CS 423
Operating System Design: Condition Variables
Mar 10

Ram Kesavan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

Logistics

MP2 out; Due (3/23) 11:59 pm CT
4Cr: No paper for next week
3/12 lecture: The 2 TAs will present 1 topic each

Mid-sem **anonymous** feedback:

https://forms.gle/NP7cTBAbPJfjC7G88

>85% participation => reward!

EXPECTATION: That students read recommended OSTEP chapters

AGENDA / LEARNING OUTCOMES

Last class:

- Critical Sections & Locks
- How are locks implemented?
  - Hardware: atomic instructions

Today:

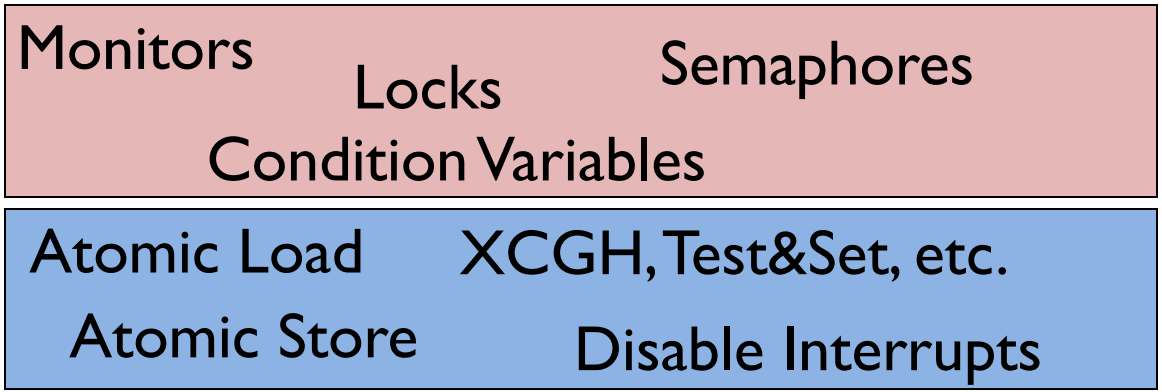
- Condition variables
  - Why do we need them? How are they implemented?
- Solving the Producer-Consumer problem

RECAP

Synchronization

Generic synchronization: for all models from previous slides

Goal: Learn how synchronization primitives are built (lib + OS)
With a little help from our friend…the hardware



LOCKING

Eval Criteria: Correctness, Progress, Bounded waits, Fairness, and Performance.

Implementation (evolution):

- Enable/disable interrupts
- Spin with load/store
- Atomic operations (XCHG)
- Block/sleep instead of spin on lock
- Add a queue: FIFO guaranteed

Spin-Wait vs Block/Sleep?

Time to context-switch? C
How long before lock is released? t

What is the best action when t < C? spin
When t > C? block/sleep
Problem: Requires knowledge of future

Hybrid approach: Two-phase Locks (not the distributed systems version!)

- Spin for some time (< C)
- Block after that

Concurrency Objectives

Mutual exclusion (A and B don't both run critical section)

- solved with *locks*

Ordering (B runs only after A has done its thing)

- solved with *condition variables* and *semaphores*

WHERE IS THE CODE?

User-space (pthread code)

https://codebrowser.dev/glibc/glibc/nptl/

pthread_mutex_[un]lock.c

pthread_cond_{wait|signal}.c

Within the Linux kernel:

/include/linux/mutex.h & /kernel/locking/mutex.c

SPINLOCKs: /include/linux/spinlock.h & /kernel/locking/spinlock.c

Isn't spinning bad? In very special cases, for very short durations, eg. interrupt handler

Uses semaphores instead of condition variables (we'll discuss those later)

/include/linux/semaphore.h, /kernel/locking/semaphore.c

END RECAP

ORDERING EXAMPLE: JOIN

```
pthread_t p1, p2;

pthread_create(&p1, NULL, mythread, "A");
pthread_create(&p2, NULL, mythread, "B");
// join waits for a pthread to exit
pthread_join(p1, NULL);
pthread_join(p2, NULL);

printf("main: children threads are done\n”);
return 0;
```

how to implement join()?

Condition Variables

Condition Variable: FIFO Queue of waiting threads

B waits for a signal on CV before it can continue

wait(CV, …)

don't spin while waiting

A sends signal to CV when it's time for ***B*** to continue

signal(CV, …)

CONDITION VARIABLES

wait(cond_t *cv, mutex_t *lock)

- the lock must be held before calling wait()
- adds caller to the FIFO queue—so fairness is guaranteed.
- puts caller to sleep & the releases the lock (atomically)
- returns when awoken, with lock reacquired

signal(cond_t *cv)

if (FIFO queue isn't empty)

wakes up one thread from queue

else does nothing (aka no-op, aka NOP)

Note: the mutex is an argument for **wait** but not **signal**

JOIN IMPLEMENTATION: ATTEMPT #1

Parent:

```
void thread_join() {
        mutex_lock(&m);        // P1
        cond_wait(&c, &m);     // P2
        mutex_unlock(&m);      // P3
}
```

Child:

```
void thread_exit() {
        mutex_lock(&m);        // C1
        cond_signal(&c);       // C2
        mutex_unlock(&m);      // C3
}
```

Example schedule:

```
Parent:        P1   P2                    P3

Child:                    C1   C2   C3
```

Global vars:

```
mutex_t  m;
cond_t   cv;
```

JOIN IMPLEMENTATION: ATTEMPT #1

Parent:

```
void thread_join() {
        mutex_lock(&m);        // P1
        cond_wait(&c, &m);     // P2
        mutex_unlock(&m);      // P3
}
```

Child:

```
void thread_exit() {
        mutex_lock(&m);        // C1
        cond_signal(&c);       // C2
        mutex_unlock(&m);      // C3
}
```

Example broken schedule:

C1, C2, C3, P1, P2→parent waits forever

Global vars:

```
mutex_t  m;
cond_t   cv;
```

RULE OF THUMB 1

Need to keep some state in addition to CV's!

CV's are used to signal threads when that 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);              // P1
        if (done == 0)               // P2
                cond_wait(&c, &m);   // P3
        mutex_unlock(&m);            // P4
}
```

Child:

```
void thread_exit() {
        done = 1;            // C1
        cond_signal(&c);// C2
}
```

Fixes previous broken schedule

Parent:		P1	P2	P3	P4
Child:	C1	C2			

Global vars:

```
mutex_t  m;
cond_t   cv;
boolean done = 0;
```

JOIN IMPLEMENTATION: ATTEMPT #2

Parent:

```
void thread_join() {
        mutex_lock(&m);              // P1
        if (done == 0)               // P2
                cond_wait(&c, &m);   // P3
        mutex_unlock(&m);            // P4
}
```

Child:

```
void thread_exit() {
        done = 1;            // C1
        cond_signal(&c);// C2
}
```

An example broken schedule:

P1, P2, C1, C2, P3 → parent waits forever

Global vars:

```
mutex_t  m;
cond_t    cv;
boolean done = 0;
```

JOIN IMPLEMENTATION: CORRECT

Parent:

```
void thread_join() {
        mutex_lock(&m);              // P1
        if (done == 0)               // P2
                cond_wait(&c, &m);   // P3
        mutex_unlock(&m);            // P4
}
```

Child:

```
void thread_exit() {
        mutex_lock(&m);      // C1
        done = 1;            // C2
        cond_signal(&c);     // C3
        mutex_unlock(&m);    // C4
}
```

Parent: P1 P2 P3 P4

Child: C1 C2 C3 C4

Global vars:

```
mutex_t  m;
cond_t   cv;
boolean done = 0;
```

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
- (3rd rule of thumb) Always use a while instead of an if when checking cond
  - To avoid getting fooled by "spurious wakeups"

```
mutex_lock(&m);
while (done == 0) {
    cond_wait(&c, &m);
}
mutex_unlock(&m);
```

❌

```
mutex_lock(&m);
if (done == 0) {
    cond_wait(&c, &m);
}
mutex_unlock(&m);
```

PRODUCER/CONSUMER PROBLEM

EXAMPLE: UNIX PIPES

```
int pipefd[2]; // read from pipe[0], write to pipe[1]
if (pipe(pipefd) == -1) {
        perror("failed to create pipe");
        exit(EXIT_FAILURE);
}

// writer thread
char msgbuf[BUFSIZ];
snprintf(msgbuf, BUFSIZ, "Hello world\n");
write(pipefd[1], msgbuf, strlen(msgbuf) + 1);

// reader thread
int nbytes;
nbytes = read(pipefd[0], msgbuf, BUFSIZ - 1);
```

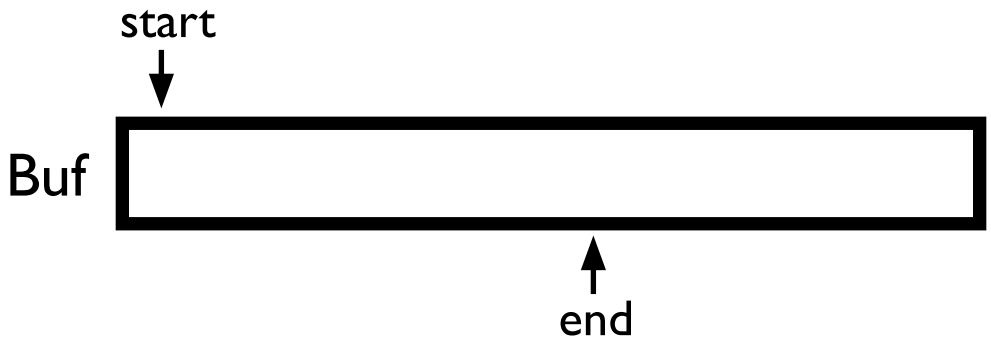
EXAMPLE: UNIX PIPES

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

Producer/Consumer Example

We'll start with easy case:

- 1 producer thread
- 1 consumer thread
- shared buffer with just 1 slot to fill/consume

Numfull: number of slots currently filled

Max: max number of slots in the buffer = 1

ATTEMPT#1

Global vars:

