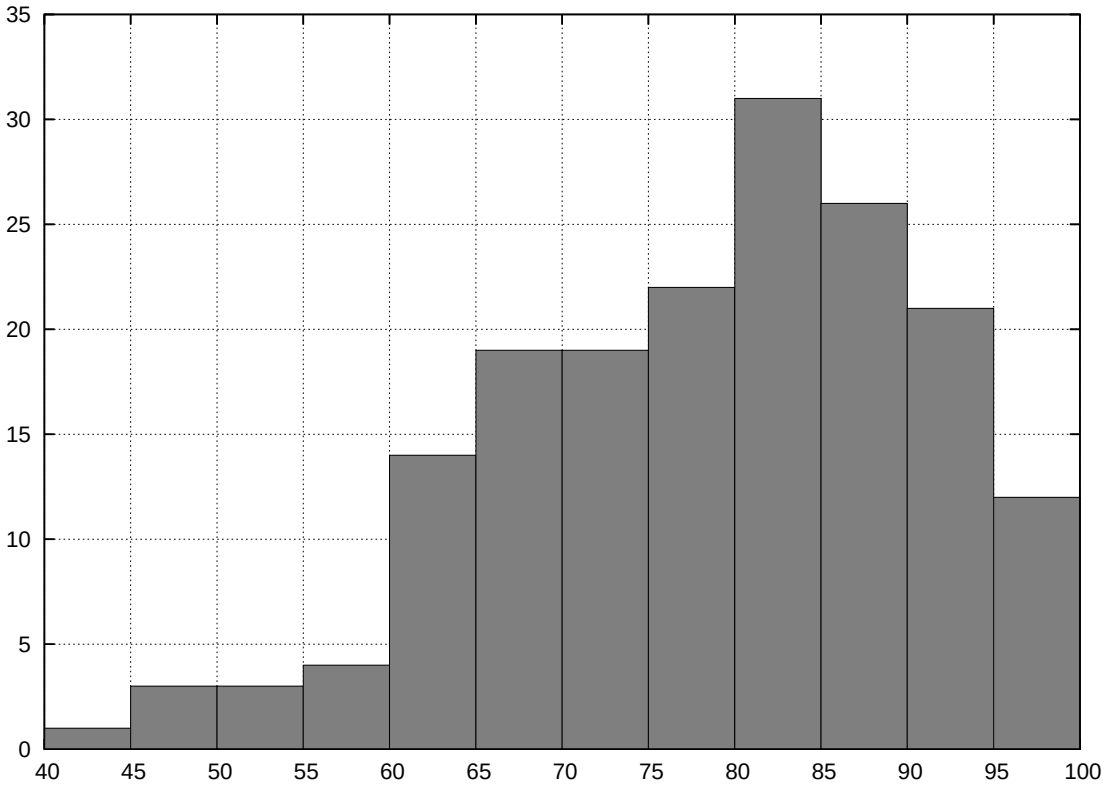
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2006

Quiz I



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.
FALSE. *The i-number of a file is simply an offset into the i-list.*

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

C. True / False Links may be made to directories.
FALSE. *The Unix paper explicitly says links cannot be made to directories.*

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

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

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

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.
TRUE. *The X server acts as an intermediary between different client programs all sharing the display. Each clients trusts that the server will display its content properly and prevent other clients from interfering with its windows.*

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.
TRUE. *See Section 7 of the X Windows paper on Exposures.*

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

D. True / False The X client sends RPCs to the X server to check if a mouse click has occurred.
FALSE. *The X server sends event to the clients via a stream; clients are not required to request them from the server.*

Name:

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.

FALSE. *The Lockset algorithm cannot detect deadlocks.*

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

TRUE. *Racetrack flags any case where two concurrent threads both access a shared variable without consistently holding some lock as a race condition. There may be situations (such as lock-free code) that contain no races but don't require this strict locking discipline to be followed. The experimental results demonstrate clearly that a number of non-race conditions are detected by the algorithm.*

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

TRUE. *Racetrack only flags race conditions that it actually sees in a trace of some execution of the program; different executions may reveal other races.*

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

FALSE. *Racetrack works just fine with more than two threads (as in the example in Figure 8 in the RaceTrack paper).*

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.

