UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

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

EXAM 2: REVIEW

Questions answered in this lecture:

What are some useful things to remember about concurrency? And I/O devices?

ANNOUNCEMENTS

Project 2: Graded in Learn@UW

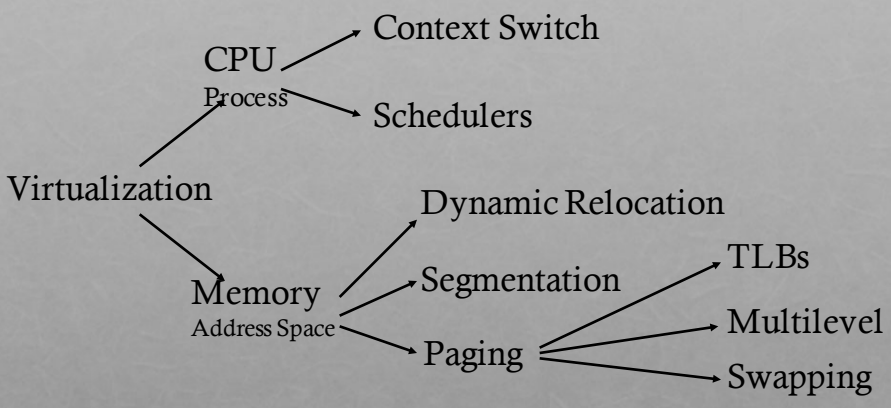
Project 3: Being graded

Project 4: Due in 2 weeks; work with a partner! Lab Hours!

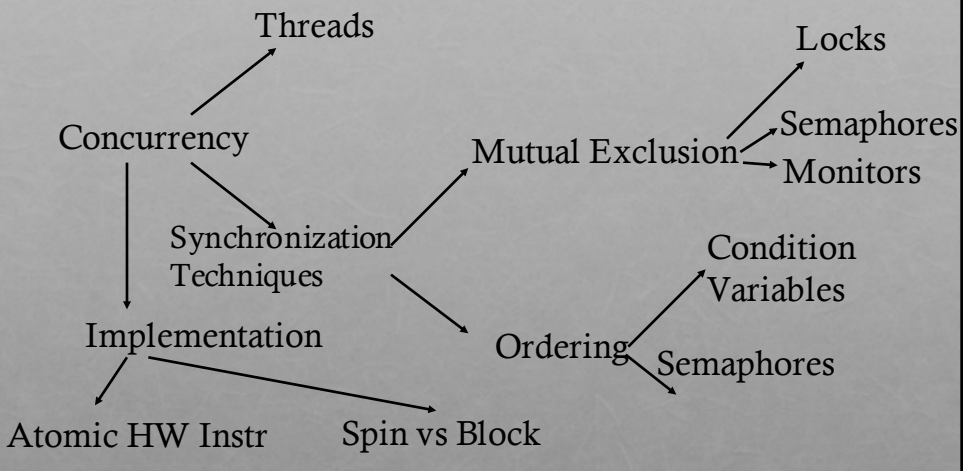
Exam – Tomorrow evening (Wed 11/9)

- Two hours – 7:15 – 9:15 pm in Humanities 3650
- Bring #2 pencils and student id
- All multiple choice
- Covers everything so far in course:
  - Lectures + Reading + Homework + Projects 1-3
  - 20% Old Material : Virtualization
  - New Material: Concurrency + I/O Devices, Disks, I/O Scheduling
  - Look over sample exams

REVIEW: EASY PIECE 1



REVIEW: EASY PIECE 2



WHAT QUESTIONS DID YOU ASK?

QUESTION: GLOBAL VARIABLES

Where are global variables stored in an address space? I know that local variables are stored in the "stack" where as "heap" contains malloced data.

Common to have two other segments:

- Code
- Data: Uninitialized and initialized - STATIC SIZE
  - Do not place between heap and stack!

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]
```

QUESTION: ATOMIC TEST_AND_SET

What happens when two processors are executing the same test_and_set instruction at the same time ?

though the operation inside test_and_set is atomic for one processor, the two processors can do this at the same time and both get the lock. What is stopping both of them getting the lock ?

Is it that the memory bus is locked by one processor till this instruction (test_and_set) completes (i.e., locked for load, operation, store) ?

XCHG: ATOMIC EXCHANGE, OR TEST-AND-SET

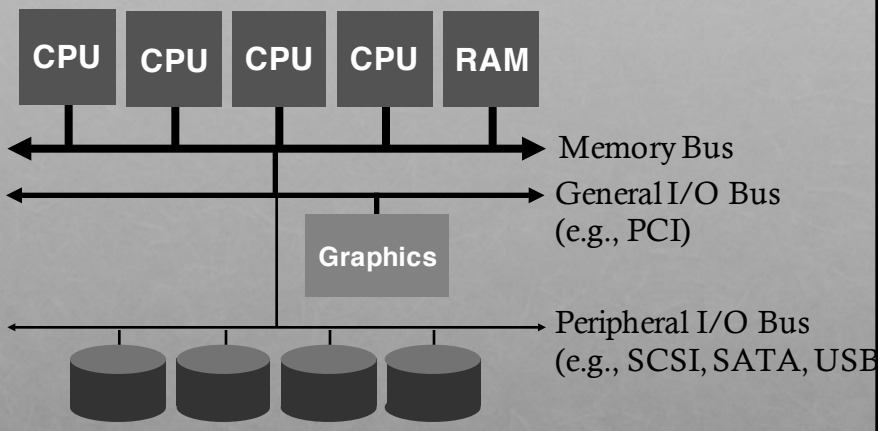
```
// xchg(int  *addr,  int  newval)
// ATOMICALLY  return  what  was  pointed  to  by  addr
// AT  THE  SAME  TIME,  store  newval  into  addr

int  xchg(int  *addr,  int  newval)  {
  int  old  =  *addr;
  *addr  =  newval;
  return  old;
}
```

Need hardware support

```
static inline uint
xchg(volatile unsigned int *addr, unsigned int newval)
{
    uint result;
    asm volatile("lock; xchgl %0, %1" :
                 "+m" (*addr), "=a" (result) :
                 "1" (newval) : "cc");
    return result;
}
```

HARDWARE PICTURE



Why use hierarchical buses?

BLOCK WHEN WAITING: FINAL CORRECT LOCK

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

setpark() fixes race condition

Park() does not block if unpark() occurred after setpark()

```
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;
}
```

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 from cv signal, 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

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()`

QUESTION: DIFFERENT DEFINITIONS OF SEMAPHORES?

Book starts by defining semaphores as:

```
Sem_wait() {
        Decrement value of semaphore by 1
        Wait if value of semaphore is negative
}
Sem_post() {
        increment value of semaphore by 1
        if thread waiting, wake one
}
```

Lecture (and end of chapter) Wait or Test
Waits until value of sem is > 0, then decrements sem value
Signal or Increment or Post
Increment sem value, then wake a single waiter

QUESTION: SEMAPHORES VS CV'S

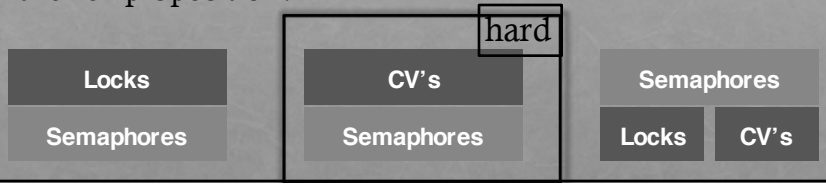
A line from the book "building semaphores out of condition variables is challenging".

Why is it so?

Can't I just use semvalue as 0 so that the semaphore behaves as a condition variable?

Incorrect statement;

Book says "Building CVs out of semaphores is a much trickier proposition."



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 from cv signal, 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

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

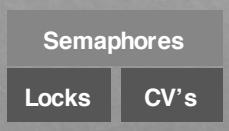
wait and post are atomic

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);
}
```

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

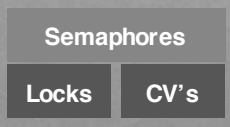


BUILD SEMAPHORE FROM LOCK AND CV

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



BACK: SEMAPHORES VS CONDITION VARIABLES

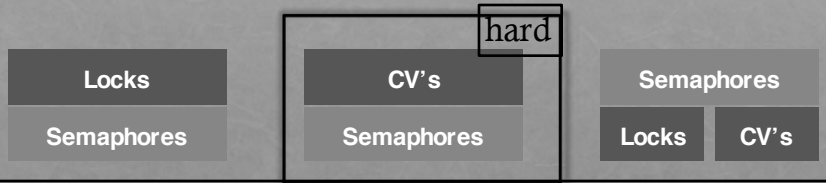
A line from the book "building semaphores out of condition variables is challenging".

Why is it so?

Can't I just use semvalue as 0 so that the semaphore behaves as a condition variable? (Assume trying to build cv)

Incorrect statement;

Book says "Building locks and CVs out of semaphores is a much trickier proposition."



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)
}
```

BUILD CV FROM SEMAPHORE???

```
Typedef struct {
    sem_t sem;
} cv_t;
Void cv_init(cv_t *cv) {
    sem_init(cv->sem, 0);
}

Cond_wait(cv_t *cv, lock_t *l) {
    mutex_unlock(l);
    Sem_wait(cv->sem);
    mutex_lock(l);
}
Cond_signal(cv_t *cv) {
    sem_post(cv->sem);
}
```

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_init(&cv->sem, 0);
void thread_join() {
        Mutex_lock(&m);          // w
        if (done == 0) {         // x
            Mutex_unlock(&m);
            Sem_wait(cv->sem);
            Mutex_lock(&m);
        }
        Mutex_unlock(&m);        // z
}
```

```
void thread_exit() {
        Mutex_lock(&m);          // a
        done = 1;                // b
        Sem_post(cv->sem);       // c
        Mutex_unlock(&m);        // d
}
```

PROBLEM: SHOULD BE GENERAL SOLUTION

How should this behave?

```
                                  cond_signal(cv);
cond_wait(cv);
```

How does this behave?

```
                                  sem_init(&cv->sem, 0);
                                  sem_post(cv->sem);
mutex_unlock(&m);
sem_wait(cv->sem);
mutex_lock(&m);
```

BACK: SEMAPHORES VS CONDITION VARIABLES

A line from the book "building semaphores out of condition variables is challenging".

Why is it so?

Can't I just use semvalue as 0 so that the semaphore behaves as a condition variable? (Assume trying to build cv)

Incorrect statement;

Book says "Building locks and CVs out of semaphores is a much trickier proposition."

QUESTION: DINING PHILOSOPHERS?

DINING PHILOSOPHERS: HOW TO APPROACH

Guarantee two goals

- **Safety:** Ensure nothing bad happens (don't violate constraints of problem)
- **Liveness:** Ensure something good happens when it can (make as much progress as possible)

Introduce state variable for each philosopher i

```
state[i] = THINKING, HUNGRY, or EATING
```

Safety:

No two adjacent philosophers eat simultaneously

```
for all i: !(state[i]==EATING  && state[i+1%5]==EATING)
```

Liveness:

Not the case that a philosopher is hungry and his neighbors are not eating

```
for all i: !(state[i]==HUNGRY  &&
       (state[i+4%5]!=EATING  && state[i+1%5]!=EATING))
```

```
sem_t mayEat[5]; // how to initialize?
sem_t mutex; // how to init?
int state[5] = {THINKING};
take_chopsticks(int  i) {
     wait(&mutex);  // enter critical  section
     state[i] = HUNGRY;
     testSafetyAndLiveness(i);  // check if I can run
     signal(&mutex);  // exit critical section
     wait(&mayEat[i]);
}
put_chopsticks(int  i) {
     wait(&mutex);  // enter critical  section
     state[i] = THINKING;
     test(i+1  %5); // check if neighbor  can run now
     test(i+4  %5);
     signal(&mutex);  // exit critical section
}
testSafetyAndLiveness(int  i) {
     if(state[i]==HUNGRY&&state[i+4%5]!=EATING&&state[i+1%5]!=EATING)  {
            state[i] = EATING;
            signal(&mayEat[i]);
     } }
```

QUESTION: READER-WRITER LOCKS

Do we assume the entire function "rwlock_acquire_readlock" is atomic?

If any context switch happens during the execution of this function then a proper read-write lock will not be achieved.

READER/WRITER LOCKS

```
1 typedef struct _rwlock_t {
2     sem_t lock;
3     sem_t writelock;
4     int readers;
5 } rwlock_t;
6
7 void rwlock_init(rwlock_t *rw) {
8     rw->readers = 0;
9     sem_init(&rw->lock, 1);
10    sem_init(&rw->writelock, 1);
11 }
12
```

READER/WRITER LOCKS

```
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14       sem_wait(&rw->lock);
15       rw->readers++;
16       if (rw->readers == 1)
17          sem_wait(&rw->writelock);
18       sem_post(&rw->lock);
19 }
21 void rwlock_release_readlock(rwlock_t *rw) {
22       sem_wait(&rw->lock);
23       rw->readers--;
24       if (rw->readers == 0)
25          sem_post(&rw->writelock);
26       sem_post(&rw->lock);
27 }
29  rwlock_acquire_writelock(rwlock_t *rw) {  sem_wait(&rw->writelock);
31 rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

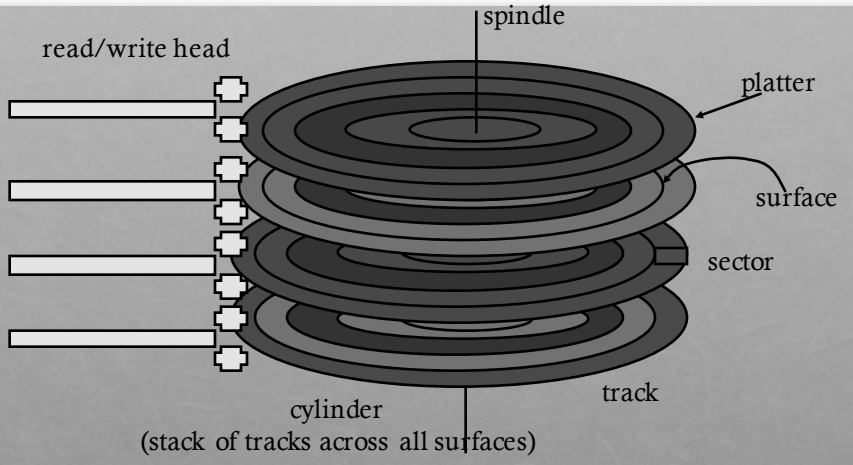
T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T6: acquire_readlock()
T1: release_readlock()
T6: release_readlock()
T1: acquire_readlock()
T2: acquire_readlock() // ???
T3: release_writelock()
// what happens???

DISKS

What are cylinders?

Why is average rotation distance ½? Does it ever change?

DISK TERMINOLOGY



WHAT IS ELEVATOR ALG?

Elevator Algorithm:

- Sweep back and forth, from one end of disk other, serving requests as pass that cylinder
- Sorts by cylinder number; ignores rotation delays
- Prevents starvation

Disadvantage?

Not too fair…

Better: C-SCAN (circular scan)

- Only sweep in one direction

PREVIOUS MIDTERM

Assume you have the following code for accessing a shared-buffer that contains max elements (for some very large value of max). Assume multiple producer and multiple consumer threads access these routines concurrently. Assume the initial state is that the mutex is not held and that all buffers are empty. Assume the semaphore empty is initialized to 0 and fill is initialized to max. Assume numfull is initialized to 0.

```
void *producer(void *arg) {
        Mutex_lock(&m); // p1
        if (numfull == max) // p2
                sema_wait(&empty); // p3
        do_fill(i); //updates numfull // p4
        sema_post(&fill); // p5
        Mutex_unlock(&m); // p6 }
void *consumer(void *arg) {
        Mutex_lock(&m); // c1
        if (numfull == 0) // c2
                sema_wait(&fill); // c3
        int tmp = do_get(); // updates numfull // c4
        sema_post(&empty); // c5
        Mutex_unlock(&m); // c6 }
```

GOOD LUCK!

- TAs plan to review for exam more in discussion section – sample exams