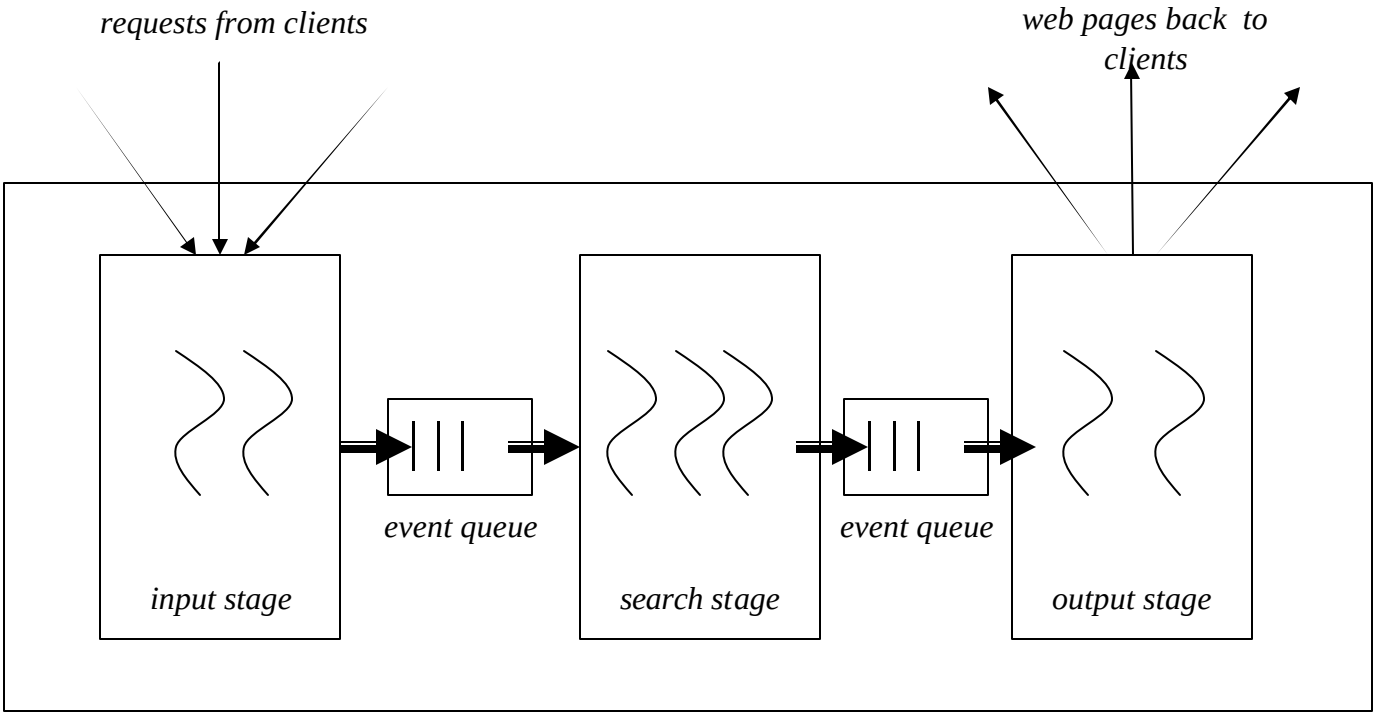
CS414 Prelim 2. November 9, 2004
4 questions, 25 points each. 75 minutes (10:10-11:25)

1. **One way to implement a multi-threaded Web site application is to structure it in stages, as follows.**

   A *stage* consists of a set of one or more threads that all perform the same task. For example, an "input" stage might have a set of threads that accept incoming requests from the network; these would be turned into events by the runtime library. A "search" stage might be a set of one or more threads that look for information useful in performing a requested task, and an "output" stage might have one or more threads that build web pages to send back to the users who issued requests.

   Threads in one stage communicate to threads in another stage by sending small messages called "events", and there is an event-queue in between any two stages that ever communicate. An event queue is unidirectional (one stage produces events and the other consumes them) and has the capacity to hold some bounded number of events. *Threads don't "wait" for responses:* a thread typically sits in a loop, waiting for an event to come into its stage, processing that event, perhaps generating additional events or sending messages over the network, and then looping. Threads don't talk directly to other threads.

   In what follows, we'll assume that threads execute in a shared address space within a single heavy-weight process and that they use monitor-style synchronization when accessing shared variables.



