



Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2005

Quiz I

There are 13 questions and 9 pages in this quiz booklet. To receive credit for a question, answer it according to the instructions given. You have **50 minutes** to answer the questions.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be harder than others. Attack them in the order that allows you to make the most progress. 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!

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

CIRCLE your recitation section number:

- 10:00** 1. Joseph/Jamieson
11:00 3. Joseph/Winstein 5. Rinard/Hull
12:00 13. Rinard/Winstein 6. Lesniewski-Laas/Ferreira
1:00 7. Liskov/Ferreira 9. Katabi/Jamieson
2:00 10. Liskov/Hull 11. Katabi/Rost

Do not write in the boxes below

1-4 (xx/32)	5-7 (xx/24)	8-9 (xx/18)	10-12 (xx/18)	13 (xx/8)	Total (xx/100)

Name:

I Complex Systems

1. **[8 points]:** Which of the following is usually observed in a complex computer system?
(Circle ALL that apply)
 - A. The underlying technology has a high rate of change.
 - B. It is not easy to write a succinct description of the behavior of the system.
 - C. It has a large number of interacting features.
 - D. It exhibits emergent properties that often make the system perform better than envisioned by the system's designers.

2. **[8 points]:** According to Leveson and Turner's paper, the Therac-25 incidents:
(Circle ALL that apply)
 - A. Were caused by errors that were dependent on timing.
 - B. Were caused by human operators entering incorrect commands.
 - C. Were caused by a software bug in which a variable overflowed.
 - D. Could have been avoided by a hardware mechanism to prevent the beam from turning on when the turntable wasn't in the correct position.

3. **[8 points]:** A newly created process in UNIX (reading #XXX)
(Circle ALL that apply)
 - A. May be created only using the *fork* system call.
 - B. Can inherit file descriptors from its parent to facilitate inter-process communication.
 - C. Has a different address space from its parent.
 - D. Must have a shell as an ancestor process.

4. **[8 points]:** The inode of a file plays an important role in the UNIX file system. Which of these statements is true of the inode data structure?
(Circle ALL that apply)
 - A. The inode of a file contains a reference count.
 - B. The reference count of the inode of a directory cannot be larger than 1.
 - C. The inode of a directory contains the inodes of the files in the directory.
 - D. The inode of a symbolic link contains the path name of the target of the link.

Name:

5. [8 points]: Which of the following statements is true about the GNU/Linux linker, ld, as described in Lecture #4.

(Circle ALL that apply)

- A. ld resolves symbols by searching through a set of contexts, where each context is a “.o” or library file. Within each context, symbol resolution is done using a table lookup.
- B. ld resolves symbols by searching through a set of contexts, where each context is a “.o” or library file. Within each context, symbol resolution is done using pathname resolution.
- C. ld’s output when operating on a set of files depends on the order in which the different “.o” and library files are specified on the command line.
- D. ld may produce an executable in which certain symbols are defined more than once.

6. [8 points]: Which of the following statements is true of the X Window system (Reading #6)?

(Circle ALL that apply)

- A. The X server is a trusted intermediary and enforces modularity between X clients in their use of the display resource.
- B. An X client uses synchronous remote procedure calls to communicate with the X server.
- C. The X server handles all display device dependencies.
- D. To obtain high performance, the X server is implemented in the kernel of the operating system of the machine on which it runs.

7. [8 points]: Which of the following mechanisms does MapReduce use to increase the performance of the computation?

(Circle ALL that apply)

- A. It executes different “map” processes in parallel.
- B. It places “reduce” computations on machines that have the data that the computation will access locally available.
- C. It writes the output of each reduce computation to disks on different machines.
- D. It may execute the same reduce computation at the same time on different machines and use the first results that are returned.

II EZ-Park

Finding a parking spot in Cambridge is quite a nightmare, and finding one at MIT is as hard as it gets. Ben Bitdiddle decides that a little technology can help, and sets about designing the EZ-Park client/server system. He gets a machine to run an EZ-Park server in his dorm room. He manages to convince MIT parking to equip each car with a tiny computer running EZ-Park client software. EZ-Park clients communicate with the server using remote procedure calls (RPCs). A client makes requests to Ben's server both to find an available spot (when the car's driver is looking for one) and to relinquish a spot (when the car's driver is leaving a spot). A car driver uses a parking spot if, and only if, EZ-Park allocates it to him or her.

In Ben's initial design, the server software runs in one address space and spawns a new thread for each client request. The server has two procedures: `FIND_SPOT()` and `RELINQUISH_SPOT()`. Each of these threads is spawned in response to the corresponding RPC request sent by a client.