```
mutex_t  m;
cond_t   cv;
int numfull = 0;
```

Thread 1:

```
void *producer(void *arg) {
        while (true) {
                mutex_lock(&m);
                if (numfull == max)
                        cond_wait(&cv, &m);
                produce_once();
                cond_signal(&cv);
                mutex_unlock(&m);
        }
}
```

Start with numfull = 0

Thread 2:

```
void *consumer(void *arg) {
        while (true) {
                mutex_lock(&m);
                if (numfull == 0)
                        cond_wait(&cv, &m);
                int tmp = consume_once();
                cond_signal(&cv);
                mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

WHAT ABOUT 2 CONSUMERS?

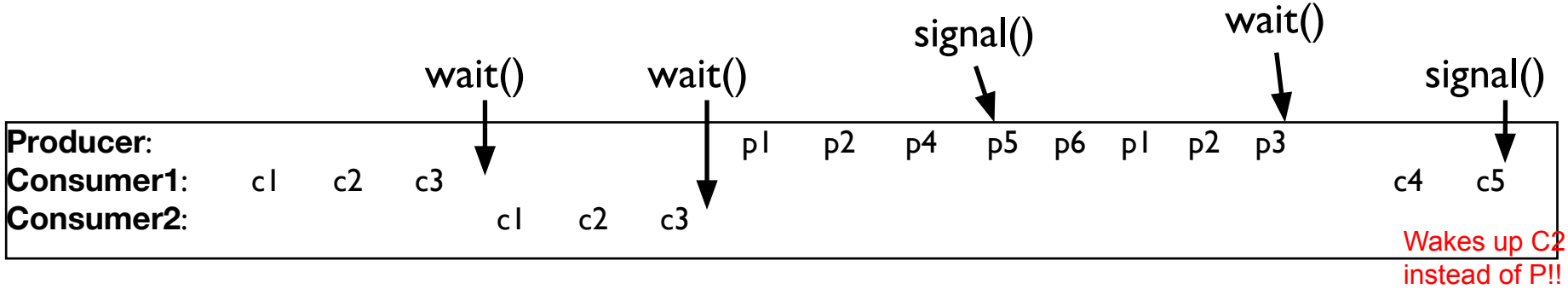
Can you find a problematic timeline with 2 consumers (still 1 producer)?

```
void *producer(void *arg) {
    while (1) {
        mutex_lock(&m);          // p1
        if (numfull == max)      //p2
            cond_wait(&cv, &m);//p3
        produce_once();          // p4
        cond_signal(&cv);        //p5
        mutex_unlock(&m);        //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        mutex_lock(&m);          // c1
        if (numfull == 0)        // c2
            cond_wait(&cv, &m); // c3
        int tmp = consume_once(); // c4
        cond_signal(&cv);        // c5
        mutex_unlock(&m);        // c6
    }
}
```

```
void *producer(void *arg) {
    while (1) {
        mutex_lock(&m);           // p1
        if (numfull == max)       //p2
            cond_wait(&cv, &m);   //p3
        produce_once();           // p4
        cond_signal(&cv);         //p5
        mutex_unlock(&m);         //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        mutex_lock(&m);           // c1
        if (numfull == 0)         // c2
            cond_wait(&cv, &m);   // c3
        int tmp = consume_once();// c4
        cond_signal(&cv);         // c5
        mutex_unlock(&m);         // c6
    }
}
```



HOW TO WAKE UP THE CORRECT THREAD?