(a) Using monitors, write a class called EventQueue with methods "enqueue(event e)" and "event dequeue()" so that the greatest possible degree of concurrency will be achieved. The application will create an instance of the EventQueue class between each pair of stages that needs an EventQueue . The EventQueue constructor should accept one argument, namely an integer telling the size of the associated EventQueue buffer.

```
class EventQueue{
        //the queue of events
        Event[] events;
        int size;
        condition NotFull, NotEmpty;
        //Number of events already in the queue
        int nItems;
        //locations in the array to/from which the next enque/deque is done
        int enqIndex, deqIndex;

        EventQueue(int theSize){
                size = Size;
                nItems = 0;
                events = new Event[size];
        }

        enqueue(event e){
                while(nItems==size){
                        NotFull.wait();
                }
                //now can safely enqueue item
                events[enqIndex] = e;
                enqIndex = (enqIndex+1)%size;
                nItems++;
                //wake up a waiting dequeuer, if any
                NotEmpty.signal();
        }

        Event dequeue(){
                while(nItems==0){
                        //this should not happen here though, since threads do not wait
                        //for events from other events; so could raise an error here
                        NotEmpty.wait();
                }
                //now can safely deque item
                Event toReturn = events[deqIndex];
                deqIndex = (deqIndex-1)%size;
                nItems--;
                //if queue was full previously and an enqueuer was waiting,
                //wake it up.
```

```
                NotFull.signal();
                return toReturn;
        }
}
```

(b) Suppose that the application can have processing loops, so that stage A sends events to stage B, stage B to stage C, etc, and eventually events "triggered" by stage A might get sent back to stage A. Given that threads *do* send events but *don't* wait for any responses, can a deadlock arise? Explain why or why not, making reference to the "necessary conditions for deadlock" as application.

Deadlocks cannot arise here. Since threads never wait for *specific* events from other threads, neither the hold and wait condition nor the circular wait condition is ever satisfied.

2. In class we learned about IP addressing in the Internet. For each of the following, write a single sentence to indicate if the claim is true or false, and why.

a. Claim: "Suppose that machine A is communicating with machine B on the Internet. There are situations in which the IP address used by A to refer to B might be different from the IP address used by B to refer to itself."

Answer: *True*. Machine B could be inside a firewall, and A outside. In such a scenario, when B talks to A the firewall could translate B's internal network address into a different external one.

b. Claim: "No matter how fast an Internet connection is, if a file is fairly small, TCP might not send data at the maximum data rate for that connection."

Answer: *True.* TCP would still be ramping up its sending rate, probably far below the maximum data rate, when the file transfer is done. This relates to TCP "slow start".

c. Claim: "When sending a long file over the Internet, TCP increases the sending data rate until it detects packet loss."

Answer: *False.* Indeed, TCP stops increasing the window size when the data loss is detected. But TCP also limits its rate on the basis of both the receiver's window size, and the sender's sending capacity. We would need to be certain that these are not the bottlenecks in order to safely conclude that TCP will increase its sending rate until a packet loss is detected.

d. Claim: "When TCP does detect a packet loss (as in part c), it reduces the data rate to the last rate at which data successfully got through, and then holds the sending rate constant thereafter."

Answer: *False.* TCP uses what is called a "linear increase, multiplicative decrease" policy to manage window sizes. Specifically, when sending successfully, TCP periodically adds a constant amount to the window size: the window grows linearly in size over time. But when a loss occurs, TCP normally halves the window size, and if more losses occur soon after it will halve the size again, and so forth. Thus the "last rate at which data got through successfully" doesn't really enter the picture, at least not in a direct sense.

e. Claim: "Sometimes, Internet addressing would allow machine A to make a connection to machine B, but won't allow machine B to connect to machine A."

Answer: ***True.*** A might be a machine that needs to be protected from incoming connections from the internet, so it might be inside a firewall.

f. Claim: "When people talk about "round trip" data rates and delays in the Internet, they are incorrectly adding together the outgoing and the incoming numbers. Since these numbers can be very different, it is better to measure the two delays separately (if one can do so) and quote two numbers: the "uplink" data rate and delay, and the "downlink" rate and delay.

Answer: ***True.*** Typically, the upload data-rate is significantly smaller than the download data-rate on home machines, so measuring the two separately gives a more accurate picture.

g. Claim: "Remote procedure call mechanisms, such as the Web Services remote method invocation feature provided by Visual Studio for C# or Java's RMI, make interprocess communication over a network look very similar to invocation of a procedure. However, unlike a traditional procedure invocation, there may be restrictions on the kinds of arguments that can be passed (because arguments must be represented by value and fit into messages), and there can be failure modes (like timeout) not normally seen when local procedures are invoked."

