NAME: ____ NETID: ______________

CS4410 Prelim 2

This exam has four questions for a total of 100 points
In class, closed book, 75 minutes….

Problem 1: Java synchronization. 20 points

An application containing 25 threads in a single process uses *windows display handles* to talk to the display. The application has a set of 3 handles available and they must be shared among the threads. Assume that a windows display handle is of type WDH. Code a Java class called DispHandles with the interface shown below, using the Java synchronization mechanism that seems most appropriate to you:

- DispHandles dhs = new DispHandles(WDH h0, WDH h1, WDH h2); *Creates a new instance of the DispHandles class and initializes it with three windows display handles. Called once during startup of the application.*
- WDH dhs.Grab(); *Obtains exclusive access to one of the three handles and returns it. May block if the three are currently in use.*
- void dhs.Release(); *Releases a previously grabbed handle back into the DispHandles object.*

A thread may call int Thread.currentThread().getId() to get a unique positive ID.

```
Class DispHandles {
        WDH[] Handles = new WDH[3];
        int[] InUse = new int[3];
        Semaphore mutex = 1, count = 3;

        // Constructor notes the Handles pointers and
        // marks each as available (InUse[k] = -1)
        Public DispHandles(WDH h0, WDH h1, WDH h2) {
                Handles[0] = h0; Handles[1] = h1; Handles[2] = h2;
                InUse[0] = InUse[1] = InUse[2] = -1;
        }

        // Grab uses a counting semaphore to make sure one is available
        // If so, get Mutex and find a free handle, mark it as in use "by me"
        Public WDH Grab() {
                count.acquire();
                mutex.acquire();
                for(int k = 0; k < 3; k++)
                        if(InUse[k] == -1)
                                break;
                InUse[k] = Thread.currentThread().getId();
```

```
            mutex.release();
            Return Handles[k];
        }

        // Release figures out which handle the caller is using and frees it
        // then releases the semaphore.  Needs Mutex to protect against
        // concurrent calls to Grab or Release
        Public void Release()
        {
            mutex.acquire();
            for(int k = 0; k < 3; k++)
                if(InUse[k] == Thread.currentThread().getId())
                    break;
            InUse[k] = -1;
            mutex.release();
            count.release();
        }
}
```

Problem 2: Programming with Threads. 10 points.

You have been hired by Amazon.com to improve the performance of a web server application that runs on a **single-core** machine rated at 2GHz. You measure its performance; the server needs 100 seconds to construct and transmit 1000 web pages in response to requests from 10 client browsers. Having received an A in Cornell's CS4410, you are a world expert on multi-threading code, and the application appeared to have a lot of potential to benefit from threading. Accordingly, you modify the application into a multi-threaded version and run it with four threads on the same machine. Unfortunately, now it takes 150 seconds to do the exact same workload! Assume that the multithreaded code is correct and thread safe, and that nothing else is running on the machine. What could cause this multi-threaded, single-core, slowdown? We're looking for a single "main" reason described in a *short* answer.

The most likely explanation is that the cost of context switching is high. Thread context switching is usually quite cheap, but if the threads use an inefficient synchronization mechanism that involves a lot of contention (e.g. for locks), they can spend a ton of time context switching from thread to thread without much work actually being done.

We will also give full credit for answers that argue that the threads are causing contention within the processor (L2) memory cache or that context switching is preventing the processor from achieving the normal level of pipelining (for example by causing branch prediction to fail). However, we're skeptical that this would be the most likely cause of the slowdown.

Problem 3: Paging. 40 points

For a-b: Suppose that you are working on an operating system that implemented a *paged page table*, meaning that the memory used to store a page table is itself virtual and might be paged in and out. Assume that the size of a page table entry is 32 bytes.

a) [5 points]. Assume that the O/S has cleared the TLB and L2 cache (both are empty). How many bytes must the CPU fetch in order to access a single byte from virtual memory?

Because the page table is paged, we'll access 1 byte from memory, 32 bytes for the PTE describing the page our byte was in, and 32 bytes for the PTE describing the page of the page table that the first PTE was in (these accesses occur in the opposite order, of course). So: 1+32+32 = 65 bytes. Pretty high overhead to access one measly little byte in memory!

b) [5 points]. Same setup, but now assume that the TLB contains all PTEs that are referenced. The L2 cache is still empty.

Now the cost will be just 1 byte. The two PTE accesses will be free because the PTEs are both in the TLB.

For c-e, assume that you are working with an operating system that clears the TLB and L2 cache for every heavyweight process context switch. The CPU has a maximum rated speed of 2GHz. Suppose that you measure performance immediately after a context switch.

c) [5 points]. Would you expect the CPU to achieve its full potential 2GHz performance? Explain.

As we saw in answer 3-a, until the TLB warms up we pay a huge overhead. For the CPU to run at full speed, the L2 cache also needs to be pretty warm. On the other hand, not that many distinct PTE entries are touched per second and it doesn't really take long for the L2 cache to warm up: perhaps a million cycles or so. Moreover, heavyweight process switch events don't occur very often. So now and then we take a "hit" for a few million processor cycles, but that hit isn't a huge percentage of 2GHz. In summary: there will certainly be a cost, maybe even a measurable cost, but it won't be huge.

d) [5 points]. What must be true for the CPU to run at its full potential speed? Answer in a short clear way that refers to the TLB, cache and pipelining. By pipelining we mean speculation, superscalar execution, instruction and data reordering, branch prediction, and similar mechanisms.

The CPU runs at full speed if the TLB and cache are warm (contain the "working set" of PTEs and main-memory values, respectively) and if the pipeline mechanisms are working well (speculation and branch predictor are achieving an average quality of prediction).

e) [5 points]. What "costs" are incurred when a page-fault occurs? By costs, we mean "anything that takes time, and would not have occurred if the page-fault had not taken place."

When a page fault occurs, we need to perform an interrupt from user-mode to kernel mode, save some registers, run the paging algorithm to identify a page into which we can load the desired page (perhaps kicking a page out to make room), then do a disk I/O to read in the needed page, and then reload the registers and return from the interrupt. In all of this the disk I/O is (by far) the dominating cost.

f) [5 points]. True or false (explain briefly): The Working Set Clock Algorithm (Denning's Algorithm as actually implemented in systems like Linux) requires perfect predictions of future paging activity, for a time-period Δ time-units into the future, and will "thrash" if predictions are poor.

Completely false. The clocked version of the working set algorithm runs on the actual page reference sequence, which it detects in a simple way by periodically marking all pages as idle, and then remarking them as "in use" one by one as pages are touched. No prediction of future behavior is needed.

g) [5 points]. True or false (explain briefly): The Working Set Algorithm (WS) and the Optimal Working Set policy (WSOPT) achieve the same page hit rate.

True. We proved this in class. Basically, at each time step t, the WS algorithm pages out the page referenced at time t-Δ if that page wasn't touched in the period t-Δ+1 through t. WSOPT made exactly the same decision, except that it paged the same page out Δ time units earlier.

h) [5 points]. True or false (explain briefly): The Working Set Algorithm (WS) and the Optimal Working Set policy (WSOPT) maintain the identical set of resident pages at all times after t=Δ.

False: WS will keep pages around for Δ time units longer than WSOPT, so in general it has extra pages beyond the actual working set.

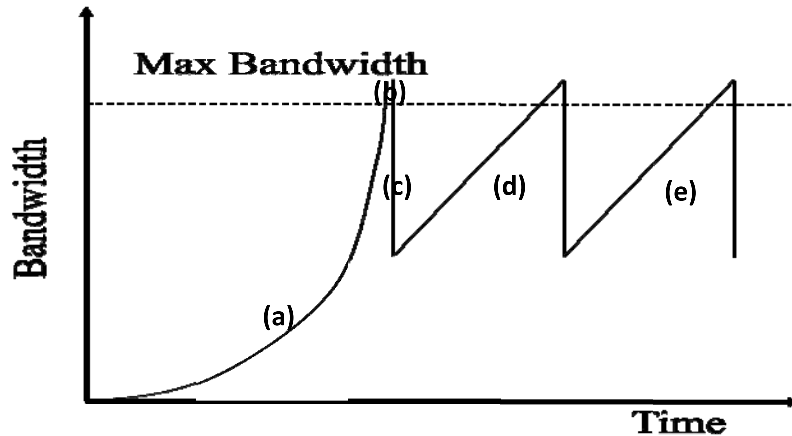
Problem 4: Internet. 30 points.

a) [10 points] Is it possible for two computers on the Internet to simultaneously have the same IP address? Explain why not, or (if it is possible), give examples of situations where this can happen.

Definitely. For example, two people might both own LinkSys wireless routers. The routers will have different external IP addresses, but within the two wireless regions, each router might use the same IP addresses. Thus there could be two machines that both have IP address 192.168.1.10 – perhaps one in your friend's house and one in your house.

b) [10 points] You measure the performance of a TCP connection between two computers in your network and end up with the following graph showing the bandwidth the sender is sending at (bytes/second), the speed of the network itself ("max bandwidth") and, on the x-axis, elapsed time. Explain why the graph has this strange shape. Notice that we've labeled parts of the graph. We're hoping that you'll use these labels in your brief written explanation to help us connect your answer to the picture in a standard way. Of course you may not need to mention every single label: to be on the safe side, we included plenty of them, in the hope that no matter what your answer is, you could still use these labels to organize it in a clear way.

(a) is showing "slow start": TCP starts up by repeatedly doubling the speed of transmission until it experiences a packet loss (b), which occurs when it exceeds the actual maximum speed of the network. At that point it uses a "multiplicative" slowdown (here it seems to halve the speed: event (c)). Now we enter a mode in which TCP ramps up ("additive" speedup"), then goes over the maximum and experiences a loss, then does a multiplicative slowdown, again and again (shown as (d), (e), etc).



c) [10 points] Suppose that you write a program that makes a TCP connection to a remote location over the Internet and then sends credit card information over the TCP connection. Is it possible that an intruder somewhere in the Internet might manage to obtain a copy of the list, without breaking into your two computers (the sender and the receiver)? Explain very briefly.

The Internet isn't very secure at all, and TCP doesn't normally encrypt or hide data – it just breaks it into chunks that fit into IP packets. Thus, an intruder who sits on a router between your two sites could just peek into the packets and see their contents. There are other ways the intruder might spy on the communication too. For example, if the communication passes over a wireless network, every message is basically a radio broadcast and anyone with an antenna can see the packets.