Wake all the threads!

pthread_cond_broadcast(cond_t *cv);

instead of

pthread_cond_signal(cond_t *cv);

Better solution (usually): use two condition variables

Fairness is lost with broadcast

ATTEMPT#2: 2 CVs ARE BETTER THAN 1

```
void *producer(void *arg) {
    while (1) {
        mutex_lock(&m);           // p1
        if (numfull == max)       //p2
            cond_wait(&empty, &m);//p3
        produce_once();           // p4
        cond_signal(&fill);       //p5
        mutex_unlock(&m);         //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        mutex_lock(&m);           // c1
        if (numfull == 0)         // c2
            cond_wait(&fill, &m); // c3
        int tmp = consume_once(); // c4
        cond_signal(&empty);      // c5
        mutex_unlock(&m);         // c6
    }
}
```

Solves the previous problem…
But can you still find a bad schedule?
Stay with 1 producer & 2 consumers

Global vars:

```
mutex_t  m;
cond_t   full, empty;
int numfull = 0;
```

Producer/Consumer: Two CVs

```
void *producer(void *arg) {
    while (1) {
        mutex_lock(&m);           // p1
        if (numfull == max)       //p2
            cond_wait(&empty, &m);//p3
        produce_once();           // p4
        cond_signal(&fill);       //p5
        mutex_unlock(&m);         //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        mutex_lock(&m);           // c1
        if (numfull == 0)         // c2
            cond_wait(&fill, &m); // c3
        int tmp = consume_once(); // c4
        cond_signal(&empty);      // c5
        mutex_unlock(&m);         // c6
    }
}
```

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.

ATTEMPT#3: Two CVs and WHILE

```
void *producer(void *arg) {
    while (1) {
        mutex_lock(&m);           // p1
        while (numfull == max)    //p2
            cond_wait(&empty, &m);//p3
        produce_once();           // p4
        cond_signal(&fill);       //p5
        mutex_unlock(&m);         //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        mutex_lock(&m);           // c1
        while (numfull == 0)      // c2
            cond_wait(&fill, &m); // c3
        int tmp = consume_once(); // c4
        cond_signal(&empty);      // c5
        mutex_unlock(&m);         // c6
    }
}
```

No concurrent access to shared state

Every time lock is acquired (or thread awakens), assumptions are reevaluated

A consumer will get to run after every produce_once()

A producer will get to run after every consume_once()

GOOD RULE OF THUMB 3

Whenever a lock is acquired, recheck assumptions about state!

Another thread may grab lock (between signal and wake up)

Note: some libs have “spurious wakeups”--may wake multiple waiting threads

Good stress test for correctness: change your signal to broadcast; code should run despite loss of fairness guarantees

HOARE VS MESA SEMANTICS

Mesa: used widely (cheaper & easier to implement)

- Signaler holds on to lock & the CPU/core
- Signal moves a waiting thread to READY/RUNNABLE state
- Not necessary that the signaled waiter runs next

Conditions may change by the time waiter runs

Hoare: almost never used (expensive & complex to implement)

- Signal gives lock & the CPU/core to waiting thread
- Waiter runs when woken up by the signaler
- When waiter finishes, it hands lock + CPU/core back to signaler

Conditions remain unchanged because waiter runs immediately after

POP QUIZ #3

https://forms.gle/zgyPoR8wQa8rH5hcA

Mid-sem **anonymous** feedback:
https://forms.gle/NP7cTBAbPJfjC7G88

>85% participation => reward!

Piazza post with this link & reward

SUMMARY: 3 RULES OF THUMB FOR CVS

1. Need to keep some state in addition to CV's

2. Always wait or signal with lock held

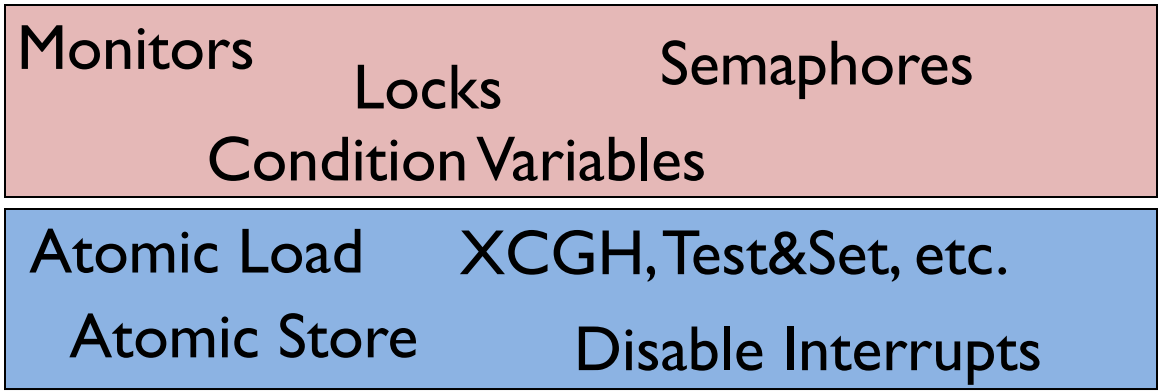
3. Always recheck state when thread wakes up from a wait

   Use a while instead of an if

Synchronization

Generic synchronization: for all models from previous slides

Goal: Learn how synchronization primitives are built (lib + OS)
With a little help from our friend…the hardware



NEXT: 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