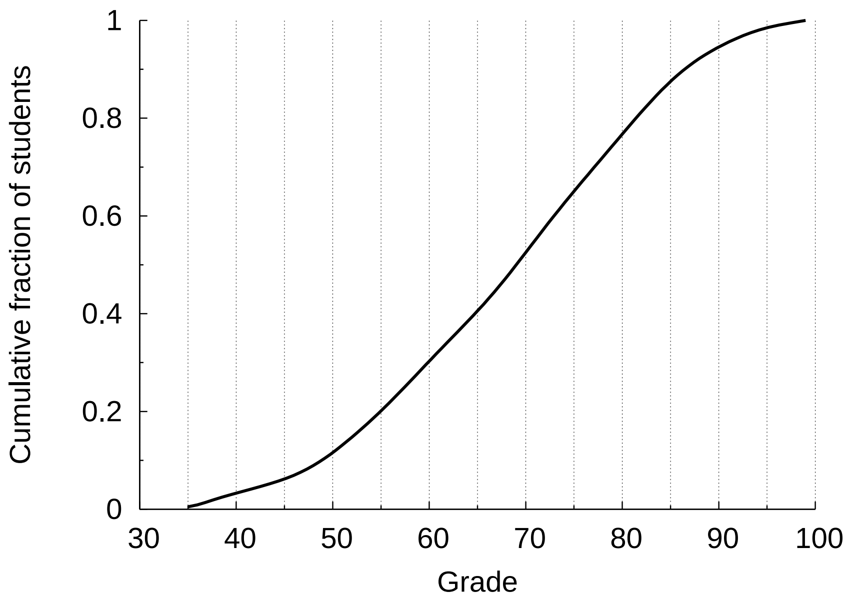
Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2005

Quiz I Solutions



Average: 68.7
Median: 69
Standard deviation: 14.2

Score range	# of students (only those who took quiz on 3/4/2005)
30–39	7
40-49	13
50-59	38
60-69	44
70-79	46
80-89	40
90-99	13

Name: <u>Solutions</u>

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.
YES. *Digital hardware, which forms the basis for today's computing, improves dramatically with time.*

B. It has a large number of interacting features.
YES. *Interacting requirements and features are observed in complex systems.*

C. It is easy to write a succinct description of the behavior of the system.
NO. *A complex system is hard to describe easily, and usually no single person understands it all.*

D. It exhibits emergent properties.
YES. *Almost all complex systems exhibit emergent properties, and complex computer systems are no exception.*

2. [8 points]: According to Leveson and Turner's paper (reading #4), some of the Therac-25 incidents:
(Circle ALL that apply)

A. Were caused by errors that were dependent on timing.
YES. *Timing-dependent errors occurred in all the incidents.*

B. Were caused by human operators who were not trained to use the machines.
NO. *The authors in fact suggest that the operators were trained and quite experienced in using the machines.*

C. Were caused by a software bug in which a variable overflowed.
YES. *An overflow bug caused at least one of the incidents.*

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.
YES. *A hardware interlock as in the Therac-20 would have prevented some of the incidents.*

3. [8 points]: A process in UNIX (as descibed in reading #5)
(Circle ALL that apply)

A. May be created only using the *fork* system call, except for the *init* process.
YES.

B. Can inherit file descriptors from its parent to facilitate inter-process communication.
YES.

C. Has a different address space from its parent.

YES.

D. Must have a shell as an ancestor process.

NO. *For example, it could have* init *as its parent 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, as described in chapter 3, appendix A?

(Circle ALL that apply)

A. The inode of a file contains a reference count.

YES. *The reference count is used to implement (hard) links and determine when to reclaim the inode and file blocks when* unlink *operations are done.*

B. The reference count of the inode of a directory should not be larger than 1.

YES. *The reference count of a file's inode increments whenever a hard link is made to the file. In UNIX, hard links to a directory are not allowed, because it complicates garbage collection. Hence, the reference count of the inode of a directory should not ever exceed 1.*

C. The inode of a directory contains the inodes of the files in the directory.

NO. *The directory itself is a file, and its blocks contain the names of the files in the directory. The inode of a directory does not contain the inodes of the files in the directory.*

D. The inode of a symbolic link contains the inode number of the target of the link.

NO. *The inode of a symbolic link contains information that determines if it is a symbolic link or not. The inode does not contain the inode number of the target of the link.*

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 an object file (a ".o" file) or a library file. Within each context, symbol resolution is done using a table lookup.

