CSE 120
Principles of Operating Systems
Fall 2021
Midterm Review
Yiying Zhang
Announcements

• Homework 1 solution posted online

• Project 1 due this Friday

• Check out project 1 submission instructions (Piazza post @236) and try submitting before the deadline
  ♦ We will not take “there’s some issue when I submitted” as a reason for late submission!

• Moving next week’s office hour to next Monday 3-5pm to answer last-min questions for the midterm

• This week’s discussion session: HW1 solution and rest of HW2
Midterm

• Next Tue 10/26 2pm-3:20pm on Canvas

• Covers material from lec1 to lec 7 (last Thur)
• Backup slides materials not in exams
• Based upon lecture material, homeworks, and project
• Open book, open everything

• 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 (on Canvas)!
    » We will not grade an exam without the signature on Canvas
Online Exam with Canvas

• Midterm will be available at “Quizzes” on Canvas at 2pm on 10/26, and will be closed at 3:20pm
  ♦ Be sure to turn in before 3:20pm!
    » It’s an automatic system and we have no way to accept late turn in.
    » Try Canvas Quiz out using the practice exam

• Canvas option set up of the real midterm is similar to practice midterm, with some differences
  ♦ Will have more problems
  ♦ Show one question at a time, but can jump to any question
  ♦ One attempt (one final submit click) only
Online Exam with Canvas (cont.)

• Join Zoom meeting link for lectures
  ♦ Mainly used for answering clarification questions during the exam
    » TAs and I will be online handling questions
    » Type your questions in chat instead of using your mic
  ♦ Also used for preventing cheating
    ♦ Join the meeting at least 5 min prior to exam start time (i.e., join on or before 4:55pm)
    ♦ Turn on video for at least one minute before taking the exam
    ♦ Turn on video again when you submit your exam
Tentative Amount of Problems (subject to change)

- 15 True or False (15x3=45 pts)
  - Choosing T or F, no explanation needed or allowed
  - Full credit or zero credit

- 4 multiple choices (4x4=16 pts)
  - Choosing 1 to max number of options, no explanation needed or allowed
  - Full credit or zero credit

- 2 code (short) questions (2x5=10 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 (5 pts)

• 2 big problems (2x12=24 pts)
  ♦ Fill slots in code, answer questions, etc.
  ♦ Explanation encouraged, but not required
  ♦ The problems will have [blank1], [blank2], … for you to fill
  ♦ Answer in the following format:

  [Blank 1 answer:] your answer

  [Blank 2 answer:] your answer

  ….
OS/Hardware/Application Interaction [lec2]

• 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 [lec3]

• 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
[lec3]

- 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?
Threads [lec4]

• 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?

• User-level and kernel-level threads
  ♦ What’s the difference?
  ♦ What are the advantages/disadvantages of one over another?
  ♦ Different user-level kernel-level thread mapping models
Synchronization [lec5-7]

- Mutual exclusion and critical section
- Locks
- Semaphore
- Conditional variables
- Concurrency bugs
Synchronization [lec5]

• 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 [lec5]

• 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?
acquire(lock);
if (no Milk)
    buy milk;
release(lock);

Critical Section

• What is the problem with this solution?
How can we separate “checking” from “buying milk” and only lock “checking”?

```c
local_flag = FALSE;

Acquire(lock);
if (no note && noMilk) {
    leave note;
    local_flag = true;
}
Release(lock);

If (local_flag) buy milk;
If (local_flag) {
    local_flag = FALSE;
    remove note;
}
```
Implementing Locks [lec5]

- 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

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)
Implementing Locks with a queue [lec5]

- If cannot hold lock, give up CPU (move to block queue)
- Use a guard on the lock itself

```c
void acquire (lock) {
    disable interrupts;
    while (test-and-set(lock→guard)) ;
    if (lock→held == 0) {
        lock→held = 1;
        lock→guard = 0;
        enable interrupts;
        return;
    }
    put current thread on lock→Q;
    lock→guard = 0;
    go to sleep;
    enable interrupts;
}
```

```c
struct lock {
    int held = 0;
    int guard = 0;
    queue Q;
}
```

```c
void release (lock) {
    disable interrupts;
    while (test-and-set(lock→guard)) ;
    if (lock→Q is empty)
        lock→held = 0;
    if (lock→Q is not empty)
        move a waiting thread to the ready queue;
    lock→guard = 0;
    enable interrupts;
}
```
Semaphores [lec6]

- 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?

```java
wait(S) {
    while (S<=0) ;
    S--;
}
signal(S) {
    S++;
    ;
}
```
Condition Variables [lec7]

• 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?
Condition Variables [lec7]

- Wait (condition)
  - Block on “condition”

- Signal (condition)
  - Wakeup one or more threads blocked on “condition”

- Conditions are like semaphores but:
  - signal is no-op if none blocked
  - There is no counting!

Queue of waiting Process trying to Enter CSes protected by lock L

Lock: L
Condition variables: x(L) y(L)

Queues associated with x, y condition

Shared data

Operations
Synchronization Problems

• Producer/Consumer
  ♦ Semaphore solution
  ♦ Conditional variable solution

• Readers/Writers
  ♦ Reader preference solution
  ♦ Writer preference solution
  ♦ Fair solution

• Dining Philosophers
  ♦ Deadlock
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

while (1) {
    produce an item;
    acquire(mutex);
    while (pool is Full) {
        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 [lec7]

- 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
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);
        }
    }
}

- Incorrect
- It can deadlock
- It fails to restore q1 or q2
- It has unmatched P's and V's