TRUE. *Although this approach may slow down the system, it will ensure there are no races. (The threads would no longer share memory, and a race condition can only happen when threads actually share memory.)*

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

TRUE. *The one-writer rule is a good strategy for helping to avoid race conditions.*

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

TRUE. *Ensuring that a shared variable is always protected by a lock will guarantee there is no race on that variable.*

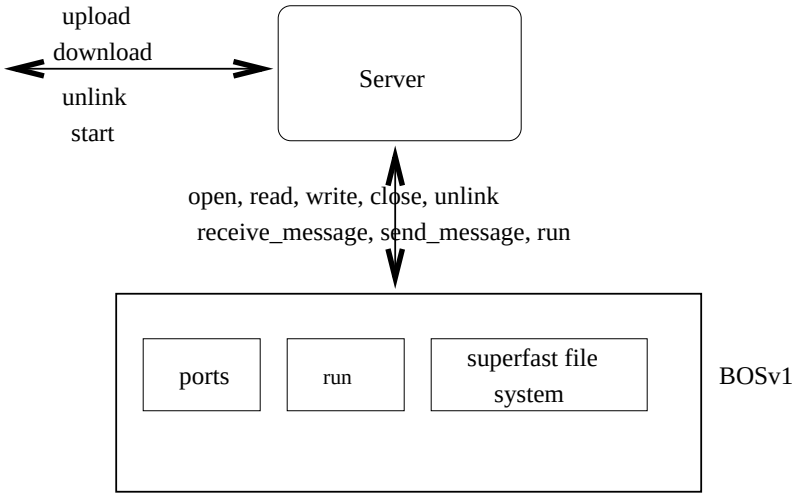
D. True / False Ensure that all locks are acquired in the same order.

FALSE. *Always acquiring locks in the same order will prevent deadlocks. While deadlocks are arguably a form of race condition, there was no deadlock here since Louis's program always completes.*

Name:

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*"
TRUE.

B. True / False "1048576"
FALSE. *This is a number. All of the others are names and would require a name resolution algorithm to resolve to a value.*

C. True / False "*request*"
TRUE.

D. True / False "*SERVER_PORT*"
TRUE.

E. True / False "UNLINK"
TRUE.

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
FALSE. *After the first execution of UPLOAD, subsequent calls will cause an error response, because UPLOAD requires that the file not yet exist.*

B. True / False DOWNLOAD
TRUE.

C. True / False UNLINK
FALSE. *Calls of UNLINK (after the first call) will cause an error because the file will no longer exist.*

D. True / False START
FALSE. *Each call of START causes a new program to start running with its own thread. Having multiple copies of a program running is not the same as having just one copy. For example, the second might be unable to make progress because the first one has tied up too many resources.*

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.

TRUE. *If the server fails and then restarts, its reply to the second UNLINK request could be an error reply (assuming it did not lose the effect of the previous execution of UNLINK in the failure).*

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

TRUE. *The second reply of the server will reflect the second execution of UNLINK, which will produce an error reply since at that point the file no longer exists.*

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

FALSE. *Since the first request message was lost, the server has not acted on that request, and therefore the file still exists when the second request message arrives.*

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.

FALSE. *The reply to the first request will indicate a successful completion of the UNLINK request, even though it arrives after the second request has been sent, because that reply reflects the effect of the first execution of the UNLINK request.*

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

TRUE. *In BOSv2 there is no protection of the address space from program errors and therefore arbitrary reads, writes, and jumps can occur; none of these is possible in BOSv1. Even arbitrary reads can be problematic because the read might access private information or be used to obtain an address that a later instruction might jump or write to.*

B. True / False Reads from arbitrary addresses

TRUE.