Answer: ***True.*** Arguments will have to be serialized into a sequence of bytes, so as to be sent across to the other side. And, since this communication takes place over a network, there might very well be packet losses/delays etc. that might lead to timeouts and will have to be handled appropriately.

h. Claim: "Because the DNS caches name-to-IP-address data, if an application updates this mapping there may be a period during which messages continue to be sent to the old IP address."

Answer: ***True.*** This is possible, though more accurate timeouts (of the DNS entries) could be used to reduce the time periods over which this could happen.

i. Claim: "The Internet routing adapts in milliseconds if load changes on a link or if a router fails. As a result, if there is a path from machine A to machine B in the Internet, messages from A to B can be counted on to get through despite disruptions. Message loss occurs during periods when there is no route between A and B."

Answer: ***False*** It could take up to 100s of seconds for a link loss to be detected and handled by the internet routing. Packets might be lost even if there is a route connecting the source to the destination, with the routers still thinking that the earlier(broken) route is valid and pumping packets along the broken route.

j. Claim: "A lossy link, like a wireless link, can cause TCP to misbehave by shrinking its adaptive window and hence running very slowly."

Answer: ***True.*** TCP will react as if congestion is causing the packet loss, when in fact it is the lossy (lousy?) medium.

Below is are two memory reference tableaus for the same reference string. We assume a fixed sized memory partition of size D = 3 pages. Fill in these two tables, one for LRU, and the other for Working Set (WS, with "window size" D = 3). We filled in the part for t=1 to t=3. Tell us what the HIT RATIO and MISS RATIO was, as a fraction, over the 15 time units starting at time t=4

(a) LRU: Note: "time of use" is defined only over the time while a page is continuously in memory!

R(t)	**8**	**3**	**1**	**3**	**2**	**8**	**3**	**1**	**3**	**2**	**8**	**3**	**8**	**2**	**8**	**3**	**8**	**2**
S_0	**8**	**3**	**1**	3	2	8	3	1	3	2	8	3	8	2	8	3	8	2
S_1	Æ	**8**	**3**	1	3	2	8	3	1	3	2	8	3	8	2	8	3	8
S_2	Æ	Æ	**8**	8	1	3	2	8	8	1	3	2	2	3	3	2	2	3
IN(t)	**8**	**3**	**1**	Æ	2	8	Æ	1	Æ	2	8	Æ	Æ	Æ	Æ	Æ	Æ	Æ
OUT(t)	Æ	Æ	Æ	Æ	8	1	Æ	2	Æ	8	1	Æ	Æ	Æ	Æ	Æ	Æ	Æ

Hit Ratio = __10_____ **/ 15** **Miss Ratio =** _____5__ **/ 15**

(b) Working Set policy with working-set size D = 3 pages. Denote an empty memory slot by "Æ"

R(t)	**8**	**3**	**1**	**3**	**2**	**8**	**3**	**1**	**3**	**2**	**8**	**3**	**8**	**2**	**8**	**3**	**8**	**2**
S_0	**8**	**3**	**1**	3	2	8	3	1	3	2	8	3	8	2	8	3	8	2
S_1	Æ	**8**	**3**	1	3	2	8	3	1	3	2	8	3	8	2	8	3	8
S_2	Æ	Æ	**8**	Æ	1	3	2	8	Æ	1	3	2	Æ	3	Æ	2	Æ	3
IN(t)	**8**	**3**	**1**	Æ	2	8	Æ	1	Æ	2	8	Æ	Æ	2	Æ	3	Æ	2
OUT(t)	Æ	Æ	Æ	8	Æ	1	Æ	2	8	Æ	1	Æ	2	Æ	3	Æ	2	Æ

Hit Ratio = _____7__ **/ 15** **Miss Ratio =** __8_____ **/ 15**

Note: If we are actually using WSClock instead of WS, there is a further complication, because pages removed from the working set are briefly retained in the "reclaim pool" and hence sometimes a page fault won't actually need to page the missing page back in, which would make the hit ratio seem magically higher. But normal WS doesn't have that reclaim mechanism.

(c) Parts (a) and (b) point to a tradeoff between the amount of memory used by an algorithm and the hit ratio. If we are mostly concerned about performance, which algorithm do you prefer: LRU or WS?

If the system has enough physical memory, LRU keeps more pages in memory. Thus it has a potentially better hit ratio and would be the better choice if all we care about is speed. But if we have lots of applications and are short on physical memory, then WS is preferable, since it might let us keep more pages for one process and less than another by selecting just the working set pages to retain in memory (since the WS might be larger for one process and smaller for another). In this case, WS would be preferable.