YES.

B. `ld` resolves symbols by searching through a set of contexts, where each context is an object file (a ".o" file) or a library file. Within each context, symbol resolution is done using pathname resolution.

NO.

C. `ld`'s output when operating on a set of files depends on the order in which the different object files (".o" files) and library files are specified on the command line.

YES. *The order of the object files doesn't matter, but the order of the library files does.*

D. None of the above.

NO.

6. [6 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 attempts to enforce modularity between X clients in their use of the display resource.

YES.

B. An X client always uses synchronous remote procedure calls to communicate with the X server.

NO. *Usually, an X client uses* asynchronous *communication with the X server for better interactivity.*

C. To obtain high performance, the entire X server is implemented in the kernel of the operating system of the machine on which it runs.

NO. *The X server runs almost entirely at user level.*

7. [8 points]: According to the authors of the MapReduce system (reading #8), which of the following mechanisms does MapReduce use?

(Circle ALL that apply)

A. It executes different "map" processes in parallel.

YES. *This is one of the primary sources of concurrency in the computation. The input (key, value) pairs are distributed across the machines. The (key, value) pairs are (usually) processed by a map computation located on that machine that executes in parallel with map computations running on other machines.*

B. It places "reduce" computations on machines that have the data that the computation will access locally available.

NO. *In general, a single reduce computation may need data produced by arbitrary map computations. MapReduce addresses this issue by transferring all of the (key, value) pairs with the same key (these pairs are in general produced by map computations distributed across all of the machines) to the machine running the reduce computation, which in turn will process the pairs.*

C. It writes the output of each reduce computation to disks on different machines.

YES. *The output of a reduce computation is written to a file in the Google file system. This file system writes each file to disk storage on two separate machines.*

D. It may execute the same reduce computation at the same time on different machines and use the first results that are returned.

YES. *MapReduce uses this technique to handle straggler threads. There might also be more than one reduce computation running if the Master is unable to communicate with a reduce worker, and therefore starts up another reduce worker to do the same reduce computation.*

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 that runs on the server (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 alloted a spot.

```
int procedure FIND_SPOT()
 1. while (TRUE){
 2.   for (i ← 0; i < NSPOTS; i ← i + 1){
 3.     if (available[i] == TRUE) then{
 4.       available[i] ← FALSE;
 5.       numcars ← numcars + 1;
 6.       return (i);     // Allocate spot i for client to use
 7.     };
 8.   }
 9. }

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 $[0, \ldots \text{NSPOTS} - 1]$ 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 at any time (1) spots are available and no parked car ever leaves in the future, (2) there are no outstanding FIND_SPOT() requests, and (3) exactly one client makes a FIND_SPOT request, then the client should get a spot.

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.

Make the following assumptions:

A. The statements to update $numcars$ are *not atomic*; they each take multiple instructions to run.
B. The server runs on a single processor with a preemptive thread scheduler.
C. The network delivers RPC messages reliably, and there are no network, server, or client failures.
D. Cars arrive and leave at random.
E. ACQUIRE() and RELEASE() are as defined in chapter 5.

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 violate the correctness specification when two FIND_SPOT() threads run.

YES. *If two* FIND_SPOT*() threads run concurrently and both update the same location in available[ ], a race condition and error occurs that causes the correctness specification to be violated.*

B. There is a race condition in accesses to $available[]$, which may violate the correctness specification when one FIND_SPOT() thread and one RELINQUISH_SPOT() thread runs.

NO. *Although the two threads can concurrently access* $available[i]$ *for some* i*, the specification is not violated.*

C. There is a race condition in accesses to $numcars$, which may violate the correctness specification when more than one thread updates $numcars$.

YES. *The specification may be violated because of this race condition whenever any two threads run concurrently.*

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.

NO. *The* average time *between requests may be larger than the average processing delay, but particular requests may arrive concurrently and violate the specification because of the race conditions mentioned in choices A and C above.*

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 ensures that the correctness specification is met? (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()
    while (TRUE){
→   ACQUIRE(avail_lock);
      for (i ← 0; i < NSPOTS; i ← i + 1){
        if (available[i] == TRUE) then{
          available[i] ← FALSE;
          numcars ← numcars + 1;
→         RELEASE(avail_lock);
          return (i);     // Allocate spot i for client to use
        }
      }
→     RELEASE(avail_lock);
    }

  procedure RELINQUISH_SPOT(int spot)
      // Called when a client car is leaving spot
→   ACQUIRE(avail_lock);
    available[spot] ← TRUE;
    numcars ← numcars − 1;
→   RELEASE(avail_lock);
```

