Midterm

• This Thur 4/30 5pm-6:20pm on Canvas

• Covers material from lec1 to lec 8 (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 5pm on 4/30, and will be closed at 6:20pm
  ♦ Be sure to turn in before 6:20pm!
    » It’s an automatic system and we have no way to accept late turn in.
  ♦ A and B version of midterm assigned randomly (visible at 5pm)

• Canvas option set up similar to practice midterm
  ♦ Show one question at a time, but can jump to any question
  ♦ One attempt (one final submit click) only
  ♦ Questions other than TorF and Multiple Choices will be manually graded afterwards (ignore Canvas autograding)
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
Format of midterm

• 17 True or False (17x2=34 pts)
  ♦ Choosing T or F, no explanation needed or allowed
  ♦ Full credit or zero credit

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

• 3 code questions (3x4=12 pts)
  ♦ Fill some slot in code piece; write outcome of a code piece
  ♦ No need for explanation, but short explanation allowed
• 3 big problems (3x12=36 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

  …..
Announcements

• Homework 2 solution posted online, grading to be finished soon

• Moving this Friday’s office hour to this Wed: Wed 9-11am
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 shells?
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
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);

• What is the problem with this solution?
Deep thinking [lec5]

- 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;
Acquire(lock)
If (local_flag){
    local_flag = FALSE;
    remove note;}
Release (lock);
```
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)

```plaintext
acquire(lock)
...
Critical section
...
release(lock)
```
Implementing Locks [lec5]

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

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

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;
    enable interrupts;
    go to sleep;
}

void release (lock) {
    disable interrupts;
    while (test-and-set(lock→guard)) ;
    if (lock→Q is not empty)
        move a waiting thread to the ready queue;
    else
        lock→held = 0;
        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?

```c
wait(S) {
    while (S<=0)
    S--;
}

signal(S) {
    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

- Wait (condition)
  - Block on “condition”

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

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

• Producer/Consumer
  ♦ Semaphore solution
  ♦ Conditional variable solution

• Readers/Writers
  ♦ Reader preference solution
  ♦ Writer preference solution
  ♦ Fair solution
Locks and Condition Vars

• In Nachos, we don’t have monitors
• But we want to be able to use condition variables
• So we isolate condition variables and make them independent (not associated with a monitor)
• Instead, we have to associate them with a lock
• Now, to use a condition variable…
  ♦ Threads must first acquire the lock
  ♦ Wait/sleep releases the lock before blocking, acquires it after waking up
Producer & Consumer – semaphore working

Producer
while (1) {
    produce an item;
    wait(EMPTY);
    acq(lock);
    insert(item to pool);
    rel(lock);
    signal(FULL)
}

Consumer
While (1) {
    wait(FULL);
    acq(lock);
    remove(item from pool);
    rel(lock)
    sginal(EMPTY);
    consume the item;
}

Init: FULL = 0; EMPTY = N; Mutex = 1;
Producer & Consumer – use condition variables

### Producer

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

### Consumer

```c
While (1) {
    acquire(mutex)
    while (pool is Empty) {
        wait(NotEMPTY)
    }
    record if pool was full
    remove(item)
} 
```

---

`Is this busy waiting?`
Concurrency Bugs [lec7]

- Deadlock, starvation, livelock
- Conditions of deadlock
- Strategies to prevent/break deadlocks
- Non-deadlock concurrency bugs
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
Other Concurrency Bugs
[lec7]

• Starvation
  ♦ Indefinite denial of a resource (CPU, lock)
  ♦ Causes
    » Side effect of scheduling
    » Side effect of synchronization

• Other non-blocking bugs
  ♦ Data race
  ♦ Atomicity violation
  ♦ Order violation
CPU Scheduling [lec8]

- What are the mechanisms?
  ♦ Preemptive and non-preemptive scheduling
  ♦ Context switch
  ♦ When does scheduling happen?
CPU Scheduling [lec8]

• Goals
  ♦ Minimize turnaround time
  ♦ Maximize throughput
  ♦ Short response time
  ♦ Fairness

• Scheduling algorithms
  ♦ FIFO
  ♦ RR
  ♦ SJCF and SRTCF
  ♦ Priority scheduling and MLFQ
What is the range of possible values for x at the end of this code??

- Any integer from -100 to 100
What’s the output of the following program?

```c
void main() {
    int local = 1;
    int pid = fork();
    if (pid == 0) {
        local = 3;
        execlp("/bin/ls", "ls", NULL);
        printf("The if-path local variable is %d\n", local);
    } else {
        local = 4;
        printf("The else-path local variable is %d\n", local);
        exit(0);
    }
}
```

"the output of ls" and "The else-path local variable is 4"
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
## Practice Midterm

Assume the following processes, each with their arrival time and run time.

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time (sec)</th>
<th>Run Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0s</td>
<td>5s</td>
</tr>
<tr>
<td>P2</td>
<td>1s</td>
<td>3s</td>
</tr>
<tr>
<td>P3</td>
<td>5s</td>
<td>1s</td>
</tr>
<tr>
<td>P4</td>
<td>6s</td>
<td>5s</td>
</tr>
</tbody>
</table>

With SJF Non-Preemptive job scheduling, what is the average **turnaround** time?

Job sequence: P1 (5s), P3 (1s), P2 (8s), P4 (8s)

\[(5+1+8+8)/4 = 5.5\text{ sec}\]
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