C. True / False Jumps to arbitrary addresses

TRUE.

D. True / False Infinite loops

FALSE. *Since both BOSv1 and BOSv2 use preemptive scheduling, program errors that result in infinite loops are handled identically in each of them.*

Name:

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.

TRUE. *In BOSv2 a switch from one thread to another can be done without entering and leaving kernel mode since all threads are already running in kernel mode.*

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

TRUE. *In BOSv2 there is a single page map used by all threads.*

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

FALSE. *Threads require their own stack in either BOSv1 or BOSv2.*

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

FALSE. *Switching threads always requires loading the PC and SP.*

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.

TRUE. *If all addresses of the program are PC-relative, it won't matter where in the address space the code and the data it uses reside.*

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

FALSE. *If the program uses absolute addresses either for global data or for naming procedures (question 10D), these addresses would need to be fixed up (relocated) to account for the actual locations of the referred to items.*

C. True / False The program uses multiple threads.

FALSE. *The use of multiple threads is irrelevant to the ease of getting a program to run in BOSv2.*

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

FALSE.

Name:

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.

TRUE. *Two threads that invoke* RECEIVE_MESSAGE *concurrently on different ports will access different elements of port_infos. They therefore access disjoint regions of the memory, and there are no data races.*

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

TRUE. *DOWN holds threadtable_lock between A and B. The first statement of UP acquires threadtable_lock. If one thread invokes UP when another thread is between A and B in DOWN, the thread that invoked UP will wait until after the thread in DOWN executes B and proceeds on to releases threadtable_lock.*

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.

TRUE. *If the instructions for changing sem in UP and DOWN were interleaved at all, UP and DOWN would not be atomic.*

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.

TRUE. *Consider the following execution sequence:*

1. INTERRUPT is called enough times to fill up $d.buffer$ completely.

2. RECEIVE_MESSAGE executes until just before the statement $d.out \leftarrow d.out + 1$; The thread running RECEIVE_MESSAGE is now preempted and does not run until later.

3. INTERRUPT is called. At this point INTERRUPT will throw the message away, even though it could safely stick the message in the buffer.

Also note that $d.int$ and $d.out$ are both long integers. Changes to long integers are not atomic — they may require multiple instructions to appropriately update the upper and lower halves of the long value stored in the integer.

Consider the following sequence:

1. INTERRUPT and RECEIVE_MESSAGE are called (perfectly interleaved, with INTERRUPT called first in each pair) $(2^{32}) - 1$ times. At this point there are no messages in the buffer and $d.in = d.out =$`0x00000000ffffffff`

2. INTERRUPT is called and inserts a message into the buffer. $d.in =$`0x0000000100000000`

3. RECEIVE_MESSAGE is called. It takes the message out of the buffer and proceeds on to the statement $d.out \leftarrow d.out+1$. It executes the first half of the update and $d.out =$`0x0000000000000000`. The thread running RECEIVE_MESSAGE is now preempted and does not run until later. Note that the second half of the update will eventually complete the increment of $d.out$ and set it to `0x0000000100000000`, but this will not occur until the thread runs later.

4. INTERRUPT is called. It computes that $d.in - d.out \geq NMSG$, and so discards the message even though it could safely stick the message in the buffer.

Name:

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

FALSE. *If mutex is initialized to 0, the call to DOWN(d.mutex) will always block and the program will deadlock.*

B. True / False 1

TRUE. *The correct initial value is 1. Only 1 thread at a time will proceed past the call DOWN(d.mutex), and atomicity is preserved.*

C. True / False 2

FALSE. *Two threads may proceed concurrently past the call DOWN(d.mutex). The statements of these two threads could interleave in such a way as to cause a race condition; for example, both could return the same message.*

D. True / False -1

FALSE. *The call to DOWN(d.mutex) will always block and the program will deadlock.*

End of Quiz I

Name: