CS 140 Midterm Review Session

Administrivia

- Project 2
- Midterm is **in class** on Monday
  - Open notes, but…
  - No textbook!
  - No electronic devices!
  - Can **print** any material you want
- 50% of class grade is from exams:

max(midterm > 0 ? final : 0, (midterm + final)/2)

Material Covered

- Threads and Processes
- Concurrency
- Scheduling
- Virtual Memory Hardware
- Virtual Memory OS Techniques
- Synchronization
- Memory Allocation

Threads and Processes

- A **process** is an instance of a program running
- Process Control Block (**PCB**) contains: page directory (defines the virtual address space), saved registers, file descriptors, process ID, priority, pointer to PCB of next process to run, accounting information (e.g recent CPU time)
  - Tracks state of process
  - Includes information necessary to run
- During a **context switch**, save state into PCB (registers, address spaces) and reload state from the next PCB

Threads and Processes

- A **thread** is a schedulable execution context
  - Most popular abstraction for concurrency
  - Lighter weight abstraction than processes
  - Allows one process to use multiple CPUs/cores
  - Allows programs to overlap I/O and computation
- **Kernel threads** can take advantage of multiprocessor, but are heavy-weight
- **User threads** are more light-weight and flexible, but can't take advantage of multiple CPUs

Concurrency

- Prevent **data races** by defining **critical sections** (require mutual exclusion, progress, and bounded waiting)
- Multiple synchronization primitives
  - **Locks**: useful for mutual exclusion; put lock around code if you want to make it atomic
  - **Condition variables**: generally used to avoid busy waiting; use *wait* and *signal*
  - **Semaphores**: typically used with a shared resource that changes availability based on an integer number of things

Synchronization

- Locks create serial code
  - Serial code gets no speedup from multiprocessors
- **Deadlock:** two or more competing actions are waiting on each other to finish
  - mutual exclusion
  - no preemption
  - multiple independent requests
  - circularity in the request graph

Practice Problem #1

```
a () {
        acquire(resource1);
        acquire(resource2);
        doWorkA();
        release(resource2);
        release(resource1);
}
b (){
        acquire(resource2);
        acquire(resource3);
        doWorkB();
        release(resource3);
        release(resource2);
}
c () {
        acquire(resource3);
        acquire(resource1);
        doWorkC();
        release(resource1);
        release(resource3);
}
```

Can this deadlock?

Practice Problem #1 - Solution

```
a () {
        acquire(resource1);
        acquire(resource2);
        doWorkA();
        release(resource2);
        release(resource1);
}
b (){
        acquire(resource2);
        acquire(resource3);
        doWorkB();
        release(resource3);
        release(resource2);
}
c () {
        acquire(resource3);
        acquire(resource1);
        doWorkC();
        release(resource1);
        release(resource3);
}
```

Can this deadlock? **YES**

Mutual exclusion: only one thread can access a resource at a time.

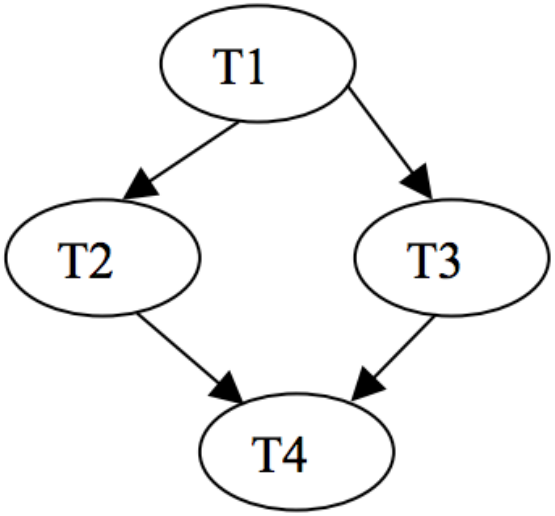
No preemption: resources cannot be forcibly taken back.

Multiple independent requests: each resource is independently requested.

Circularity: A can wait on B, which can wait on C, which can wait on A

Practice Problem #2

Assume you are given a graph that represents the relationship between four threads (T1, T2, T3, T4). An arrow from one thread (Tx) to another (Ty) means that thread Tx must finish its computation before Ty starts. Assume that the threads can arrive in any order. Use semaphores to enforce this relationship specified by the graph. Be sure to show the initial values and the locations of the semaphore operations.



Practice Problem #2 - Possible Solution

Init(sema1, 0); // Initialize sema1 to 0

Init(sema2, 0); // Initialize sema2 to 0

T1	T2	T3	T4
Computation	Down(sema1)	Down(sema1)	Down(sema2)
Up(sema1)	Computation	Computation	Down(sema2)
Up(sema1)	Up(sema2)	Up(sema2)	Computation

Scheduling

- CPU scheduling: decide which processes to run
- Many different scheduling algorithms attempting to optimize for throughput, turnaround time, response time, CPU utilization, waiting times
  - FIFO
  - Shortest Job First
  - Round-robin
  - Priority scheduling
  - Multilevel feedback queue

Practice Problem #3

- True or False: as the ratio of time slice to job length decreases, round-robin scheduling becomes equivalent to first-come-first-serve scheduling

Practice Problem #3 - Solution

- True or False: as the ratio of time slice to job length decreases, round-robin scheduling becomes equivalent to first-come-first-serve
- **False**
  - As time slice length goes to infinity (ratio increases), each job completes in a single time slice (the first time it runs), equivalent to FCFS.
  - As time slice length goes to 0 (ratio decreases), round-robin behaves increasingly different from FCFS.

Virtual Memory Hardware

- OS gives each program its own *virtual* address space
- **Segmentation**: divide memory into segments; keep track of base and bound registers
  - Causes **external fragmentation:** enough total memory, but not contiguous so cannot be used
- **Paging**: map virtual pages to physical pages
  - Causes **internal fragmentation:** memory block assigned to process is larger than needed

Virtual Memory OS Techniques

- Have more virtual memory than physical memory: save unused virtual pages to disk
- Several **page replacement** algorithms decide what memory to write to write to disk
  - FIFO
  - Clock algorithm
  - LRU
- **Thrashing:** constant paging because **working set** cannot be in memory at once
  - Working set model: process can be in memory iff all pages used by process can be in memory

Practice Problem #4a

- True or False: Increasing the page size will likely decrease the working set size (in bytes).

Practice Problem #4a - Solution

- True or False: Increasing the page size will likely decrease the working set size (in bytes).
  - **False**

Practice Problem #4b

- True or False: Increasing the page size will likely decrease the working set size (in bytes).
  - **False**
- True or False: Increasing the page size will likely decrease the chance that page revocation requires a disk write.

Practice Problem #4b - Solution

- True or False: Increasing the page size will likely decrease the working set size (in bytes).
  - **False**
- True or False: Increasing the page size will likely decrease the chance that page revocation requires a disk write.
  - **False:** As page size increases, the likelihood that page is **dirty** increases. Dirty pages are evicted to disk rather than cache.

Memory Allocation

- Dynamic allocation's main problem is **fragmentation**: memory is allocated in non-continuous blocks, resulting in a lot of unusable unallocated memory
- Various memory allocation techniques
  - First fit
  - Best fit: smallest block it'll fit in
  - Segregated free lists
- Garbage collection techniques
  - Stop-and-copy: split memory in half
  - Reference counting

Practice Problem #5

- Explain a situation where reference counting GC may leak memory that a stop-and-copy GC would probably collect?

Practice Problem #5 - Solution

- Explain why a reference counting GC may leak memory that a stop-and-copy GC would probably collect?
  - A data structure may be unreachable from any global pointers yet have a cycle. For example, in the case of A -> B -> C -> A, A, B, and C will have a ref count of 1 and not be collected under reference counting, while stop-and-copy would not reach these objects and thus wouldn't copy them to the new heap.

Practice Problem 6

- You operate a restaurant that makes very simple hamburgers: just two(identical) buns and a patty. Your job is to make sure that all hamburgers have exactly two buns and one patty.
- When a new bun has been baked, it will call the bunArrived() function. This function should only return once the bun has been used to make a hamburger.
- When a new patty has been cooked, it calls the pattyArrived() function. This function should only return once the patty has been used to make a hamburger.
- Note: while you are processing bunArrived(), a call to pattyArrived() may interrupt you, so be careful about modifying shared state.
- Define global variables, and implement the init(), bunArrived() and pattyArrived() functions using locks and condition variables as defined in Pintos (init() will run to completion before any buns or patties arrive):

Practice Problem 6

```
************************************************************
struct lock
void lock_init (struct lock *lock)
void lock_acquire (struct lock *lock)
bool lock_try_acquire (struct lock *lock)
void lock_release (struct lock *lock)

struct condition
void cond_init (struct condition * cond)
void cond_wait (struct condition *cond, struct lock *lock)
void cond_signal (struct condition *cond, struct lock *lock)
************************************************************

// Global Variables

init () {

}

bunArrived (){

}

pattyArrived () {

}
```

Practice Problem 6

```
// Global Variables
struct lock global_lock;
struct cond bun;
struct cond patty;
int waiting_buns;
int waiting_patties;
int buns_cleared_to_leave;
int patties_cleared_to_leave;

init(){
    lock_init(global_lock);
    cond_init(wait_bun);
    cond_init(wait_patty);
    waiting_buns = 0;
    waiting_patties = 0;
    buns_cleared_to_leave = 0;
    patties_cleared_to_leave = 0;
}
```

Practice Problem 6

```
bunArrived(){
    lock_acquire(global_lock);
    waiting_buns++;

    // if any buns are "finished" we're good to leave
    while (buns_cleared_to_leave == 0)
    {
        if(waiting_buns >= 2 && waiting_patties >= 1)
        {
            waiting_buns -= 2;
            waiting_patties -= 1;
            buns_cleared_to_leave += 2;
            patties_cleared_to_leave += 1;
            cond_signal(bun, global_lock);
            cond_signal(patty, global_lock);
        }
        else
        {
            // if not enough ready, wait
            cond_wait(bun, global_lock);
        }
    }
    buns_cleared_to_leave--;
    lock_release(global_lock);
}
```

Practice Problem 6

```
pattyArrived(){
    lock_acquire(global_lock);
    waiting_patties++;

    // if any patties are "finished" we're good to leave
    while (patties_cleared_to_leave == 0)
    {
        if(waiting_buns >= 2 && waiting_patties >= 1)
        {
            waiting_buns -= 2;
            waiting_patties -= 1;
            buns_cleared_to_leave += 2;
            patties_cleared_to_leave += 1;
            cond_signal(bun, global_lock);
            cond_signal(bun, global_lock);
        }
        else
        {
            // if not enough ready, wait
            cond_wait(patty, global_lock);
        }
    }
    patties_cleared_to_leave--;
    lock_release(global_lock);
}
```

General Tips

- Practice with the old exams
  - Time yourself accordingly
  - Don't just look at questions and solutions for every archived exam
- Ask questions on Google Groups
- Rewatch lectures on SCPD
- Don't rely on notes; know the material

Good luck!