Announcements

Project 2:
- Part 2a will be graded this week
- Part 2b take longer since we compare all graphs…

Project 3: Shared memory segments
- Linux: use shmget and shmat across server + client processes
  - semaphores for locks; catch ctrl-C to do clean-up
  - Can work with a project partner (request new one if desired)
  - No videos
- Xv6: Implement combination of shmgetat() – Watch video!
- Due Wed 11/02 by 9:00 pm

Class feedback for mid-course evaluations
- Receive email about survey to fill out until this Friday

Today's Reading: Chapter 31

UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

Semaphores

Questions answered in this lecture:

Review: How to implement join with condition variables?

Review: How to implement producer/consumer with condition variables?

What is the difference between **semaphores** and condition variables?

How to implement a **lock** with semaphores?

How to implement semaphores with locks and condition variables?

How to implement **join** and producer/consumer with semaphores?

Review: **Processes** vs Threads

```
int a = 0;

int main() {
  fork();
  a++;
  fork();
  a++;
  if (fork() == 0) {
    printf("Hello!\n");
  } else {
    printf("Goodbye!\n");
  }
  a++;
  printf("a is %d\n", a);
}
```

How many times will "Hello!\n" be displayed?

4

What will be the **final** value of "a" as displayed by the final line of the program?

3

Review: Processes vs **Threads**

```
volatile int balance = 0;

void *mythread(void *arg) {
    int result = 0;
    result = result + 200;
    balance = balance + 200;
    printf("Result is %d\n", result);
    printf("Balance is %d\n", balance);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final Balance is %d\n", balance);
}
```

How many total threads are part of this process?

3

When thread p1 prints "Result is %d\n", what value of `result` will be printed?

200. 'result' is a local variable allocated on the stack; each thread has its own private copy which only it increments, therefore there are no race conditions.

When thread p1 prints "Balance is %d\n", what value of `balance` will be printed?

Unknown. balance is allocated on the heap and shared between the two threads that are each accessing it without locks; there is a race condition.

Sample Homework: HW-ThreadsIntro

./x86.py -p looping-race-nolock.s -t 2 -r -i 3

```
# assumes %bx has loop count in it

.main
.top
mov 2000, %ax # get the value at the address
add $1, %ax   # increment it
mov %ax, 2000 # store it back


# see if we're still looping
sub $1, %bx
test $0, %bx
jgt .top
halt
```

Looping-race-nolocks.s (addr 2000 has 0)

```
Thread 0                      Thread 1
1000 mov 2000, %ax
1001 add $1, %ax
------ Interrupt ------       ------ Interrupt ------
                              1000 mov 2000, %ax
------ Interrupt ------       ------ Interrupt ------
1002 mov %ax, 2000
------ Interrupt ------       ------ Interrupt ------
                              1001 add $1, %ax
                              1002 mov %ax, 2000
                              1003 sub  $1, %bx
------ Interrupt ------       ------ Interrupt ------
1003 sub  $1, %bx
------ Interrupt ------       ------ Interrupt ------
                              1004 test $0, %bx
------ Interrupt ------       ------ Interrupt ------
1004 test $0, %bx
------ Interrupt ------       ------ Interrupt ------
                              1005 jgt .top
                              1000 mov 2000, %ax
------ Interrupt ------       ------ Interrupt ------
1005 jgt .top
1000 mov 2000, %ax
1001 add $1, %ax
------ Interrupt ------       ------ Interrupt ------
                              1001 add $1, %ax
                              1002 mov %ax, 2000
------ Interrupt ------       ------ Interrupt ------
1002 mov %ax, 2000
```

```
Contents of ax = [redacted]
Contents of ax = [redacted]

Contents of ax = [redacted]

50) Contents of addr = [redacted]

Contents of ax = [redacted]
51) Contents of addr = [redacted]

Contents of ax = [redacted]

Contents of ax = [redacted]
Contents of ax = [redacted]

Contents of ax = [redacted]
52) Contents of addr = [redacted]

53) Contents of addr = [redacted]
```

Homework: HW-ThreadsLock

```
[HW-ThreadsLocks] % more ticket.s
.var ticket
.var turn
.var count

.main
.top

.acquire
mov $1, %ax
fetchadd %ax, ticket  # grab a ticket (keep it in dx)
.tryagain
mov turn, %cx         # check if it's your turn
test %cx, %ax
jne .tryagain

# critical section
mov  count, %ax       # get the value at the address
add  $1, %ax          # increment it
mov  %ax, count       # store it back

# release lock
mov $1, %ax
fetchadd %ax, turn

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

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*

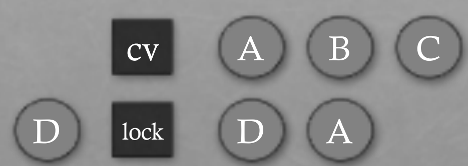
Condition Variables

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



signal(cv) - what happens?

release(lock) - what happens?

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() {
        done = 1;                // b
        Mutex_lock(&m);          // a
        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

Why okay to have "if" instead of "while"?

Producer/Consumer Problem

Producers generate data (like pipe writers)

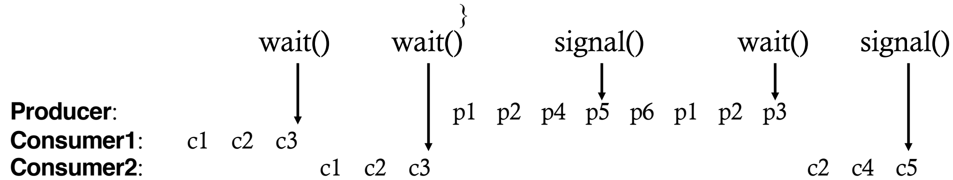
Consumers grab data and process it (like pipe readers)

Use condition variables to:
make producers wait when buffers are full
make consumers wait when there is nothing to consume

Broken Implementation of Producer Consumer

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m); // p1
        while(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
        while(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
    }
}
```