YES. *This method protects the entire $available$[] array, and prevents race conditions. It meets the correctness specification.*

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

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

  procedure RELINQUISH_SPOT(int spot)
→   ACQUIRE(avail_lock);
    available[spot] ← TRUE;
    numcars ← numcars − 1;
→   RELEASE(avail_lock);
```

NO. *This code appears to work, but has a bug because certain* ACQUIRE() *calls don't have a matching* RELEASE()*. The result is that condition C of the specification is violated. Suppose* *a thread acquires the lock in* FIND_SPOT*() but* $available[i]$ *is FALSE for some* i*. When it tries to acquire the lock for the next* i*, it will "freeze", and the car won't be able to get a spot even though spots may be available.*

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

```
int procedure FIND_SPOT()
    while (TRUE){
      for (int i ← 0; i < NSPOTS; i ← i + 1){
→       ACQUIRE(avail_lock);
        if (available[i] == TRUE) then{
          available[i] ← FALSE;
          numcars ← numcars + 1;
→         RELEASE(avail_lock);
          return (i);    // Allocate spot i for client to use
→       }else RELEASE(avail_lock);
      }
    }

procedure RELINQUISH_SPOT(int spot)
→   ACQUIRE(avail_lock);
    available[spot] ← TRUE;
    numcars ← numcars − 1;
→   RELEASE(avail_lock);
```

YES. *This code works correctly.*

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

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

procedure RELINQUISH_SPOT(int spot)
   available[spot] ← TRUE;
   numcars ← numcars − 1;
→  RELEASE(avail_lock);
```

NO! *This code is actually quite weird—it does not have any race conditions, but it leads to a situation where at most one car can be in the parking lot at any one time! It violates correctness condition C.*

For the rest of the questions, assume that Ben picked a correct answer to Question #9. Ben now decides to combat parking at a truly crowded location... Gillette Stadium, where there are always cars looking for spots! 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, assuming that multiple cars may be making requests?

A. The client will not get a response until at least one car relinquishes a spot.

YES. *If there are no spots, and the only time the server responds is when it allocates a spot, then obviously the statement above must hold.*

B. The client may never get a response even when other cars relinquish their spots.

YES. *This system may display "starvation", where a car may never get a spot because other cars keep entering the system and get any spot that frees up.*

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. Alyssa warns Ben that this change may cause the system to violate the correctness specification.

11. [8 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.

YES. *The server runs a thread every time a client makes a request. If a client times out and retries, there are now two threads running at the server on behalf of the same client car.*

B. The server may assign the same spot to two cars making requests.

NO/YES! *There was an ambiguity as to whether the RPC timeouts applied to* RELINQUISH_SPOT*() or not. The problem is that the lead-in to this question suggested that it applied only to* FIND_SPOT*(). That was in fact the original intent of the question, but it was never clearly specified. If it only applies to* FIND_SPOT*(), then the answer is "No"; otherwise, it is "Yes". We did not consider this choice while grading the exam.*

C. $numcars$ may be smaller than the actual number of cars parked in the parking lot.

NO/YES! *There was an ambiguity as to whether the RPC timeouts applied to* RELINQUISH_SPOT*() or not. The problem is that the lead-in to this question suggested that it applied only to* FIND_SPOT*(). That was in fact the original intent of the question, but it was never clearly specified. If it only applies to* FIND_SPOT*(), then the answer is "No"; otherwise, it is "Yes". We did not consider this choice while grading the exam.*

D. $numcars$ may be larger than the actual number of cars parked in the parking lot.

YES. *A car may get multiple spots allocated to it by the server, and the server will increment* $numcars$ *for each allocation. But the car can be parked only in one of those spots!*

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 saves the values of the PMAR, PC, SP, and several registers.

NO. *There's no need to save the PMAR because the address space is the same.*

B. On any thread switch, the operating system saves the values of the PC, SP, and several registers.

YES.