The server threads use a shared array, `available[]`, of size `NSPOTS` (the total number of parking spots). `available[j]` is set to `TRUE` if spot j is free, and `FALSE` otherwise; it is initialized to `FALSE`, and there are no cars parked to begin with. The `NSPOTS` parking spots are numbered from 0 through `NSPOTS-1`. `numcars` is a global variable that counts the total number of cars parked; it is initialized to 0.

Ben implements the following pseudocode (line numbers are shown at the left of each line). Each `FIND_SPOT()` thread is in a **while** loop that terminates only when the car is allotted a spot.

```

int procedure FIND_SPOT()
1. while (TRUE){
2.   for (int  $i \leftarrow 0; i < NSPOTS; i \leftarrow i + 1$ ){
3.     if (available[i] == TRUE) then
4.       available[i] ← FALSE; numcars ← numcars + 1;
5.       return ( $i$ ); // Allocate spot  $i$  for client to use
6.     }
7.   }

procedure RELINQUISH_SPOT(int  $spot$ )
1. available[spot] ← TRUE;
2. numcars ← numcars - 1;

```

Ben's intended correct behavior for his server (the "correctness specification") is as follows:

- A. `FIND_SPOT()` allocates any given spot in $[0, \dots, NSPOTS - 1]$ to at most one car at a time, even when cars are concurrently sending requests to the server requesting spots.
- B. `numcars` must correctly maintain the number of parked cars.
- C. If spots are available and exactly one client makes a `FIND_SPOT()` request, the client should get a spot even if no parked cars relinquish spots.

Ben runs the server and finds that when there are no concurrent requests, EZ-Park works correctly. However, when he deploys the system, he finds that sometimes multiple cars are assigned the same spot, leading to collisions! His system does not meet the correctness specification when concurrent requests are made.

Name:

Make the following assumptions:

- A. The statements to update *numcars* are *not atomic*; they each take multiple instructions to run.
- B. Cars arrive and leave at random.
- C. The server runs on a single processor with a preemptive thread scheduler.
- D. The network delivers RPC messages reliably, and there are no network, server, or client failures.

8. [8 points]: Which of these statements is true about the problems with Ben's design?

(Circle ALL that apply)

- A. There is a race condition in accesses to *available*[], which may be triggered when more than one FIND_SPOT() thread runs.
- B. There is a race condition in accesses to *available*[], which may be triggered when one FIND_SPOT() thread and one RELINQUISH_SPOT() thread runs.
- C. There is a race condition in accesses to *numcars*, which may be triggered when more than one thread updates *numcars*.
- D. There is no race condition as long as the average time between client requests to find a spot is larger than the average processing delay for a request.

9. [10 points]: Ben enlists Alyssa's help to fix the problem with his server, and she tells him that he needs to set some locks. Which of these modifications correctly fixes the problem and meets the correctness specification? (An arrow is shown to the left of the newly added lines.)

(Circle ALL that apply)

- A. Method #1: Use ACQUIRE() and RELEASE() as shown below.

```

int procedure FIND_SPOT()
  1. while (TRUE){
→ 1'. ACQUIRE(avail_lock);
  2. for (int i ← 0; i < NSPOTS; i ← i + 1){
  3. if (available[i] == TRUE) then
  4.   available[i] ← FALSE; numcars ← numcars + 1;
→ 4'.  RELEASE(avail_lock);
  5.   return (i); // Allocate spot i for client to use
  6. }
→ 6'.  RELEASE(avail_lock);
  7. }

```

```

procedure RELINQUISH_SPOT(int spot)
  // Called when a client car is leaving spot
→ 0'. ACQUIRE(avail_lock);
  1. available[spot] ← TRUE;
  2. numcars ← numcars - 1;
→ 2'. RELEASE(avail_lock);

```

Name:

B. Method #2: Use ACQUIRE() and RELEASE() as shown below.

```

int procedure FIND_SPOT()
1. while (TRUE){
2.   for (int  $i \leftarrow 0; i < \text{NSPOTS}; i \leftarrow i + 1$ ){
→ 2'. ACQUIRE(avail_lock);
3.   if (available[ $i$ ] == TRUE) then
4.     available[ $i$ ] ← FALSE; numcars ← numcars + 1;
→ 4'. RELEASE(avail_lock);
5.     return ( $i$ ); // Allocate spot  $i$  for client to use
6.   }
7. }

procedure RELINQUISH_SPOT(int spot)
→ 0'. ACQUIRE(avail_lock);
1. available[ $spot$ ] ← TRUE;
2. numcars ← numcars - 1;
→ 2'. RELEASE(avail_lock);