does last signal wake producer or consumer2?

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);  // c1
        if (numfull == 0) // c2
            Cond_wait(&fill, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&empty); // c5
        Mutex_unlock(&m);  // c6
    }
}
```

Is this correct? Can you find a bad schedule?

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:		p1 p2 p4 p5 p6		
Consumer1:	c1 c2 c3			c4! ERROR
Consumer2:			c1 c2 c4 c5 c6	

CV Rule of Thumb 3

Whenever a lock is acquired, recheck assumptions about state!
  Use “while” intead of “if”

Possible for another thread to grab lock between signal and wakeup from wait
- Difference between Mesa (practical implementation) and Hoare (theoretical) semantics
- Signal() simply makes a thread runnable, does not guarantee thread run next

Note that some libraries also have “spurious wakeups”
- May wake multiple waiting threads at signal or at any time

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);
    }
}
```

Is this correct? Can you find a bad schedule?

Correct!
- 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()`

Summary: Rules of Thumb for CVs

Keep state in addition to CV's

Always do wait/signal with lock held

Whenever thread wakes from waiting, recheck state

UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

SEMAPHORES

Questions answered in this lecture:

~~Review: How to implement join with condition variables?~~

~~Review: How to implement producer/consumer with condition variables?~~

What is the difference between **semaphores** and condition variables?

How to implement a **lock** with semaphores?

How to implement semaphores with locks and condition variables?

How to implement **join** and producer/consumer with semaphores?

CONDITION VARIABLES VS 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

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

Wait or Test (sometime P() for Dutch word)

Waits until value of sem is > 0, then decrements sem value

Signal or Increment or Post (sometime V() for Dutch)

Increment sem value, then wake a single waiter (so it can check)

wait and post are atomic

Join with CV vs Semaphores

CVs:

```
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
}
```

Semaphores:

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

```
sem_t s;
sem_init(&s, ???);
```

Initialize to 0 (so sem_wait() must wait…)

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

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

EQUIVALENCE CLAIM

Semaphores are equally powerful to Locks+CVs

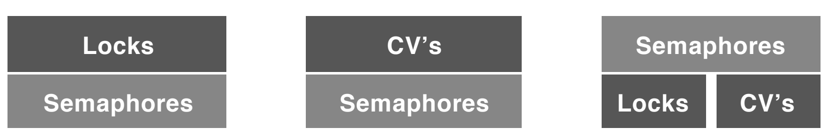
- what does this mean?

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

Equivalence means each can be built from the other

PROOF STEPS

Want to show we can do these three things:



Build Lock from Semaphore

```
typedef struct __lock_t {
// whatever data structs you need go here
} lock_t;

void init(lock_t *lock) {
}

void acquire(lock_t *lock) {
}

void release(lock_t *lock) {
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Locks
Semaphores

Build Lock from Semaphore

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

void init(lock_t *lock) {
  sem_init(&lock->sem, ??);
}
void acquire(lock_t *lock) {
  sem_wait(&lock->sem);
}
void release(lock_t *lock) {
  sem_post(&lock->sem);
}
```

1 → 1 thread can grab lock

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Locks
Semaphores

Building CV's over Semaphores

Possible, but really hard to do right

Read about Microsoft Research's attempts:

http://research.microsoft.com/pubs/64242/ImplementingCVs.pdf

Build Semaphore from Lock and CV

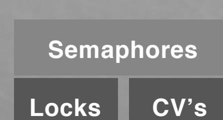
```
Typedef struct {
    // what goes here?


} sem_t;

Void sem_init(sem_t *s, int value) {
    // what goes here?


}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter



Build Semaphore from Lock and CV

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

Void sem_init(sem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}
```

Semaphores

Locks | CV's

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Build Semaphore from Lock and CV

```
Sem_wait{sem_t *s) {
    // what goes here?

}
```

```
Sem_post{sem_t *s) {
    // what goes here?

}
```

Semaphores

Locks | CV's

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Build Semaphore from Lock and CV

```
Sem_wait{sem_t *s) {
    lock_acquire(&s->lock);
    // this stuff is atomic


    lock_release(&s->lock);
}
```

```
Sem_post{sem_t *s) {
    lock_acquire(&s->lock);
    // this stuff is atomic

    lock_release(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Semaphores

Locks | CV's

Build Semaphore from Lock and CV

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

```
Sem_post{sem_t *s) {
    lock_acquire(&s->lock);
    // this stuff is atomic

    lock_release(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Semaphores

Locks | CV's

BUILD SEMAPHORE FROM LOCK AND CV

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

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

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Semaphores
Locks | CV's

BUILD SEMAPHORE FROM LOCK AND CV

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

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

What if sem initialized to 2?
What if sem initialized to -1?

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Semaphores
Locks | CV's

BREAK

What have you done that you are most proud of?

PRODUCER/CONSUMER: SEMAPHORES #1

Simplest case:

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

Requirements

- Consumer must wait for producer to fill buffer
- Producer must wait for consumer to empty buffer (if filled)

Requires 2 semaphores

- emptyBuffer: Initialize to ??? 1 → 1 empty buffer; producer can run 1 time first
- fullBuffer: Initialize to ??? 0 → 0 full buffers; consumer can run 0 times first

Producer

```
While (1) {
   sem_wait(&emptyBuffer);
   Fill(&buffer);
   sem_signal(&fullBuffer);
}
```

Consumer

```
While (1) {
   sem_wait(&fullBuffer);
   Use(&buffer);
   sem_signal(&emptyBuffer);
}
```

Producer/Consumer: Semaphores #2

Next case: **Circular Buffer**

- Single producer thread, single consumer thread
- Shared buffer with **N** elements between producer and consumer

Requires 2 semaphores

- emptyBuffer: Initialize to ??? N → N empty buffers; producer can run N times first
- fullBuffer: Initialize to ??? 0 → 0 full buffers; consumer can run 0 times first

Producer

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

Consumer

```
j = 0;
While (1) {
   sem_wait(&fullBuffer);
   Use(&buffer[j]);
   j = (j+1)%N;
   sem_signal(&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
- **Why will previous code (shown below) not work???**

Producer

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

Consumer

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

Are i and j private or shared? Need each producer to grab unique buffer

Producer/Consumer: Multiple Threads

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

Track state of each element (FULL, EMPTY, FILLING, USING)

Producer

```
While (1) {
    sem_wait(&emptyBuffer);
    myi = findempty(&buffer);
    Fill(&buffer[myi]);
    sem_signal(&fullBuffer);
}
```

Consumer

```
While (1) {
    sem_wait(&fullBuffer);
    myj = findfull(&buffer);
    Use(&buffer[myj]);
    sem_signal(&emptyBuffer);
}
```

Are myi and myj 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);
myi = findempty(&buffer);
Fill(&buffer[myi]);
sem_signal(&fullBuffer);
sem_signal(&mutex);
```

Consumer #1

```
sem_wait(&mutex);
sem_wait(&fullBuffer);
myj = findfull(&buffer);
Use(&buffer[myj]);
sem_signal(&emptyBuffer);
sem_signal(&mutex);
```

Problem: Deadlock at mutex (e.g., consumer runs first; won’t release mutex)

Producer/Consumer: Multiple Threads

Consider three possible locations for mutual exclusion

Which work??? Which is best???

Producer #2

```
sem_wait(&emptyBuffer);
sem_wait(&mutex);
myi = findempty(&buffer);
Fill(&buffer[myi]);
sem_signal(&mutex);
sem_signal(&fullBuffer);
```

Consumer #2

```
sem_wait(&fullBuffer);
sem_wait(&mutex);
myj = findfull(&buffer);
Use(&buffer[myj]);
sem_signal(&mutex);
sem_signal(&emptyBuffer);
```

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

Producer/Consumer: Multiple Threads

Consider three possible locations for mutual exclusion

Which work??? Which is best???

Producer #3

```
sem_wait(&emptyBuffer);
sem_wait(&mutex);
myi = findempty(&buffer);
sem_signal(&mutex);
Fill(&buffer[myi]);
sem_signal(&fullBuffer);
```

Consumer #3

```
sem_wait(&fullBuffer);
sem_wait(&mutex);
myj = findfull(&buffer);
sem_signal(&mutex);
Use(&buffer[myj]);
sem_signal(&emptyBuffer);
```

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

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 1: Mutex
- Init to 0: Join (1 thread must arrive first, then other)
- Init to N: Number of available resources

Sem_wait(): Waits until value > 0, then decrement (atomic)

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

Can use semaphores in producer/consumer relationships and for reader/writer locks (next lecture)

Reader/Writer Locks

Goal:

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()
T4: acquire_readlock()
T5: acquire_readlock()  // ???
T3: release_writelock()
// what happens???
```

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 1: Mutex
- Init to 0: Join (1 thread must arrive first, then other)
- Init to N: Number of available resources

Sem_wait(): Waits until value > 0, then decrement (atomic)

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

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