C. On any thread switch between two RELINQUISH_SPOT() threads, the operating system saves *only* the value of the PC, since RELINQUISH_SPOT() has no return value.

NO. *The threads need separate stacks and may have used registers.*

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.

NO. *The work required to save the stack pointer is not proportional to the size of the 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, which uses the RSL (Read and Set Lock) instruction. 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 (defined on page 5-43). To help you, we have given some scaffolding pseudocode, 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. Don't worry about "wraparound" problems with the eventcount and integer variables. Also, an application calling ACQUIRE without a corresponding RELEASE is a bug in the application, not a bug in your implementation of ACQUIRE and RELEASE.

Solution: Let's start with the solution that uses eventcounts as they were intended to be used.

```
// 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) {
        wait(released, num_acquired);
        RSL L, R0;
    }

    num_acquired = num_acquired + 1;

} // end of ACQUIRE()

procedure RELEASE(L) {

    released = released + 1;
    L = FALSE;
    notify(released);
}
```

The basic idea in the preceding solution is to keep track of the number of times the lock has been acquired and released. If it has been acquired more times than it has been released, threads trying to acquire the lock should call wait() explicitly to suspend (and avoid consuming unnecessary CPU cycles) until the lock is released, at which point they should try again to acquire the lock. A deeper analysis would say that the important property to track is whether the lock is acquired or free, have ACQUIRE check the property and suspend if the lock is acquired, then be sure that RELEASE signals any suspended ACQUIRE threads when it releases the lock so that the other threads can go around again and try to acquire the newly free lock. In the preceding solution, the value of the expression (released - num_acquired) is 1 if the lock is free and 0 if the lock is acquired. But as long as we maintain some property that 1) tells us when the lock is free and when it is acquired and 2) we can check with wait, we should be able to use this property to obtain a solution.

Here are some alternate solutions based on this analysis. They are more confusing and harder to reason about.

```
  // 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) {
        wait(released, num_acquired);
        RSL L, R0;
    }

    num_acquired = num_acquired + 1;

} // end of ACQUIRE()

procedure RELEASE(L) {

    num_acquired =
      num_acquired - 1;
    L = FALSE;
    notify(released);
}
```

In the preceding solution, num_acquired is 1 when the lock is acquired and 0 when it is free; released does not change and is therefore always 1. The following dual solution uses released instead of num_acquired — in this solution, released is 1 when the lock is free and 0 when it is acquired. num_acquired increases without bound, but is irrelevant to the computation.

```
  // 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) {
        wait(released, 0);
        RSL L, R0;
    }
    released = 0;

procedure RELEASE(L) {

    released = 1;
    L = FALSE;
    notify(released);
}
```

```
    num_acquired = num_acquired + 1;

} // end of ACQUIRE()
```

The following code is semantically identical to the preceding solution, but a bit more complicated.

```
  // 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) {
        wait(released, 0);
        RSL L, R0;
    }
    released = released - 1;
    num_acquired = num_acquired + 1;

} // end of ACQUIRE()

procedure RELEASE(L) {

    released =
      released + 1;
    L = FALSE;
    notify(released);
}
```

The following alleged solution is INCORRECT because of a race condition on `released`.

```
  // 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) {
        wait(released, num_acquired);
        RSL L, R0;
    }

    num_acquired = num_acquired + 1;

} // end of ACQUIRE()

procedure RELEASE(L) {

    L = FALSE;
    released =
      released + 1;
    notify(released);
}
```

Here is what can go wrong.

- A thread acquires the lock.
- The thread releases the lock, executing the statement `L = FALSE;`. But the thread is preempted before it can execute the next statement `released=released+1;`. Another thread comes in, acquires the lock, computes, then releases the lock.
- At this point there is a race condition on `released`. Note that the statement `released=released+1;` does not necessarily increment `released` atomically — it reads `released`, adds one to the read value, then writes the new value back. It is therefore possible for the value of `released` to become incorrect in that it may be less than the number of times the lock has been released.
- If `released` is less than it should be, the following steps can cause `ACQUIRE` to wait when it should not.
- A thread acquires the lock.
- Another thread T attempts to acquire the lock and is preempted just before the call to wait.
- The thread holding the lock releases the lock.
- The thread T resumes, then suspends in `wait`. The correct behavior is for T to not suspend in `wait`, but to instead return back out of `wait` immediately and acquire the lock.

End of Quiz I!