CSE 120
Principles of Operating Systems
Spring 2023
Midterm Review
Yiying Zhang
Announcements

- Project 1 and Homework 2 due tomorrow
- Check out project 1 submission instructions and try submitting before the deadline
  - We will not take “there’s some issue when I submitted” as a reason for late submission!
- Additional office hour this Wed 2-3pm to answer last-min questions for the midterm
Midterm

• This Wed 5/3 7pm-9pm in LEDDN AUD 216

• Covers material from lec1 to lec 11 (last Fri)
• Backup slides materials not in exams
• Based upon lecture material, homeworks, and project
• One double-sided A4 cheat sheet allowed, nothing else

• Obligatory: Please, do not cheat
  ♦ No one involved will be happy, particularly the teaching staff
  ♦ Sign a zero-point agreement at the beginning of the exam!
    » We will not grade an exam without the signature
Tentative Amount and Type of Problems (subject to change)

• 15 True or False (15x2=30 pts)
  ♦ Choosing T or F, no explanation needed or allowed

• 4 multiple choices (6x3=18 pts)
  ♦ Choosing 0 to max number of options, no explanation needed or allowed

• 4 short questions (4x5=20 pts)
  ♦ Fill some slot in code piece; write outcome of a code piece
  ♦ No need for explanation, but short explanation allowed
Format of midterm (cont.)

- 1 project question (10 pts)

- 2 big problems (10+12=22 pts)
  - Fill slots in code, find bugs, etc.
  - Explanation encouraged, but not required
  - The problems will have ???111, ???222, … for you to fill
OS/Hardware/Application Interaction

• Dual-mode operation (mode switch)
  ♦ What causes a mode switch (trap)?

• Privileged instructions
  ♦ What type of instructions should be privileged?
  ♦ Who checks that?

• Interrupts
  ♦ What are the two ways to handle external events?
  ♦ Types of interrupt
  ♦ Handling interrupts (OS as a giant interrupt handler)

• System calls
  ♦ Flow of system call handling (and return)
Processes

• Process concept
  ♦ What is a process?
  ♦ What is the difference between a process and a program?
  ♦ What is contained in a process?
  ♦ What is PID?

• Process Control Blocks (PCBs)
  ♦ What information does it contain?
  ♦ How is it used in a context switch?

• Process State
  ♦ What states can a process be in?
  ♦ When does a process change state?
Process Creation/Termination

- What does fork() on Unix do?
  - What does it mean for it to “return twice”?
- What does exec() on Unix do?
  - How is it different from fork?
- How are fork and exec used to implement shell?
- Exercise: homework 1 problem 6
Threads

- What is a thread?
  - What is the difference between a thread and a process?
  - How are they related?
- Why are threads useful?
- How are threads managed by the run-time system?
  - Thread control blocks, thread queues
  - How is this different from process management?
Synchronization

- Mutual exclusion and critical section
- Locks
- Semaphore
- Conditional variables
- Concurrency bugs
Synchronization

• Why do we need synchronization?
  ♦ Coordinate access to shared data structures
  ♦ Coordinate thread/process execution

• When are resources shared?
  ♦ Global variables, static objects
  ♦ Heap objects
  ♦ Not shared: local variables

• What can happen to shared data structures if synchronization is not used?
  ♦ Bank account example, too much milk example
 Mutual Exclusion

• What is mutual exclusion?
• What is a critical section?
  ♦ What are the requirements of critical sections?
    » Mutual exclusion (safety)
    » Progress (liveness)
    » Bounded waiting (no starvation: liveness)
    » Performance

• How does lock work? How to use lock?
Implementing Locks

- **Goal:** Use mutual exclusion to protect critical sections of code that access shared resources
- **Method:** Use locks (spinlocks or disable interrupts)
- **Problem:** Critical sections (CS) can be long

**Spinlocks:**
- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater chance for lock holder to be interrupted

```plaintext
acquire(lock)
...
Critical section
...
release(lock)
```

**Disabling Interrupts:**
- Doesn’t work on multiprocessor
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)
Condition Variables

- What is a condition variable used for?
  - Coordinating the execution of threads
  - Not mutual exclusion

- Operations
  - What are the semantics of Wait?
  - What are the semantics of Signal?
  - What are the semantics of Broadcast?

- How are condition variables different from semaphores?
Semaphores

• What is a semaphore?
  ♦ What does Wait/P do?
  ♦ What does Signal/V do?
  ♦ How does a semaphore differ from a lock?
  ♦ What is the difference between a binary semaphore and a counting semaphore?

• Using semaphores to solve synchronization problems
  ♦ How many semaphores to use?
  ♦ How to set their initial values? Binary or counting semaphore?
  ♦ Where to call wait/signal?
  ♦ What is the critical section?

```c
wait(S) {
  while (S<=0) S++;
  S--;
}
signal(S) {
  S++;
}
```
Synchronization Problems

- Producer/Consumer
  - Semaphore solution
  - Conditional variable solution

- Readers/Writers
  - (solution not in exam)

- Dining Philosophers
  - (solution not in exam)
Producer & Consumer – semaphore

Producer

while (1) {

produce an item;

wait(EMPTY);

acq(lock);

insert(item to pool);

rel(lock);

signal(FILLED)
}

Init: FILLED = 0; EMPTY = N;

Consumer

While (1) {

wait(FILLED);

acq(lock);
remove(item from pool);
rel(lock)

signal(EMPTY);

consume the item;
}
**Producer & Consumer – use condition variables**

**Producer**

```
while (1) {
    produce an item;
    acquire(mutex);
    while (pool is Full) {
        wait(NotFULL);
        wait(NotFULL);
    }
    record if pool was empty;
    insert(item)
    if (pool was empty)
        signal(NotEMPTY)
    release(mutex)
}
```

**Consumer**

```
While (1) {
    acquire(mutex)
    while (pool is Empty) {
        wait(NotEMPTY)
    }
    record if pool was full
    remove(item)
    if (pool was Full)
        signal(NotFULL)
    release(mutex)
    consume the item;
}
```

*Is this busy waiting?*
Deadlock

• When does deadlock happen?
  ♦ Processes are waiting on each other and cannot make progress

• What are the conditions for deadlock?
  ♦ Mutual exclusion
  ♦ Hold and wait
  ♦ No preemption
  ♦ Circular wait

• Dealing with deadlock
  ♦ Ignore it
  ♦ Prevent it (prevent one of the four conditions)
  ♦ Avoid it (have tight control over resource allocation)
  ♦ Detect and recover from it
Practice Midterm

```c
volatile int balance = 0;

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

int main(int argc, char *argv[])
{
    pthread_t p1, p2, p3;
    pthread_create(&p1, NULL, mythread, "A");
    pthread_join(p1, NULL);
    pthread_create(&p2, NULL, mythread, "B");
    pthread_join(p2, NULL);
    pthread_create(&p3, NULL, mythread, "C");
    pthread_join(p3, NULL);
    printf("Final Balance is %d\n", balance);
}
```

p1: Balance is 200  (p2: Balance is 400, p3: Balance is 600)
Final Balance is 600

What if there are no join statements?
You are given a game with 3 players and 3 colors (red, blue, green). You must ensure that the players only move pieces in the order: red, blue, green, red, blue, green, etc. Write three routines using semaphores for the players to call: MoveRed, MoveBlue, and MoveGreen.
Semaphore red = 1;
Semaphore blue = 0;
Semaphore green = 0;

MoveRed() {
    wait(red);
    play();
    signal(blue);
}

MoveBlue() {
    wait(blue);
    play();
    signal(green);
}

MoveGreen() {
    wait(green);
    play();
    signal(red);
}
You have been hired by Large-Concurrent-Systems-R-Us, Inc. to review their code. Below is their atomic_swap procedure. It is intended to work as follows:

atomic_swap should take two queues as arguments, dequeue an item from each, and enqueue each item onto the opposite queue. If either queue is empty, the swap should fail and the queues should be left as they were before the swap was attempted. The swap must appear to occur atomically – an external thread should not be able to observe that an item has been removed from one queue but not pushed onto the other one. In addition, the implementation must be concurrent – it must allow multiple swaps between unrelated queues to happen in parallel. Finally, the system should never deadlock.

