CS 423
Operating System Design: Condition Variables and Semaphores
03/12

Ram Alagappan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

Logistics

Grades released for MP1

Change in schedule:

- Zoom lecture on Friday (will post a link on Piazza)
- Lecture hall needed for some other event
- People traveling early for break can attend on Zoom

AGENDA / LEARNING OUTCOMES

So far:

- Queue locks
- Condition variables

Today:

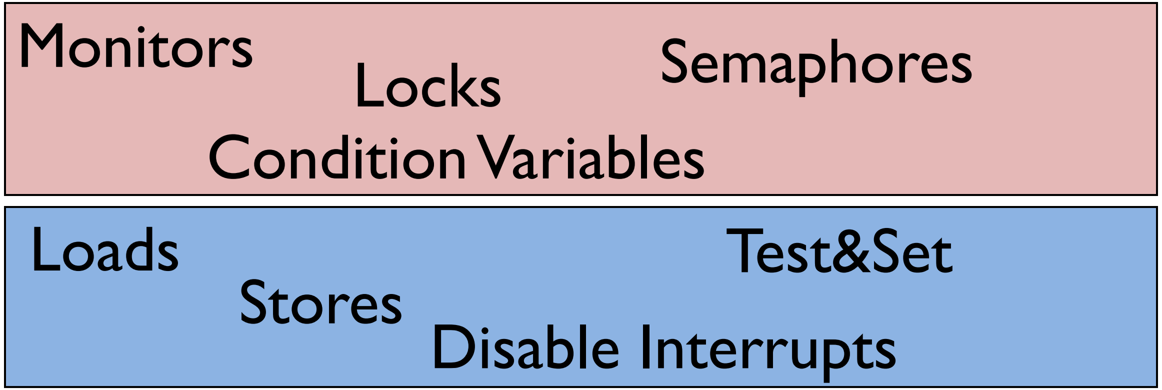
- Continue condition variables - producer consumer problem
- Semaphores

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



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

END RECAP

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

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:

```
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:

```
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?

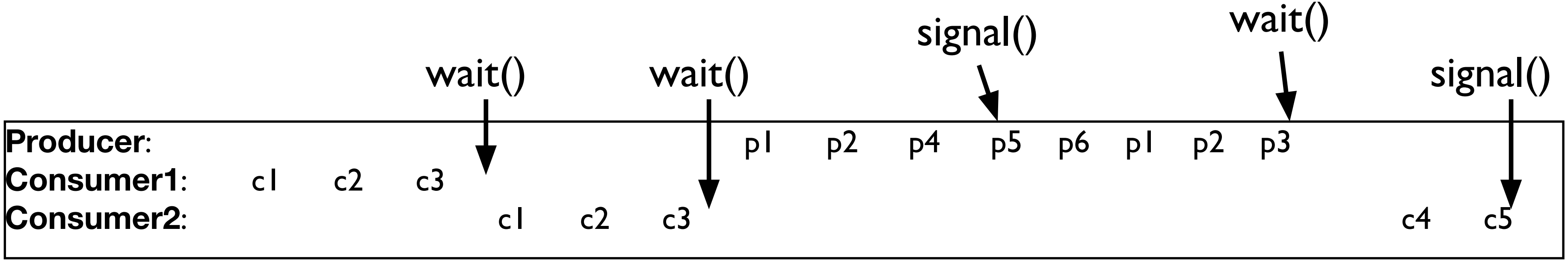
Can you find a problematic timeline with 2 consumers (still 1 producer)?

```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);
    }
}
```

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

INTRODUCING Semaphores

Condition variables have no state (other than waiting queue)

- Programmer must track additional state

Semaphores have state: track integer value

- State cannot be directly accessed by user program, but state determines behavior of semaphore operations

Equivalence

Semaphores are equally powerful to Locks+CVs

- what does this mean?

One might be more convenient, but that's not relevant

Equivalence means each can be built from the other

SEMAPHORE OPERATIONS

Allocate and Initialize

```
sem_t sem;
sem_init(sem_t *s, int initval) {
  s->value = initval;
}
```

User cannot read or write value directly after initialization

SEMAPHORE OPERATIONS

Wait or Test: sem_wait(sem_t*)

Decrements sem value by 1, Waits if value of sem is negative (< 0)

Signal or Post: sem_post(sem_t*)

Increment sem value by 1, then wake a single waiter if exists

Wait and Signal are atomic

Value of the semaphore, when negative = the number of waiting threads

BINARY Semaphore (LOCK)

```
typedef struct __lock_t {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {

}

void acquire(lock_t *lock) {

}

void release(lock_t *lock) {

}
```

sem_init(sem_t*, int initial)
sem_wait(sem_t*): Decrement, wait if value < 0
sem_post(sem_t*): Increment value
then wake a single waiter

Join with CV vs Semaphores

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

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

sem_t s;
sem_init(&s, ___-);

sem_wait(): Decrement, wait if value < 0
sem_post(): Increment value, then wake a single waiter

```
void thread_join() {
        sem_wait(&s);
}
```

```
void thread_exit() {
        sem_post(&s)
}
```

Producer/Consumer: Semaphores #1

Single producer thread, single consumer thread
Single shared buffer between producer and consumer

Use 2 semaphores

- emptyBuffer: Initialize to ________
- fullBuffer: Initialize to __________

Producer

```
while (1) {
    sem_wait(&emptyBuffer);
    Fill(&buffer);
    sem_post(&fullBuffer);
}
```

Consumer

```
while (1) {
    sem_wait(&fullBuffer);
    Use(&buffer);
    sem_post(&emptyBuffer);
}
```

Producer/Consumer: Semaphores #2

Single producer thread, single consumer thread
Shared buffer with **N elements** between producer and consumer
Use 2 semaphores

- emptyBuffer: Initialize to ____________
- fullBuffer: Initialize to _____________

```
Producer
i = 0;
while (1) {
    sem_wait(&emptyBuffer);
    Fill(&buffer[i]);
    i = (i+1)%N;
    sem_post(&fullBuffer);
}
```

```
Consumer
j = 0;
While (1) {
    sem_wait(&fullBuffer);
    Use(&buffer[j]);
    j = (j+1)%N;
    sem_post(&emptyBuffer);
}
```

Producer/Consumer: Semaphore #3

Final case:

- Multiple producer threads, multiple consumer threads
- Shared buffer with N elements between producer and consumer

Requirements

- Each consumer must grab unique filled element
- Each producer must grab unique empty element

Producer/Consumer: Multiple Threads

```
Producer
while (1) {
    sem_wait(&emptyBuffer);
    my_i = findempty(&buffer);
    Fill(&buffer[my_i]);
    sem_post(&fullBuffer);
}
```

```
Consumer
while (1) {
    sem_wait(&fullBuffer);
    my_j = findfull(&buffer);
    Use(&buffer[my_j]);
    sem_post(&emptyBuffer);
}
```

Are `my_i` and `my_j` private or shared? Where is mutual exclusion needed???

Producer/Consumer: Multiple Threads

Consider three possible locations for mutual exclusion
Which work??? Which is best???

```
Producer #1
    sem_wait(&mutex);
    sem_wait(&emptyBuffer);
    my_i = findempty(&buffer);
    Fill(&buffer[my_i]);
    sem_post(&fullBuffer);
    sem_post(&mutex);
```

```
Consumer #1
    sem_wait(&mutex);
    sem_wait(&fullBuffer);
    my_j = findfull(&buffer);
    Use(&buffer[my_j]);
    sem_post(&emptyBuffer);
    sem_post(&mutex);
```

Producer/Consumer: Multiple Threads

```
Producer #2
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    myi = findempty(&buffer);
    Fill(&buffer[myi]);
    sem_post(&mutex);
    sem_post(&fullBuffer);
```

```
Consumer #2
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    myj = findfull(&buffer);
    Use(&buffer[myj]);
    sem_post(&mutex);
    sem_post(&emptyBuffer);
```

Works, but limits concurrency:
Only 1 thread at a time can be using or filling different buffers

Producer/Consumer: Multiple Threads

```
Producer #3
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    myi = findempty(&buffer);
    sem_post(&mutex);
    Fill(&buffer[myi]);
    sem_post(&fullBuffer);
```

```
Consumer #3
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    myj = findfull(&buffer);
    sem_post(&mutex);
    Use(&buffer[myj]);
    sem_post(&emptyBuffer);
```

Works and increases concurrency; only finding a buffer is protected by mutex;
Filling or Using different buffers can proceed concurrently

Reader/Writer Locks

Let multiple reader threads grab lock (shared)

Only one writer thread can grab lock (exclusive)

- No reader threads
- No other writer threads

Let us see if we can understand code…

Reader/Writer Locks

```
typedef struct _rwlock_t {
    sem_t lock;
    sem_t writelock;
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 1);
    sem_init(&rw->writelock, 1);
}
```

Reader/Writer Locks

```
void rwlock_acquire_readlock(rwlock_t *rw) {
     sem_wait(&rw->lock);
     rw->readers++;
     if (rw->readers == 1)
         sem_wait(&rw->writelock);
     sem_post(&rw->lock);
}
void rwlock_release_readlock(rwlock_t *rw) {
     sem_wait(&rw->lock);
     rw->readers--;
     if (rw->readers == 0)
         sem_post(&rw->writelock);
     sem_post(&rw->lock);
}
rwlock_acquire_writelock(rwlock_t *rw) { sem_wait(&rw->writelock); }
rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()
    // who runs?
T4: acquire_readlock()
    // what happens?
T5: acquire_readlock()
    // where blocked?
T3: release_writelock()
    // what happens next?

Reader/Writer Locks

```
void rwlock_acquire_readlock(rwlock_t *rw) {
        sem_wait(&rw->lock);
        rw->readers++;
        if (rw->readers == 1)
            sem_wait(&rw->writelock);
        sem_post(&rw->lock);
}
void rwlock_release_readlock(rwlock_t *rw) {
        sem_wait(&rw->lock);
        rw->readers--;
        if (rw->readers == 0)
            sem_post(&rw->writelock);
        sem_post(&rw->lock);
}
rwlock_acquire_writelock(rwlock_t *rw) { sem_wait(&rw->writelock); }
rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T4: release_readlock()
    // what happens?
    // what's the problem?

Build Zemaphore!

```
Typedef struct {
    int value;
    cond_t cond;
    lock_t lock;
} zem_t;

void zem_init(zem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}
```

zem_wait(): Waits while value <= 0, Decrement
zem_post(): Increment value, then wake a single waiter

Build Zemaphore from LOCKs AND CV

```
zem_wait(zem_t *s) {
    lock_acquire(&s->lock);
    while (s->value <= 0)
        cond_wait(&s->cond);
    s->value--;
    lock_release(&s->lock);
}
```

```
zem_post(zem_t *s) {
    lock_acquire(&s->lock);
    s->value++;
    cond_signal(&s->cond);
    lock_release(&s->lock);
}
```

zem_wait(): Waits while value <= 0, Decrement
zem_post(): Increment value, then wake a single waiter

Semaphores

Semaphores are equivalent to locks + condition variables

- Can be used for both mutual exclusion and ordering

Semaphores contain **state**

- How they are initialized depends on how they will be used
- Init to 0: Join (1 thread must arrive first, then other)
- Init to N: Number of available resources

Sem_wait(): Decrement and then wait if < 0 (atomic)

Sem_post(): Increment value, then wake a single waiter (atomic)

Can use semaphores in producer/consumer and for reader/writer locks

NEXT STEPS

Next class: Deadlocks