```

C. Method #3: Use ACQUIRE() and RELEASE() as shown below.

```

int procedure FIND_SPOT()
1. while (TRUE){
2.   for (int  $i \leftarrow 0; i < \text{NSPOTS}; i \leftarrow i + 1$ ){
→ 2'. ACQUIRE(avail_lock);
3.   if (available[ $i$ ] == TRUE) then
4.     available[ $i$ ] ← FALSE; numcars ← numcars + 1;
→ 4'. RELEASE(avail_lock);
5.     return ( $i$ ); // Allocate spot  $i$  for client to use
→ 5'. else RELEASE(avail_lock);
6.   }
7. }

procedure RELINQUISH_SPOT(int spot)
→ 0'. ACQUIRE(avail_lock);
1. available[ $spot$ ] ← TRUE;
2. numcars ← numcars - 1;
→ 2'. RELEASE(avail_lock);

```

D. Method #4: Use ACQUIRE() and RELEASE() as shown below.

```

int procedure FIND_SPOT()
1. while (TRUE){
→ 1'. ACQUIRE(avail_lock);
2. for (int i ← 0; i < NSPOTS; i ← i + 1){
3. if (available[i] == TRUE) then
4.   available[i] ← FALSE; numcars ← numcars + 1;
5.   return (i); // Allocate spot i for client to use
6. }
7. }

procedure RELINQUISH_SPOT(int spot)
1. available[spot] ← TRUE;
2. numcars ← numcars - 1;
→ 2'. RELEASE(avail_lock);

```

E. None of the above.

For the rest of the questions, assume that Ben has managed to implement a lock-based EZ-Park system that has no race conditions. Ben now decides to combat parking at a truly crowded location... Gillette Stadium! He updates NSPOTS and deploys the system during the first home game of the Patriots' season. He finds that many clients complain that his server is slow or unresponsive.

10. [4 points]: If a client invokes the FIND_SPOT() RPC when the parking lot is full, how quickly will it get a response?

- A.** The client will not get a response until at least one car relinquishes a spot.
- B.** The client may never get a response even when other cars relinquish their spots.

Alyssa tells Ben to add a client-side timeout to his RPC system if the server does not respond within 4 seconds. Upon a timeout, the car's driver may retry the request, or leave the stadium to watch the game on TV.

11. [6 points]: When Ben adds the timeout code to his client, he finds some surprises. Which of the following statements is true of Ben's implementation?

(Circle ALL that apply)

- A. The server may be running multiple active threads on behalf of the same client car at any given time.
- B. The server may assign the same spot to two cars making requests.
- C. *numcars* may be smaller than the actual number of cars parked in the parking lot.
- D. *numcars* may be larger than the actual number of cars parked in the parking lot.

12. [8 points]: Alyssa thinks that the operating system running Ben's server may be spending a fair amount of time switching between threads when many RPC requests are being processed concurrently. Which of these statements about the work required to perform the switch is correct? Notation: PC = program counter; SP = stack pointer; PMAR = page map address register. Assume that the operating system behaves according to the description in chapter 5 and lecture.

(Circle ALL that apply)

- A. On any thread switch, the operating system has to save the values of the PMAR, PC, SP, and several registers.
- B. On any thread switch, the operating system has to save the values of the PC, SP, and several registers.
- C. On any thread switch between two RELINQUISH_SPOT() threads, the operating system has to save *only* the value of the PC, since RELINQUISH_SPOT() has no return value.
- D. The number of instructions required to switch from one thread to another is proportional to the number of bytes currently on the thread's stack.

Ben finds that the CPU utilization of his server is often close to 100%, and he wants to reduce it. Alyssa studies the server source code carefully and finds that a lot of CPU time is being spent in the ACQUIRE() function to obtain a lock. Ben's current implementation of ACQUIRE() and RELEASE() uses the following spin-lock code. The current pseudocode is similar to the one in chapter 5 (page 5-56) and to the scheme presented in lecture:

```

procedure ACQUIRE(L) {
    RSL L, R0;
    while (R0 == TRUE) {
        RSL L, R0;
    }
}

```

```

procedure RELEASE(L) {
    L = FALSE;
}

```

13. [8 points]: Modify the implementation of ACQUIRE(*lock*) and RELEASE(*lock*) to use WAIT and NOTIFY. To help you, we have given some scaffolding code, and you have to fill in missing lines of pseudocode in the spaces provided. Not all spaces need to be filled. A correct solution does not need more than three or four lines to be written at the right places.

```

// released and num_acquired are shared by all threads
eventcount released = 1;
int num_acquired = 0; // incremented on each successful acquire

procedure ACQUIRE(L) {
    RSL L, R0;

    while (R0 == TRUE) {

        RSL L, R0;

    }

    num_acquired = num_acquired + 1;
} // end of ACQUIRE()

procedure RELEASE(L) {
    L <-- FALSE;

} // end of RELEASE()

```

End of Quiz I!