CS 140

Midterm Review

Administrivia

- Project 2: User Programs was due today at noon
- Midterm “Quiz” Monday Feb. 10, 4:15-5:30pm in Gates B01.
  - Open book
  - Open notes
  - No electronic devices

Midterm Coverage

- Processes/Threads
- Concurrency
- Scheduling
- Virtual Memory
- Synchronization
- Memory Allocation

Practice Problem 1

- Multi-programming (or multi-tasking) enables more than a single process to apparently execute simultaneously.
- How is this achieved on a uniprocessor?

Practice Problem 1

- Possible Answer:
  - Multiprogramming is achieved on a uniprocessor by the concept of threading.
  - Every process' total running time is divided up into threads, which are a subset of the process' instructions that can be completed in a certain amount of time, called a timeslice.
  - When a thread's timeslice is finished, CPU time has to switch to a different thread. On a large scale, these timeslices are nanoseconds long, so it appears to the user that the processor is processing processes concurrently.
  - The ultimate goal is to keep the system responsive while really maximizing the processor's ability to process.

Practice Problem 2

- Provide two examples of an application that benefits from multithreading, and two examples where multithreading does not provide better performance than a single-threaded solution. Explain why.

Practice Problem 2

- Possible Answers:
- Yay Multithreading:
  - Application with User interface. UI thread separate from background processing thread gives better responsiveness.
  - Image processing. Any program that operates on many (mostly) independent pieces of data.
  - Any program with heavy CPU usage along with I/O. Allows the program to make progress while waiting for input/output.
- Boo Multithreading:
  - Programs with heavy I/O, hardly any CPU usage. Bottleneck will always be at I/O, threading will only add more overhead.
  - Shell programs. Have to track and keep update current state. Synchronization overhead.
  - Mainly sequential programs that modify small chunks of data. Synchronization overhead will outweigh threading benefits.

Practice Problem 3

Consider the following implementation of the producer/consumer code from lecture 5 (Note that the control statement surrounding the cond_wait call in get() is an if statement instead of a while statement).

```
char buffer[SIZE];
int count = 0, head = 0, tail = 0;
struct lock l;
struct condition dataAvailable;
struct condition spaceAvailable;
lock_init(&l);
cond_init(&dataAvailable);cond_init(&spaceAvailable);
```

```
void put(char c) {
    lock_acquire(&l);
    while (count == SIZE) {
        cond_wait(&spaceAvailable, &l);
    }
    count++;
    buffer[head] = c;
    head++;
    if (head == SIZE) {
        head = 0;
    }
    cond_signal(&dataAvailable, &l);
    lock_release(&l);
}
```

```
char get() {
    char c;
    lock_acquire(&l);
    if (count == 0) {
        cond_wait(&dataAvailable, &l);
    }
    count--;
    c = buffer[tail];
    tail++;
    if (tail == SIZE) {
        tail = 0;
    }
    cond_signal(&spaceAvailable, &l);
    lock_release(&l);
    return c;
}
```

Assume cond_signal always wakes up at most one thread and that there are no spurious wakeups. Also assume that multiple threads can be executing the put() function at any given time.

Is the above code correct? Justify your answer.

Practice Problem 3

- Incorrect.
- Consider “getter” that is waiting to be signaled.
- “Putter” signals data_ready, “getter” is unblocked.
- Before “getter” wakes up, another thread runs get();

Practice Problem 4

- 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 4

```
************************************************************
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 4

```
// 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 4

```
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 4

```
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);
}
```

Practice Problem 5

- The 4 conditions for deadlock are:
  - 1. Limited Access (mutual exclusion)
  - 2. No preemption
  - 3. Multiple Independent requests
  - 4. Circularity in request graph
- Consider the following three threads, where resource1, resource2, resource3 are generic resources that only one thread can hold at a time.

Practice Problem 5

```
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);
}
```

Practice Problem 5

- This system has the potential for deadlock.
- For each of the 4 conditions,describe:
  - 1. Why the condition is satisfied
  - 2. What changes to the definition of the system would cause the condition not to be satisfied

Practice Problem 5

- Limited Access:
  - Resources are mutually exclusive. Only one thread can access at a time.
  - If we defined the resources to be shareable, we wouldn't have deadlock.
- No Preemption:
  - Resources cannot be forcibly taken back.
  - If we defined resources to preemptable, we wouldn't have deadlock.
- Multiple Independent Requests:
  - 3 separate resources are requested individually.
  - If all resources were packaged inside one "super-resource" we wouldn't have deadlock.
- Circularity in Request Graph:
  - There is a situation where A can be waiting on B, who is waiting on C, who is waiting on A…
  - If we reordered the requests, we could avoid deadlock.

Practice Problem 6

- You are tasked with writing a multi-threaded basic word-processor (e.g. notepad). You are trying to decide whether to use multiple kernel threads or multiple user threads inside a single kernel thread.
- Give 2 advantages of user threads over kernel threads , and 2 advantages of kernel threads over user threads.

Practice Problem 6

- Possible answers:
- User:
  - Portable. Not as dependent on the target OS.
  - Usually faster. Don't have to take hit of full context switch or fork/join.
  - Usually more memory efficient. Don't need kernel to allocate tracking data.
- Kernel:
  - Can make us of multi-processor machines more easily.
  - Can make blocking calls without stopping entire program (although thread packages often abstract this).
  - Allow kernel to make scheduling decisions (ease of programming and access to full kernel info).