buffers are full

make consumers wait when there is nothing to consume

# PRODUCE/CONSUMER EXAMPLE

size = 1



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 = 0

buffer

$\boxed{2\ 3\ 5}$

Thread 1 state:

iterations

```
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);
    }
}
```

signal to consumer

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);
    }
}
```

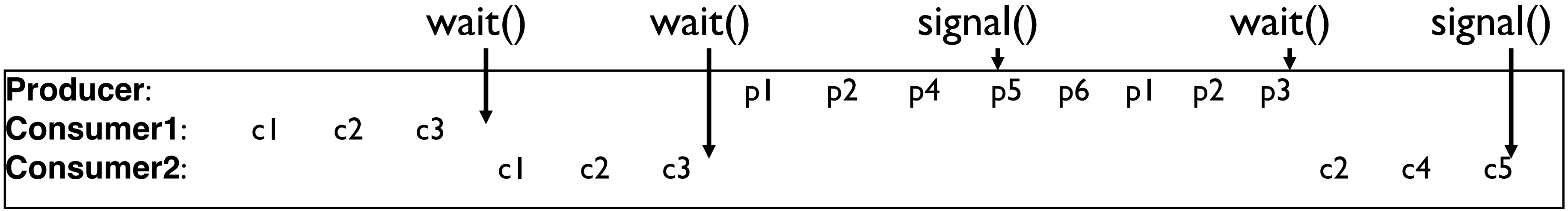
signal to producer

# 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) {
    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