Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2006

Quiz I

There are 12 questions and 12 pages in this quiz booklet. Answer each question according to the instructions given. You have **50 minutes** to answer the questions.

All questions are multiple-choice questions. Next to each choice, circle the word **True** or **False**, as appropriate. A correct choice will earn positive points, a wrong choice may earn negative points, and not picking a choice will score 0. The exact number of positive and negative points for each choice in a question depends on the question and choice. The maximum score for each question is given near each question; the minimum for each question is 0. Some questions are harder than others and some questions earn more points than others—you may want to skim all quesions before starting.

If you find a question ambiguous, be sure to write down any assumptions you make. **Be neat and legible.** If we can't understand your answer, we can't give you credit!

Write your name in the space below AND at the bottom of each page of this booklet.

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.
NO PHONES, NO COMPUTERS, NO LAPTOPS, NO PDAS, ETC.

CIRCLE your recitation section number:

10:00	1. Madden/Hu	
11:00	2. Madden/Seater	3. Rinard/Ports
12:00	4 Rinard/Seater	5. Walfish/Ports
1:00	6. Walfish/Winstein	7. Katabi/Hu
2:00	8. Katabi/Winstein	

Do not write in the boxes below

1-4 (xx/30)	5-10 (xx/46)	11-12 (xx/24)	Total (xx/100)

Name:

I Reading Questions

1. [6 points]: Which of the following statements is true for UNIX as described in reading #5 (Ritchie and Thompson. "The UNIX time-sharing system", Bell System Technical Journal, 57, 6, part 2, 1978)?

(Circle True or False for each choice.)

A. True / False The i-number of a file is a disk address.

B. True / False Directory entries contain the names of files and their corresponding i-numbers.

C. True / False Links may be made to directories.

D. True / False A pipe between two processes cannot be established after both have started.

E. True / False A parent process shares open files at the time of FORK with its children.

F. True / False A parent process knows the "processid" of its child process when FORK completes but not vice versa.

2. [8 points]: Which of the following statements is true of the X Window System as described in Reading #6 (Scheifler and Gettys. "The X window System", ACM Trans. on Graphics, Vol 5, 2, April 1986)?

(Circle True or False for each choice.)

A. True / False The X server is an example of a trusted intermediary.

B. True / False The X server notifies the client when regions of the client's window become visible, but not when regions of the client's window become obscured.

C. True / False The X server runs in user mode.

D. True / False The X client sends RPCs to the X server to check if a mouse click has occurred.

3. [8 points]: Which of the following statements about the Lockset algorithm as used in the RaceTrack paper (Reading #7 "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking" by Yu, Rodeheffer, and Chen, Proc. of the 20th ACM Symposium on Operating Systems Principles, 2005) is true?

(Circle True or False for each choice.)

A. True / False It can be used to detect deadlocks in multi-threaded programs.

B. True / False It can report false race conditions that are not actually present in the code.

C. True / False It can fail to detect race conditions that are actually present in the code.

D. True / False It cannot detect race conditions involving three or more threads.

Name:

4. [8 points]: Louis writes a multithreaded program, which produces an incorrect answer some of the time, but always completes. He suspects a race condition. Which of the following are strategies that can reduce or eliminate race conditions in Louis's program?

(Circle True or False for each choice.)

A. True / False Separate a multi-threaded program into multiple single-threaded programs (each with its own address space) and share data between them via an inter-program communication primitive like pipes.

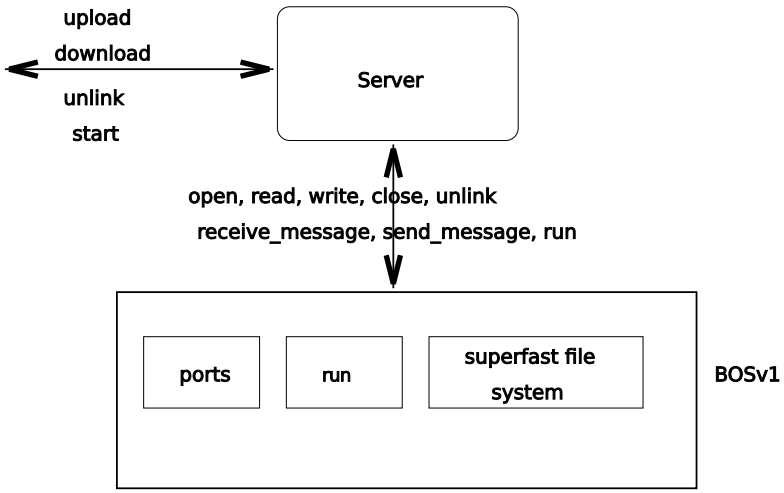
B. True / False Apply the one-writer rule.

