Your Name: __SAMPLE SOLUTION SET_______________ NetID: _____

CS414 Fall 2003 Final Exam (Version 1)

1. [25 questions, 2 points each, for a total of 50 points]

	True	**False**
a) Consider a CSEnter/CSExit implementation based on this code: "while(test_and_set(flag) == true) loop;" This algorithm is deadlock-free.	**X**	
b) The Bakery Algorithm can livelock if two processes happen to pick the same "number" value. (Hint: code for algorithm is in problem 2).		**X**
c) In the Bakery Algorithm if process *i* is looping in the "while(choosing[k]) loop," process *k* always enters and exits the critical section before process *i*. (Hint: code for algorithm is in problem 2).		**X**
d) In a semaphore-based implementation of the bounded buffer algorithm, as many as *b* producers can concurrently enter the critical section to add an element to the buffer, assuming that the buffer can hold *b* objects.		**X**
e) In a semaphore-based implementation of the readers and writers algorithm, many readers can concurrently obtain read access to the protected object, but only one writer can access it at a time.	**X**	
f) A language that supports monitors automatically handles mutual exclusion. Thus there is no separate CSEnter/CSExit mechanism.	**X**	
g) The Banker's Algorithm is deadlock-free because it breaks the "no preemption" condition, one of four necessary conditions for deadlock.		**X**
h) A machine might be addressable by its IP address on one side of a network address translator (NAT), but not addressable, at all, from the other side.	**X**	
i) In the Internet, the route taken from machine *i* to machine *j* is the reverse of the route from *j* to *i*.		**X**
j) UDP leaves flow control to the application developer, whereas flow-control is built into the TCP protocol.	**X**	
k) Whereas UDP imposes a size limit for messages sent, TCP imposes no size limit at all – any object a process can hold in memory can be sent.	**X**	
l) If neither end-point crashes, a TCP connection is reliable: it hides any packet loss, duplication or out-of-order packet delivery from the application.		**X**
m) Akamai uses the DNS to load-balance by "mapping" a single host name to the closest lightly loaded server in its pool of servers.	**X**	
n) A demand paging algorithm only pages something out when a page fault occurs.	**X**	

o) If the working set (WS) algorithm is compared with LRU, using the same value for Δ, WS often requires less physical memory to run a given program.	**X**	
p) Spatial locality is defined as a situation in which a program repeatedly accesses a small set of pages in the working set.		**X**
q) If the WSCLOCK algorithm makes a mistake and something it pages out is needed again, the page may still be in the reclaim pool.	**X**	
r) The pages of a mapped file must be pinned into memory because normal virtual memory management algorithms can't be applied to mapped files.		**X**
s) We can simulate the page "reference" and "dirty" bits using memory protection hardware and taking an interrupt the first time a page is accessed and/or written. An advantage is that the hardware to manage the TLB can be simplified, and it can be flushed without writing PTEs back to the page table.	**X**	
t) Suppose the trigonometry functions on a computer are in a DLL, and program "render" has been linked with that DLL. Now suppose that two instances of render are running simultaneously (on the same computer but as different processes). If they print the address at which the arctan function resides in memory, they may print different addresses.	**X**	
u) An inode includes a count of the number of <u>symbolic</u> links pointing to the file.		**X**
v) A disk buffer pool may delay writing a block of a file back to disk, hence if a computer crashes soon after a program updates a file, the update can be lost.	**X**	
w) As the NFS is normally configured, it trusts the IP address of a machine accessing it, the user-id, and the group-id information in each request.	**X**	
x) Recursion and reentrancy are basically the same idea. In particular, any recursive procedure is reentrant, too.		**X**
y) If a program wasn't written to be multithreaded and you add even a single additional thread, it is important to make sure that every procedure is either reentrant, or can't be reentered.	**X**	

Your Name: __SAMPLE SOLUTION SET_______________ NetID: _____

2. [20 points] Below is the Bakery algorithm for N threads, using the same code and notation we employed in class. Recall that (a,x)<(b,y) means "(a < b) or (a=b and x<y)".

```
#define         true            1
#define         false           0
unsigned byte number[N] = { 0, …. 0};
boolean         choosing[N = { false, …. false };

Process Pi:
        while (1) {


                do something else


            choosing[i] = true;
            number[i] = max(number[0]…number[N-1])+1;
            choosing[i] = false;
            for(k = 0; k < N; ++k) {
                    while(choosing[k]) loop;
                    while((number[k]!=0)&&((number[k],k) < (number[i],i))) loop;
            }


            Critical Section for Process i


            number[i] := 0;

        }
```

a) In class when we discussed the algorithm we assumed that a process that enters the critical section will always make "finite progress" and eventually exit the critical section. But what if this wasn't the case? For example, suppose that process *i* enters the critical section but then goes into an infinite loop while inside it. What would happen?

If a process can get stuck in the critical section, no other process will be allowed in. Other processes will pile up in the CS Enter code – in this case, they will be looping in the second while loop.

b) An issue with the algorithm, as shown here, is that the "number" can overflow. Suppose that number is an array of 8 bit unsigned integers and that process *j* experiences such an overflow: "max" returns 255, so process *j* picks number[j] = 0. Define *safety* very briefly and indicate whether safety could be violated in this situation. If not, explain why, and if so, show us how it happens.

Safety can definitely be violated. Process j has number[j]=0, but the algorithm will interpret this as an indication that j is not in the critical section <u>and</u> <u>not</u> <u>trying</u> <u>to</u> <u>enter</u>! Thus j can enter (it has the smallest possible "number" value) and any other process will be able to enter too, while j is still inside.

c) Suppose that you are hired by a company doing a new operating system for a uniprocessor. You are asked to implement critical sections for a user-level threads package (e.g. not in the O/S itself). The threads package will in turn be used to implement a programming language that supports monitors. Would the Bakery algorithm be your first choice? Why?

On a uniprocessor, the only concurrency is due to explicit scheduling. A busy-waiting algorithm such as the Bakery Algorithm would be a poor choice because it would waste CPU time: if p_i wants to enter the critical section and must wait for p_j to exit, we don't want p_i to sit in a loop. We would prefer to context switch to p_j as soon as possible!

(Obviously, you would also object to the use of potentially buggy code associated with overflow in the code shown here, but there is a simple way to fix that, and this isn't the answer we were looking for.)

d) Suppose that when process *j* calls CSEnter, process *i* is already inside, and that number[i] = 25. Process *j* picks number[j] = 26 and is "about" to execute chosing[j] = false. Is it possible that process *i* could leave the critical section and yet reenter it before *j* is able to do so? Explain.

Yes, it is possible. Suppose that we context switch away from p_j leaving it "about" to chose number 26. Process p_i now leaves the critical section, setting its own number to 0, zips to the CS Enter code, and executes it. If nobody else has chosen a number so far, p_i will pick number 1! Now, p_i has number[i]=1, and p_j will eventually finish setting number[j]=26. So p_i can get back in (once) before p_j.

There is also another scenario in which p_i can pick 26, just as p_j did (namely, if some other process p_k hasn't entered yet and had number[k]=25). In that case, i gets in before j if i<j.

Any correct scenario will get you full credit.

3. [25 points] Write a monitor to solve the following synchronization problem.

There are three threads in an application:

- **Threads A and B produce endless streams of objects of types a, and b respectively.**
- **Thread C is a rendering application: it waits for one object each from A and B, (one of each type), then renders them on the screen, then goes back for more.**

Your monitor should have two entry points:

- **produce_object(object o, string thread_id), where "thread_id" will be either "A", "B", and object o is of type a or type b, respectively.**
- **consume_objects(out object oset[2]). Waits for two objects, and then returns them in the "out" variable oset, setting oset[0] = obj_a, and oset[1] = obj_b.**

To maximize performance, design the monitor to allow the producer threads to get a little ahead of the consumer, but not enormously. In particular, a parameter K is predefined with

