CS414 Final Exam, Version 1 (December 2, 2004)
75 Minutes, closed book. Five questions, 20 points each

1. Below is the code for a class supporting keyed binary trees. The system adds nodes to the tree as it learns new keys (it never deletes a node), and stores a value for each key. The caller can save (string,value) pairs, and can lookup a string, obtaining its value, or -1.0 if the key is unknown:

```
public class BinaryTree {
     public class BinaryTreeNode {
          string         NodeName;
          double         NodeValue;
          BinaryTreeNode Left;
          BinaryTreeNode Right;

          public BinaryTreeNode(string s, double v)
          {
               NodeName = s;
               NodeValue = v;
               Left = Right = (BinaryTreeNode)null;
          }
     }

     BinaryTreeNode root = new BinaryTreeNode("***");

     public void Save(string s, double v)
     {
          Save(root, s, v);
     }

     void Save(BinaryTreeNode n, string s, double v)
     {
          if(n.NodeName == s) {
               n.value = v;
               return;
          }
          else if(n.NodeName > s) {
               if(n.Left == (BinaryTreeNode)null)
                    n.Left = new BinaryTreeNode(s, v);
               else
                    Save(n.Left, s, v);
          }
          else {
               if(n.Right == (BinaryTreeNode)null)
                    n.Right = new BinaryTreeNode(s, v);
               else
                    Save(n.Right, s, v);
```

```
        }
    }
    public double Find(string s)
    {
        return(Find(root, s));
    }

    double Find(BinaryTreeNode n, string s)
    {
        if(n == (BinaryTreeNode)null)
            return(-1.0);
        if(n.NodeName == s)
            return(n.NodeValue);
        else if(n.Name > s)
            return(Find(n.Left, s));
        return(Find(n.Right, s));
    }
}
```

a. Suppose that this class is called from a threaded program. Would it be possible for a (string,value) pair inserted using "Save" to be lost, so that a subsequent "Find" operation is unable to find the value? *(Note: Find is called after Save returns… but there might be other concurrent calls to Save or Find.)* Explain your answer.

Yes. There is a race condition in this code, in the Save routine. If Save is called concurrently two invocations might both try to set the n.Left or n.Right field of the same object and one of the two new nodes would simply be lost.

b. Now suppose that we wish to use semaphores to protect against concurrency-related bugs. Our goal is to obtain the highest possible degree of concurrency. Which of the following approaches would you recommend? Indicate why the mechanism you propose is correct, and why you think it offers the most concurrency. *(Note: if Find is called concurrently with a Save for the same key, it is ok for the Find to return the old value or even -1).*

i. Associate a single semaphore (a "lock") with the entire tree. A Save or Find operation first locks the tree, then runs the code seen above, then unlocks it.
ii. Implement the readers and writers solution, using semaphores. A Save operation does a StartWrite, then runs the code for Save, then calls EndWrite. A Find operation does a StartRead, runs code for Find, then calls EndRead.
iii. Associate a semaphore (a lock) with each Left and Right pointer (hence each BinaryTreeNode will have two semaphores). Obtain the lock before accessing the corresponding field (n.Left or n.Right) and then release it after accessing the field. Do this in both Save and Find.
iv. Same locking mechanism as in (iii), but only obtain a lock in the Save() code. Release the lock after inserting a new node, or before calling Save recursively, depending on which way the test for a null node goes.

Among these options, <u>any of them</u> would eliminate the bug we saw. Option i works correctly, but has the undesired effect of eliminating all our concurrency: only one thread at a time can access the tree. So that's overkill. Option ii would also work, but also reduced concurrency more than we need to because we really only need to worry about the case where two calls to Save are attaching a new node at the same place on the tree. Option iii and iv thus seem to be the best choices. Option iv would be my favorite, since a Find will either see a null field or will see a pointer to a valid node. With option iii, we force Finders to get and then release a lock and this isn't really necessary.

So I would go with option iv.

2. The Windows system uses what are called "display handle" objects to permit concurrent access to a computer's display. The computers in the CSUG lab are running Windows, and have a hardware limit of five display handles in simultaneous use. Accordingly, you've coded your very concurrent program so that before doing a set of I/O operations to update the display, a thread will call DisplayHandle GetHandle(), and after doing the I/O will call ReleaseHandle(DisplayHandle h). GetHandle returns the handle the calling thread can use for its I/O, and ReleaseHandle frees that handle so some other thread can use it. Implement a Monitor called Handles that manages an array of five handles and supports these two methods. Initialize the five handles by calls to InitializeHandle(). InitializeHandle is very slow and should only be done once per handle. *(Note: if your solution calls InitializeHandle more than five times, over the entire run of the program, you've coded it incorrectly!)*

```
Monitor Handles {
     DisplayHandle[] Handles = new Handles[5];
     Boolean[] is_in_use = new Boolean[5];
     Condition WantHandle;

     public Handles()// constructor
     {
          for(int i = 0; i < 5; i++)
               Handles[i] = InitializeHandle();
     }
     public DisplayHandle GetHandle()
     {
          if(in_use == 5)
               WantHandle.Wait();
          ++in_use;
          for(int i = 0; i < 5; i++)
               if(is_in_use[i] == false)
               {
                    is_in_use[i] = true;
                    return Handles[i];
               }
          return (DisplayHandle)0; // Never happens
     }
     public ReleaseHandle(DisplayHandle h)
     {
          for(int i = 0; i < 5; i++)
               if(Handles[i] == h)
               {
                    is_in_use[i] = false;
                    break;
               }
          --in_use;
          WantHandle.Signal();
     }
}
```

3. You were hired by AlphaBeta corporation to help improve performance of their software for predicting stocks to buy and sell. The key program is executed once per hour and spends about half an hour running; it then writes out some files which are used for AlphaBeta over the next hour. If the runtime can be improved, AlphaBeta will be using more accurate data and hence will earn more money, and you'll get a big holiday bonus.

    a. You start by using "vmstat" and "pstat", and determine that the program is doing about 5 page-in or page-out operations per second over the course of the hour – and that most are page-out operations. The CPU is 99.6% busy. The virtual memory size of the program is 50MB, and the machine has 65MB of available memory on it. Explain the probable cause of the paging activity we're seeing. Is the program thrashing? *[You may assume that the disk can do 75 I/O operations per second, that the operating system implements the "working set" virtual memory algorithm (more specifically, the WSCLOCK algorithm), and that there are no other programs running or I/O's occurring while the program runs.]*

    This doesn't sound like thrashing: the disk isn't really all that busy and is mostly doing page-out operations, not page-in operations. Most likely the issue is that in Working Set we page something out if it isn't touched for ? time units. Perhaps these are simply pages that haven't been touched for a while.

    b. Using "gprof," you identify a central compute loop in which the program spends 92% of its compute time. It turns out that most of this time is spent recalculating some parameters that could have been computed once and saved in a table. The new table takes 20MB of memory. By recoding, and doing a bit of additional optimization, you are able to speed this section of the program up dramatically. Runtime drops from 30 minutes to 3 minutes, but now vmstat shows that the program is doing 50 paging I/O operations per second (they are still mostly page-out operations). Is the program thrashing now? Explain.

    I still wouldn't call this thrashing but clearly the overhead of the VM algorithm has become a significant problem and if you want to say that this <u>is</u> thrashing we'll accept the answer. Either way you cut it, the program is now spending a high percentage of its time doing page-out operations.

    c. Your boss was impressed by your work and is wondering whether the program might be able to run even faster. She purchases 100MB of additional main memory for the computer, increasing the total to 165MB, yet the paging rate doesn't change. She asks you if it might make sense to recompile the operating system to use the LRU paging algorithm instead of WSCLOCK, or to increase the "working set window" size. What will you tell her? Would either change be likely to help?

    Either change might work. With a larger working set window size we might realize that these pages don't actually need to be written out (since page-in rates are so low apparently they <u>do</u> eventually get touched again). LRU would probably eliminate this unneeded paging entirely.

4. Program A writes a file over the network on file server S. Then it sends a message to program B, telling it to read the file. (A, S and B run on three different machines). Yet B sees the old file contents, not the new contents! Explain how this can happen and what could be done to correct the problem.

   This is an easy problem. Due to "write-through caching" the changed file is apparently still in the cache on machine A. A should call the fsync system call (I don't mind if you can't remember the name of the system call as long as you remembered that there is such a call), or just close the file. By the time the O/S returns from the fsync or close operation, the file should be current on the file server and B can safely read it.

5. You are implementing a distributed computing system to capture video images from an airplane and transfer them into a file server on the ground. The system consists of your "photo capture" program, a TCP link to the file server, and a "photo store" program, and is completely idle other than running your code. The video images total several gigabytes of data. Your program transfers images over this single TCP link: first it sends a small header giving the file name and size, then the data, and then it loops for the next image. The connection from the plane to the file server has 10Mb performance and the measured packet loss rate is extremely low – over a period of more than one hour, not a single packet loss is observed.

    a. While your program is running, you instrument the performance of the network connection and discover that your TCP connection isn't running at a steady rate. Instead, TCP seems to send data faster and faster, then suddenly slows down again, then speeds up again. What might be causing this behavior? Is it a bug?

    This "sawtooth" behavior is standard for TCP. TCP ramps its speed up linearly until packet loss occurs, then cuts the transmit rate in half and does that forever, hoping to detect opportunities to send a bit more data.

    b. Your boss installs a firewall between A and B. The firewall is theoretically able to run at 100Mb per second (10 times the rate of the link from the airplane), but due to a glitch in the firewall hardware, now and then it drops a packet. When you work out the bandwidth available, accounting for the packet loss, you calculate that even with the loss it still runs at much more than 9.95Mb, which is the expected maximum rate for the link. But now TCP seems to be running at *half the speed* relative to what it was doing before the firewall was installed: around 5Mb per second! Why should such a minor data loss rate cause such a big slowdown for TCP?

    This sort of problem fools TCP into thinking the network is overloaded. It cuts the sending rate in half repeatedly each time it sees a packet loss.

    c. In the process of tracking the performance problem just mentioned, you look closely at the packets on the network. You notice that whereas previously they had the IP addresses of A and B in them, now the addresses of A have been replaced by some other IP address. Is this a sign of a problem?

    No, that's a normal feature of firewalls that do "network address translation", which most firewalls incorporate as a standard thing now. It shouldn't prevent A from talking to B as long as A is the guy inside the firewall and B is on the outside. And in fact we already know that TCP is getting through the firewall from parts a and b of the problem. So apparently the firewall isn't blocking TCP and the address translation is totally unrelated to the lousy performance.