C. True / False Ensure that for each shared variable v, it is protected by some lock l_v.

D. True / False Ensure that all locks are acquired in the same order.

II Ben's OS (BOS)

Ben is having a blast with design project 1. To get a better feeling for the workloads that his superfast file system might experience, he sketches out a server:



The server supports the following requests:

- UPLOAD: upload a file to the server. Attempting to write an existing file results in an error.
- DOWNLOAD: download a file from the server. Attempting to read a file that doesn't exist results in an error.
- UNLINK: remove a file. Attempting to unlink a file that doesn't exist results in an error.
- START: start a new program. This request is not required by DP1 but Ben added it to make it possible to start programs on the server. This request may fail if there are not enough resources to start the program.

To support the server, Ben's operating system (BOS) supports the following supervisor calls (also sometimes called system calls) in addition to the file system calls OPEN, WRITE, READ, CLOSE, and UNLINK:

- RECEIVE_MESSAGE(*port*): A program calling RECEIVE_MESSAGE will block until a message destined for *port* arrives on this machine.
- SEND_MESSAGE(*message*): The procedure SEND_MESSAGE sends a message to port *dest_port* on machine *destination* (see message structure in figure 1).
- RUN(*name*): Applications can start a new program using RUN. RUN creates a new user-level address space, loads the program specified in its argument into the address space, creates a thread to run the program, and returns to the caller. The new program may call any of the supervisor calls.

Ben's names this first BOS implementation BOSv1.
Name:

The server runs like any other application (i.e., it has been created using RUN) and is implemented as shown in figure 1. (We suggest you skim the code and continue reading the text of the quiz. The implementation of the server is straightforward and it doesn't include any quiz traps. For specific questions you may want to go back to the code to firm up your understanding of what the specific question is asking.)

5. [5 points]: Looking at the underlined strings in figure 1, which of the following are examples of names?

(Circle True or False for each choice.)

A. True / False "*source*"
B. True / False "1048576"
C. True / False "*request*"
D. True / False "*SERVER_PORT*"
E. True / False "UNLINK"

To handle failures, the RPC stub on the client resends a request if it doesn't receive a reply within a certain period of time. On receiving a reply for the request, the stub returns.

6. [8 points]: Assume a single client. Which of the following requests are idempotent (i.e., the request can be repeated and will always produce the same result as if the request completed once)?

(Circle True or False for each choice.)

A. True / False UPLOAD
B. True / False DOWNLOAD
C. True / False UNLINK
D. True / False START

Name:

```
structure message {
 address destination;     // destination address
 int dest_port;     // destination port
 address source;     // source address
 int src_port;     // source port
 int opcode;     // operation code of request
 int result;     // result of request
 char name[MAXNAMELEN];     // name of file, no more than MAXNAMELEN charaters
 int len;     // length of data
 char data[1048576];     // data of message, up to 1 Megabyte of characters
}

procedure SERVER()
  structure message request, reply;
  int fd;
  while TRUE do
   request ← RECEIVE_MESSAGE(SERVER_PORT);      // Wait for a message sent to port SERVER_PORT
   if request.opcode = UPLOAD then     // upload request?
    fd ← OPEN(request.name, O_EXCL|O_CREATE|O_WRONLY);     // Writing an existing file is an error
    if fd < 0 then reply.result ← fd;     // error opening the file?
    else {
     reply.result ← WRITE(fd, request.data, request.len);
     CLOSE(fd);
    }
   else if request.opcode = DOWNLOAD then     // download request?
    fd ← OPEN(name, READ_ONLY);     // Attempt to open the file for reading
    if fd < 0 then reply.result ← fd;     // error opening the file?
    else {
     reply.len ← request.len;
     reply.result ← READ(fd, reply.data, reply.len);
     CLOSE(fd);
    }
   else if request.opcode = UNLINK then     // unlink request?
     reply.result ← UNLINK(request.name);
   else if request.opcode = START then     // start a program?
     reply.result ← RUN(request.name);
   else {    // reply with an error
     reply.result ← ERROR_OPCODE;
   }
   reply.destination ← request.source;
   reply.dest_port ← request.src_port;
   reply.source ← MYMACHINE;
   reply.src_port ← SERVER_PORT;
   reply.opcode ← request.opcode;
   SEND_MESSAGE(reply);
```

Figure 1: Ben's server. (Some strings are underlined for question 5.)

Name:

7. [9 points]: A single client uses the server. The client sends an RPC to the server to upload a file and then sends another RPC to unlink the file. The client repeats this sequence many times. Occasionally the client observes that the reply from the server for the unlink RPC contains an error, indicating that the file didn't exist. Which of the following faults could, by itself, caused the observed behavior? (Remember that the client retries each request until it receives a reply.)

(Circle True or False for each choice.)

A. True / False The server failed after the server processed an earlier unlink request but before sending a reply, and then restarted.

B. True / False The network between the client and the server lost a reply.

C. True / False The network between the client and the server lost a request.

D. True / False The server is so slow that the client, for a given unlink RPC, resends the request and then receives the reply for the first request for that RPC.

Ben measures the performance of the server on BOSv1 when it runs many programs concurrently, and is disappointed with the measured performance. Ben modifies RUN to make the system faster. The new version of RUN loads the program in the kernel address space and creates a thread to run the program in the kernel address space. Thus, all threads run in kernel mode in a single address space. The threads are scheduled preemptively. Ben names this version BOSv2.

8. [8 points]: What program errors can BOSv1 (where each program runs in its own user-level address space) isolate well and BOSv2 not?

(Circle True or False for each choice.)

A. True / False Writes to arbitrary addresses

B. True / False Reads from arbitrary addresses

C. True / False Jumps to arbitrary addresses

D. True / False Infinite loops

9. [8 points]: Which overheads can BOSv2 avoid (compared to BOSv1)?

(Circle True or False for each choice.)

A. True / False The performance overhead of entering and leaving the kernel.

B. True / False The performance overhead of switching the page-map address register.

C. True / False The memory overhead of allocating a stack per thread.

D. True / False The performance overhead of loading PC and SP when switching threads.

10. [8 points]: Programs in BOSv1 assume they run in their own virtual address space. In BOSv2 the programs and the kernel share a single virtual address space. Ben doesn't want to recompile or inspect (and perhaps rewrite) all BOSv1 programs. Which of the following properties of a BOSv1 program would allow Ben to start the program in BOSv2 (using RUN) without having to recompile or rewrite the program?

(Circle True or False for each choice.)

A. True / False All addresses of the program are PC relative.

B. True / False Global data structures in the program are addressed using absolute addresses.

C. True / False The program uses multiple threads.

D. True / False Procedures in the program are addressed using absolute addresses.

Ben just learned about semaphores, a coordination primitive similar to eventcounts, but different. Semaphores support the following two operations:

- DOWN (**semaphore** *sem*): decrement if *sem* > 0 and return; otherwise, wait until another thread increases *sem* and then try to decrement again.
- UP (**semaphore** *sem*): increment *sem*, wake up all threads waiting on *sem*, and return.

For completeness, figure 2 lists the pseudocode, which works in the same style as the implementation of eventscounts in the class notes (see section E.3 of chapter 5). ACQUIRE uses a spin lock and turns off interrupts. RELEASE releases the lock and enables interrupts.

For all questions you can assume that the thread manager implements the procedures UP and DOWN correctly; that is, you can just skim the code—there are no quiz traps. In particular, the thread manager correctly guarantees that UP and DOWN are atomic with respect to concurrent invocations by threads and interrupt handlers.

```
shared lock threadtable_lock;     // the global lock for the thread manager
procedure UP(semaphore sem)
  ACQUIRE(threadtable_lock);
  sem ← sem + 1;
  WAKEUP(sem);     // set the state of all threads that are waiting on sem to RUNNABLE
  RELEASE(threadtable_lock);


procedure DOWN(semaphore sem)
  ACQUIRE(threadtable_lock);
  while sem < 1  do {    // A
   SETWAITING(sem);     // B; set this thread's state to WAITING and record that it is waiting on sem
   RELEASE(threadtable_lock);
   YIELD(CONTINUE);     // calling thread releases the processor
   ACQUIRE(threadtable_lock);
  }
  sem ← sem − 1;
  RELEASE(threadtable_lock);
```

Figure 2: Implementation of semaphores. WAKEUP, SETWAITING, and YIELD are procedures implemented by the thread manager. WAKEUP sets the state of all threads that are waiting on semaphore *sem* to RUNNABLE. SETWAITING sets the state of the calling thread to WAITING and records the semaphore the thread is waiting on.