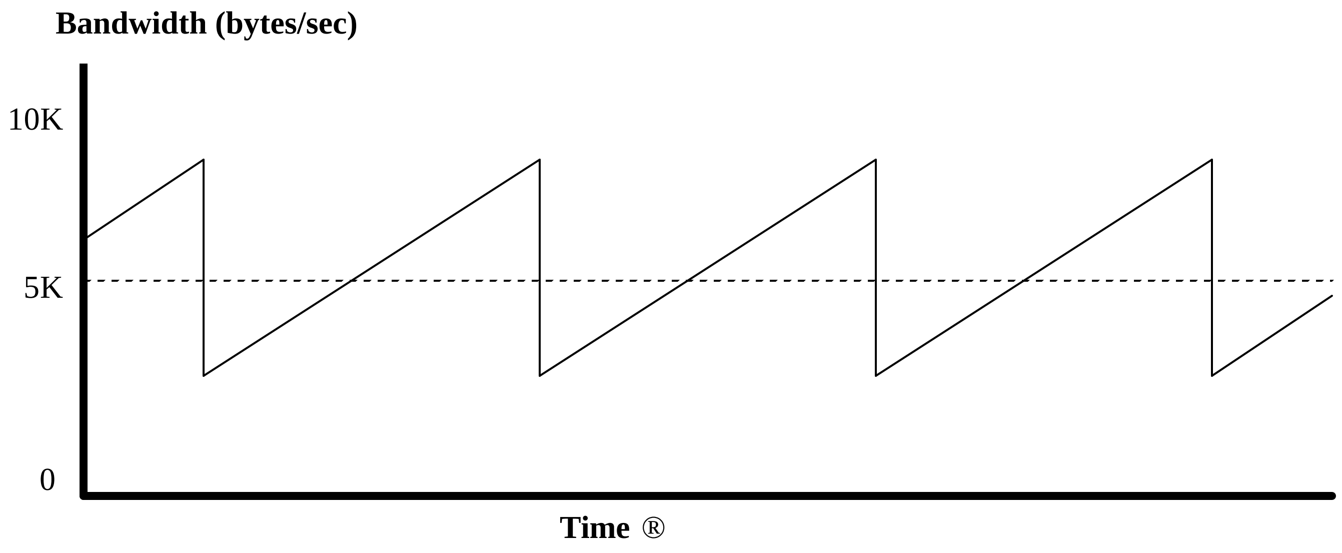
some small value, like 5, and the monitor shouldn't block a producer thread unless it gets more than K objects "ahead" of the consumer.

```
/*
 * This is just a standard producer-consumer monitor
 * except that it has two side-by-side buffers.
 */
monitor prodcom {
   int cnt_a = 0, cnt_b = 0;
   int ao_pt = 0, ai_pt = 0;
   int bo_pt = 0, bi_pt = 0;
   condition needa, needb, havea, haveb;
   object buf_a[0..K-1], buf_b[0..K-1];

   void consume_objects(out object oset[2]) {
        if(cnt_a == 0) needa.wait();
        if(cnt_b == 0) needb.wait();
        oset[0] = buf_a[ai_pt++ % K];
        oset[1] = buf_b[bi_pt++ % K];
        cnt_a--; cnt_b--;
        havea.signal();
        haveb.signal();
   }

   produce_object(object o, string thread_id) {
        if(thread_id == "A") {
             if(cnt_a == K) havea.wait();
             cnt_a++;
             buf_a[ao_pt++ % K] = o;
             needa.signal();
        } else {
             if(cnt_b == K) haveb.wait();
             cnt_b++;
             buf_b[bo_pt++ % K]  = o;
             needb.signal();
        }
    }
}
```

Your Name: __SAMPLE SOLUTION SET_______________ NetID: _____

4. [5 points]. You are hired to help tune up the performance of a Web server program that sends documents over TCP. Using a TCP monitoring tool, you carefully measure the speed of data transfer between the server and a client. After discarding the first 30 seconds of data to let things "settle down", the throughput works out to 5K bytes/second (the dashed line). Yet when you graph the speed of the connection for the next few minutes, you obtain the graph shown below. Explain whether the "erratic" behavior evident in the graph is a sign of a problem that needs to be fixed. What could cause this sort of erratic throughput? Given that the network is in a stable state, why is the performance so variable? Be brief!

The graph shows typical behavior for TCP. The protocol constantly tries to push its bandwidth up, and it does this by incrementing its "window size" (in effect, its sending rate). But as the rate rises the connection will eventually become overloaded or a router will overload and data will be lost. When this happens, TCP halves its sending rate. So you get the sawtooth seen in the graph.


Bandwidth (bytes/sec)
10K
5K
Time ®