Is the implementation below correct?
extern Item *dequeue(Queue *);  // pops an item from a stack

extern void enqueue(Queue *, Item *);  // pushes an item onto a stack

void atomic_swap(Queue *q1, Queue *q2) {
    Item *item1;
    Item *item2;  // items being transferred
    P(q1->lock);
    item1 = pop(q1);
    if(item1 != NULL) {
        P(q2->lock);
        item2 = pop(q2);
        if(item2 != NULL) {
            push(q2, item1);
            push(q1, item2);
            V(q2->lock);
            V(q1->lock);
        }
    }
}
3. A common pattern in parallel scientific programs is to have a set of threads do a computation in a sequence of phases. In each phase $i$, all threads must finish phase $i$ before any thread starts computing phase $i+1$. One way to accomplish this is with barrier synchronization. At the end of each phase, each thread executes $\text{Barrier::Done}(n)$, where $n$ is the number of threads in the computation. A call to $\text{Barrier::Done}$ blocks until all of the $n$ threads have called $\text{Barrier::Done}$. Then, all threads proceed. You may assume that the process allocates a new Barrier for each iteration, and that all threads of the program will call $\text{Done}$ with the same value.

Implement Barrier using lock and condition variable (Mesa style signal).

```java
class Barrier {
    int called = 0;
    Lock lock;
    Condition barrier;

    Barrier () {
        lock = new Lock();
        barrier = new Condition(lock);
    }

    void Done (int needed) {
        ???
        ???
        if (called == needed) {
            ???
            barrier.???;
        } else {
            barrier.???;
        }
        ???
    }
}
```
Homework 2

4. Microsoft .NET provides a synchronization primitive called a CountdownEvent. Programs use CountdownEvent to notify the completion of many threads (similar to CountDownLatch in Java). A CountdownEvent is initialized with a count value that can be in two states, nonsignalled and signalled. Threads use a CountdownEvent in the nonsignalled state to decrease the count and wakes up (unblocks) all waiting threads. Once a CountdownEvent has transitioned from nonsignalled to signalled state, it remains in the signal state. In the signalled state, at any time a thread may call the methods to decrease the count and Increment to increase the count. In the signalled state, Wait, Decrement, and Increment methods:

```java
class CountdownEvent {
    int counter;
    bool signalled;
    Lock lock;
    Condition cv;

    CountdownEvent (int count) {
        counter = count;
        if (counter > 0) {
            signalled = false;
        } else {
            signalled = true;
        }
        lock = new Lock();
        cv = new Condition(lock);
    }

    void Increment () {
        if (????) {
            counter++;
        }
    }

    void Decrement () {
        if (????) {
            counter--;
            if (????) {
                ???
                cv.???
            }
        }
    }

    void Wait () {
        ???
        if (????) {
            ???
            cv.???
        }
    }
}
```

Notes:

- The `CountdownEvent` constructor takes an integer `count` as input and initializes the CountdownEvent with this count value. Values of `count` cause the CountdownEvent to be constructed in the nonsignalled state. Other values cause the CountdownEvent to be constructed in the signalled state.
- `Increment` increments the internal counter.
- `Decrement` decrements the internal counter. If the counter reaches zero, the CountdownEvent transitions to the nonsignalled state and unblocks any waiting threads.
- `Wait` blocks the calling thread if the CountdownEvent is in the nonsignalled state, and otherwise returns immediately.
- Each of these methods is relatively short.