Name:

Using DOWN and UP, Ben implements a bounded buffer for each port as follows:

```
structure port_info {
 semaphore n ← 0;
 structure message buffer[NMSG];     // an array of NMSG messages
 long integer in ← 0;
 long integer out ← 0;
} port_infos[NPORT];     // an array of port_info's

procedure INTERRUPT(structure message m)
     // an interrupt announcing the arrival of message m
   structure port_info d;     // a local reference to a port_info structure
   d ← port_infos[m.dest_port];
   if d.in − d.out ≥ NMSG  then {     // is there space in the buffer?
    return;     // No, return; i.e., throw message away.
   }
   d.buffer[d.in mod NMSG] ← m;
   d.in ← d.in + 1;
   UP(d.n);


procedure RECEIVE_MESSAGE(dest_port)
   structure port_info d;     // a local reference to a port_info structure
   d ← port_infos[dest_port];
   DOWN(d.n);
   m ← d.buffer[d.out mod NMSG];
   d.out ← d.out + 1;
   return m;
```

The BOS implementation maintains an array of *port_info*s. Each *port_info* contains a bounded buffer. When a message arrives from the network, it generates an interrupt, and the network interrupt handler (INTERRUPT) puts the message in the bounded buffer of the port specified in the message. If there is no space in that bounded buffer, the interrupt handler throws the message away. A thread (e.g., Ben's server) consumes a message by calling RECEIVE_MESSAGE, which removes a message from the bounded buffer of the port it is receiving from.

To coordinate the interrupt handler and a thread calling RECEIVE_MESSAGE, the BOS implementation uses a semaphore. For each port, BOS keeps a semaphore *n* that counts the number of messages in the port's bounded buffer. If *n* reaches 0, the thread calling DOWN in RECEIVE_MESSAGE will enter the WAITING state. When INTERRUPT adds a message to the buffer, it calls UP on n, which will wake up the thread (i.e., set the thread's state to RUNNABLE).

Name:

11. [16 points]: Assume that there are no concurrent invocations of INTERRUPT, and that there are no concurrent invocations of RECEIVE_MESSAGE on the same port. Which of the following statements is true about the implementation of INTERRUPT and RECEIVE_MESSAGE?

(Circle True or False for each choice.)

A. True / False There are no race conditions between two threads that invoke RECEIVE_MESSAGE concurrently on different ports.

B. True / False The complete execution of UP in INTERRUPT will not be interleaved between the statements labeled A and B in DOWN.

C. True / False Because DOWN and UP are atomic, the processor instructions necessary for subtracting of *sem* in DOWN and adding to *sem* in UP won't be interleaved incorrectly.

D. True / False Because *in* and *out* may be shared between the interrupt handler running INTERRUPT and a thread calling RECEIVE_MESSAGE on the same port, it is possible for INTERRUPT to throw away a message even though there is space in the bounded buffer.

Alyssa claims that semaphores can also be used to make operations atomic. She proposes the following modification to a *port_info* structure and RECEIVE_MESSAGE to allow threads to concurrently invoke RECEIVE_MESSAGE on the same port without race conditions (only the commented lines changed):

```
structure port_info {
 semaphore n ← 0;
 semaphore mutex ←????;     // see question below
 message buffer[NMSG];
 long integer in ← 0;
 long integer out ← 0;
} port_infos[NPORT];

procedure RECEIVE_MESSAGE(dest_port)
   structure port_info d;
   d ← port_infos[dest_port];
   DOWN(d.mutex);     // enter atomic section
   DOWN(d.n);
   m ← d.buffer[d.out mod NMSG];
   d.out ← d.out + 1;
   UP(d.mutex);     // leave atomic section
   return m;
```

12. [8 points]: To what value can *mutex* be initialized to avoid race conditions and deadlocks when multiple threads call RECEIVE_MESSAGE on the same port?

(Circle True or False for each choice.)

A. True / False 0

B. True / False 1

C. True / False 2

D. True / False -1

End of Quiz I

Name: