CS4410 Final Exam : Solution set

1. Bakery Algorithm after optimization:

```
1     int ticket[N] = 0;
2     boolean choosing[N] = false;
3
4     void CSEnter(int i)
5     {
6         choosing[i] = true;                          // Removed by optimizer
7         ticket[i] = max(ticket[0], … ticket[N-1])+1;
8         choosing[i] = false;                         // Removed by optimizer
9         for(k = 0; k < N; k++) {
10                while(choosing[k])                    // Removed by optimizer
11                    continue;                         // Removed by optimizer
12                while(ticket[k] && (ticket[k],k) < (ticket[i],i))
13                    continue;
14        }
15    }
16
17    void CSExit(int i)
18    {
19        ticket[i] = 0;
20    }
```

b) [10 pts] Suppose that you were to compile and execute the optimized code. Would it still satisfy the conditions for correctness (safety, bounded delay and fairness)? Explain.

No. With these lines removed by the optimizer, a race condition can arise in which a thread X is trying to enter the critical section and executes the "who goes first" test (lines 12-13) while thread Y is still selecting its ticket (line 7). In this situation if X can't tell that Y is picking, X might not realize that Y has the smaller ticket value. Both would enter the critical section: X because it would be unaware that Y was even trying (it would see that ticket[Y] is zero) and Y because it received the smaller ticket (and hence when Y executes line 7, it "wins" the competition to enter first. This violates safety.

Bounded delay and fairness are not impacted by this change.

2. Below is a correct implementation of the Bounded Buffer algorithm we studied in class:

```
Shared: Semaphores mutex, empty, full;
      mutex = 1;  /* for mutual exclusion*/
      empty = N; /* number empty buf entries */
      full = 0;     /* number full buf entries */
```

Producer

```
do {
            . . .
            // produce an item in nextp
            . . .
            empty.acquire();
            mutex.acquire();
            . . .
            // add nextp to buffer
            . . .
            mutex.release();
            full.release();
} while (true);
```

Consumer

```
do {
            full.acquire();
            mutex.acquire();
            . . .
            // remove item to nextc
            . . .
            mutex.release();
            empty.release();
             . . .
             // consume item in nextc
             . . .
} while (true);
```

a) [10 pts] Would it be correct to modify both the producer and the consumer as shown below, moving the mutex operations so as to treat the whole code block as a critical section? Explain either why this is correct, or if not, what goes wrong.

Producer (modified)

```
            mutex.acquire();
            empty.acquire();
            . . .
            full.release();
            mutex.release();
```

Consumer (modified)

```
            mutex.acquire();
            full.acquire();
            . . .
             empty.release();
             mutex.release();
```

No. With this modification, a deadlock can arise very easily. Imagine that a producer acquires the mutex but the buffer has no empty slots. It will wait on the empty.acquire semaphore but won't release the mutex first. No consumer can enter hence the code deadlocks. The same can occur in with a consumer attempting to pull data from an empty buffer: it waits for some slot to be full, but similarly retains the mutex lock.

b) [10 pts] Assume there is a single producer and a single consumer thread using the original, unmodified code, with N=20. The threads have been tested in isolation and run at *exactly the same rate*. Now, suppose the two are launched simultaneously in a process on a dual-core computer, with each thread is assigned its own core. Can the buffer ever fill completely? If yes, tell us what factors might let this happen; if no, explain why.

Yes, and it may even be likely. Recall that a traditional single-core scheduler doesn't need to give exactly the same amount of time to each thread – and it can give a "block" of time (called a quantum) to first one thread, then the other. So on a single-core system, the scheduler would simply need to give the producer enough time to fill the buffer before it happens to schedule the consumer.

Now when we move to a multicore system and dedicate one core to each thread, it would be reasonable to expect that each thread will execute continuously and hence that this kind of scheduling effect can't arise. But there are still conditions that can cause a thread to be temporarily descheduled. A good example is paging: if either thread takes a page fault, it may be delayed for a while and yet the other thread could keep running. A system call can have a similar temporary impact and even if the other thread is also doing system calls at the same rate, nothing in the problem statement says that it happens in perfect synchrony. And one could also imagine that there are other processes on the dual-core computer beyond the one we're focused on. So one or both cores could sometimes be temporarily assigned to a different process.

In summary: scheduling effects could definitely have this consequence.

3. [20 points] Here's a classic "merge-sort" algorithm for an input vector of length 2^k for some k. We didn't include the code for merge but it works just as you would expect.

```
// Break the data vector in half.  Sort the first half, then the second half, then merge
// the two sorted vectors
public void sort(int[] data, int start, int len)
{
        if(len == 1) return;                        // Vector of length 1 is already "sorted"
        int l2 = len/2;                             // … otherwise, compute half-way point
        FORK sort(data, start, l2);                     // Sort the first half
        FORK sort(data, start+l2, l2);                  // Sort the second half
        WAIT;                                           // Wait until both sorts are done
        Mutex.acquire();                            // Protect "merge" against reentrancy
        merge(data, start, l2, start+l2, l2);       // Merge the two sorted pieces, in place
        Mutex.release();
}
```

For our purposes, it seems adequate to fork off the sort procedures as threads, then to wait until both parts are sorted. If we could peek inside of the merge procedure we might be able to come up with a fine-grained locking scheme to let merge run concurrently with sort, but we're not permitted to do so. I've also added a mutex lock to protect s"merge" against reentrancy to be safe (although most ways of coding merge would be safe executed concurrently in distinct threads, one could imagine versions that would malfunction).

Solutions that add locks to get the desired effect are also fine with us (if correct).

4. [20 points: 5 per answer] After years of living a clean and virtuous life at Cornell, your computer becomes infected with a virus. It remains infected for a few days, after which your virus scanning software detects and removes it. Assume that the scanner was successful and the virus is really gone, and that there was no other virus on your machine. Now consider the following statements and, for each, indicate whether it is true or false:

a) The virus-writers may have copies of everything that was on the machine, a log of every web page you saw, email you sent or received, and everything you typed in (such as passwords on web sites)

Absolutely – many viruses install logging components that capture every keystroke you type and from this it would be a tiny step to log everything else.

b) The virus-writers may have a complete copy of all I/O that occurred since your machine was purchased (ie everything that was ever typed on the keyboard, everything that was ever read or written to the disk, every network message, etc).

No, this is not possible. Modern systems don't track all I/O and without logs of what was done in the past, the virus has no chance of getting that information.

c) The virus may have downloaded files onto your system (including child pornography or terrorist-related materials or other illegal content), and they could still be there.

Absolutely. When a file is downloaded there is no way to know if the site was visited by you as opposed to by the virus pretending to be you. Later when the virus scanner deletes the virus, if the scanner vendor was well aware of the tendency of this virus to download that sort of material, it might check for such files and delete them. On the other hand, if the scanner has no way to know and no idea what to check for, it definitely could delete the virus and leave the files.

d) The virus writers may have extracted the secret keys from the trusted platform module (TPM) on your machine

A TPM is that it is hardware device with keys burned into it that can only be accessed through a narrow, well defined hardware interface. Unless the TPM itself is defective a virus can't extract those keys – in principle, no known method can extract them.

5. [20 points: 5 per answer] A multithreaded application runs a single-core machine that with a 2GHz clock speed and an O/S that uses preemptive thread scheduling. The application has been moved to a four-core machine in which each core runs at 750MHz. Although all four courses are assigned to the application, it slows down by 20% (but it still runs to completion). The application consists of a single process with many (perhaps as many as 50) threads. Nothing else is running and the two machines are identical in all other ways. The application had no bugs and, when executed on a single-core machine, is provably free of deadlock and livelock.

For each of the following, could it have caused the issue?

a) Lock contention. *Absolutely. With multiple cores and different threads now running concurrently, there may be much more contention for locks than there was with a single-core version that used scheduling to context switch among threads. This lock contention can slow things down drastically.*

b) TLB flushing. *No. There is no need to flush the TLB when context switching between threads in the same process – the same address space.*

c) Virtual memory thrashing. *Perhaps. The larger number of threads, running all at once, may be simultaneously allocating lots of memory – memory that in a single-core setting would not have been needed all at once. As the demand for memory rises, the working set expands and could exceed the resident page limits of the O/S – triggering thrashing.*

d) A livelock has arisen inside the application. *No: If the application used preemptive multithreading and was proved not to have livelocks, this remains true when we move to a multicore platform. So this can't be the cause.*

6. [20 points] NASA has given you a disk that contains images taken by the Hubble space telescope. You need to write an application that will search these images looking for previously-unknown asteroids. There are 100 images per hour, 24 hours a day for a 10 year period: 8.76 million in total. The file name is the date and time of the photo, for example "100492-03:16" (May 10 1992, 3:16am). Each image is compressed and the typical file is about 4Kbytes in size. The total size of the disk is 50Gb, and it contains a single file system, with all of the files in a single directory called "/img".

The machine has many cores, and you have decided to use 10 at a time for your search. You are considering:

Option A: write a non-threaded (heavyweight) process that searches one file at a time, and then run it 10 times in parallel, one core per process.

Option B: write a single heavyweight process with ten threads in it, and then run the code from option A in these threads. Assign 10 cores to the resulting application.

You've picked option B. Having implemented the application, you profile it and discover that the file open operation is extremely slow.

a) [10 points] Explain what might be causing the slow file open times.

With so many files, searching the directory itself would be pretty slow. Basically, systems like Linux do a simple linear-time search, top to bottom, to find files.

b) [10 points] Propose a solution to the problem you identified in part c.

The simplest solution would be to just structure the image directory with subdirectories, perhaps organized by year/day or year/day/hour. This imposes a log-time search down to the level of the folder with the actual files in it, which will still be scanned top-to-bottom, but done this way there would only be a few files in that leaf-level of the hierarchy. So a linear scan would be quite fast.

7. [20 points: 4 per answer] TCP protocol.

a) Sketch the classic TCP throughput curve and label the "slow start", "additive increase" and "multiplicative decrease" portions.

b) In (a), what triggers TCP into switching from additive increase to multiplicative decrease?

Packet loss: TCP keeps increasing the sender window size and hence sender throughput rate until a loss occurs, then halves the window size. The receiver senses the loss and sends a NAK, so from the sender's point of view the most accurate answer is "reception of a NAK".

c) Describe a pattern of communication in which TCP would send packets at a *constant rate* without any change at all. (That is, tell us how you would design a TCP application so that if someone was watching the network they would see one packet go past at a time with a fixed interpacket spacing that doesn't change).

Obviously, for TCP to increase the sending rate, the protocol needs to have an unbounded source of data to send. If the producer is sending at a fixed rate and TCP doesn't experience loss at that fixed rate, TCP will settle into sending at that same rate.

d) Describe a situation in which a student could give out the IP address of his own machine, and yet friends would be unable to connect to a game server he is hosting on his machine, just as if the IP address wasn't in use at all on the Internet.

This can easily happen if the student has a firewall that also does network address translation. From the outside, the group of machines behind that NAT box look like a single machine with a single IP address (the IP address of NAT box). Attempts to talk to that "internal only" IP address just won't make sense except to other machines behind the same firewall – those addresses aren't visible outside the firewall.

e) Suppose that an intruder were to "wiretap" the web connection out of Cornell, making a copy of every TCP-produced packet. Would that individual be able to reconstruct entire emails and web pages and files? If so, what would they need to do; if not, why not?

Yes, they could do this. They would basically "sort" the IP packets roughly by time of day and then, within a period of time, by TCP session (source IP address and port, destination IP address and port) and reassemble the TCP stream. At this point everything sent in plain text over TCP becomes visible to the attacker,

8) [20 points: 4 per answer] Cryptography

a) Harry and Sally need to arrange a very secret meeting via email. Is there a way to do this so that nobody except the two of them can read the messages they exchange?

Sure. As we saw in class, Harry encrypts his email with his private key and then with Sally's public key, then sends it to Sally; she does the same with her private key and his public key to reply. To decrypt, Sally applies her private key and Harry's public key, and vice versa. This works because the intruder doesn't have the private keys.

b) Explain what a "digital signature" is and how it differs from an "encrypted message"

A digital signature works by computing a smaller "message digest", usually 128 bits long (but it could be any length you like), then encrypting this small object. The message itself is sent in the clear, but the receiver can detect any tampering because when he recomputes the digest, a modified message will won't match the one in the digital signature.

c) The security of the famous RSA public-key cryptographic scheme depends upon one basic assumption that isn't actually known to be true. What is that assumption?

The assumption is that a very large number formed as the product of two very large prime numbers can't be factored in a practical way. If we could factor large numbers we could easily (trivially) compute the private key corresponding to a public key.

d) Why do protocols like HTTPS use asymmetric (public key) cryptography to connect to a web site, but then switch to a symmetric key cryptographic method, rather than just using the asymmetric scheme for the entire session?

Asymmetric cryptography is fairly slow but very secure, so we use the slow scheme to exchange a shared (symmetric) key and then use faster symmetric cryptography. This also makes it easy to share the keys – a public key scheme is incredibly powerful in that way.

e) Some experts believe that we can never eliminate the role of passwords that people need to type when challenged. What can a password do that public key infrastructure can't?

One can steal digital keys in many ways… in the limit, an intruder can just steal your computer or insert a virus into it that pretends to be you. But a password is a thing that you "know" and that isn't stored online. Thus as long as we also ask for passwords, a virus can

only get so far without somehow also stealing that password. In theory, a password you type when challenged thus proved that there is a "you" physically at the terminal.