Department of Electrical Engineering and Computer Science

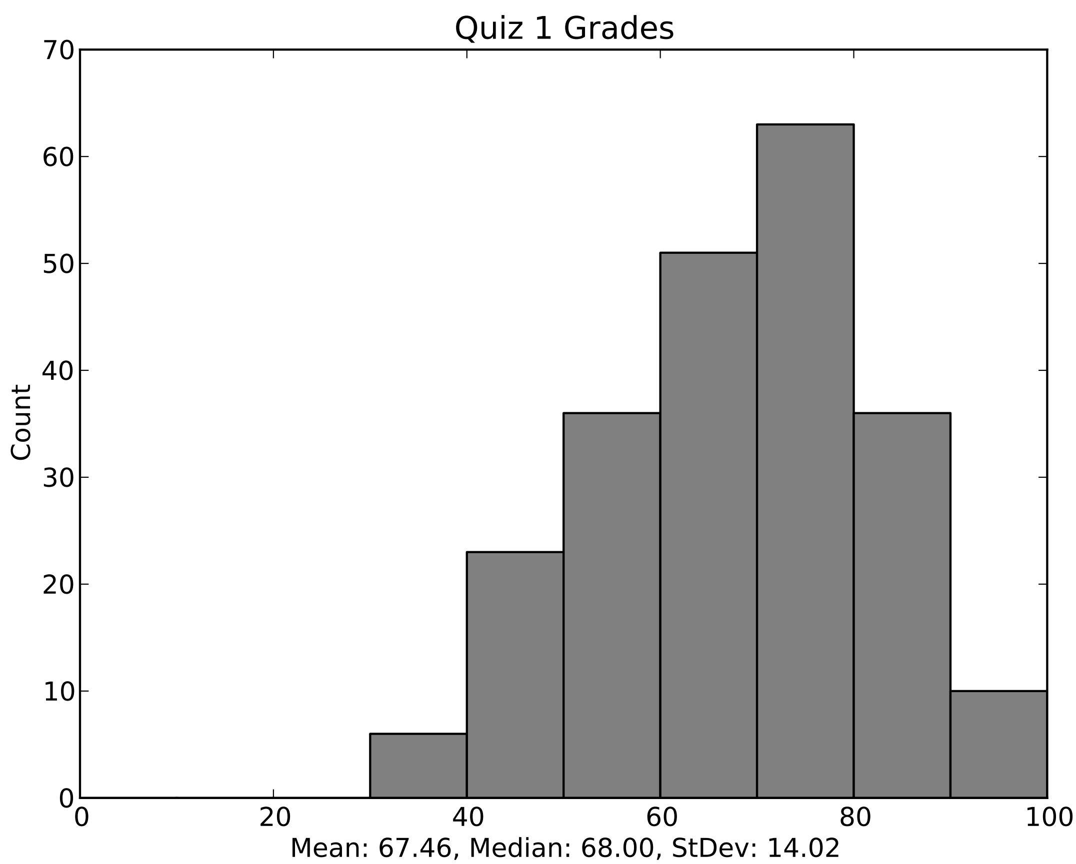
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2012

Quiz I Solutions

There are 13 questions and 9 pages in this quiz booklet. Answer each question according to the instructions given. You have **50 minutes** to answer the questions.

Grade distribution histogram:



I Reading Questions

1. [4 points]: Simon, in "The architecture of complexity," claims that hierarchical systems evolve more quickly than non-hierarchical systems of comparable size. Which of the following arguments does he say support this claim.

(Circle True or False for each choice.)

A. True / False The observation that complex systems are more comprehensible if they are neatly decomposable.

Answer: False.

B. True / False The parable of the watchmakers.

Answer: True.

C. True / False Thermodynamic considerations about entropy.

Answer: False.

D. True / False The distinction between state descriptions and process descriptions.

Answer: False.

2. [5 points]: Answer the following questions based on the X Window System paper.

(Circle True or False for each choice.)

A. True / False When a Web browser runs on a Unix workstation and displays its pages on that workstation using X, the browser is the client of both the web server and the X server.

Answer: True.

B. True / False The window manager must be built into the core of the X server because it can operate on the windows of multiple clients.

Answer: False.

C. True / False The X server of the 1980's has a complex management infrastructure for color map management because display controllers of that era could not provide enough memory to store the color of every pixel in each image.

Answer: True.

D. True / False The X server informs its clients when a region of one of their windows becomes obscured so that the clients can stop sending update requests for that region.

Answer: False.

E. True / False Synchronization errors could happen between the X server and its clients when network latencies delayed client responses.

Answer: True. See section 9.3 in the paper.

Initials:

3. [6 points]: Answer the following questions based on the Unix paper.
(Circle True or False for each choice.)

A. True / False Checking the return value of the `fork()` function enables a child process to execute different instructions from its parent.

Answer: True.

B. True / False Since a child process can write to all files that are open by its parent at the time of the fork, these writes can create a race condition with writes from the parent.

Answer: True. The two processes can write to their respective file descriptors in any order, since there is no file locking in the original version of Unix.

C. True / False One of the advantages of multitasking is that it makes the system more responsive to user inputs.

Answer: True.

4. [6 points]: This question is in the context of the Eraser paper. Assume a multi-threaded program has three locks: `mu`, `mu0`, and `mu1`, as well as an array `a` with two locations, `a[0]` and `a[1]`. Whenever a thread is about to modify the whole array (i.e., both `a[0]` and `a[1]`) it acquires the lock `mu`, but whenever it is about to modify `a[0]` alone it acquires `mu0`, and whenever it is about to modify `a[1]` alone it acquires `mu1`.

A. Yes / No Could Eraser's lockset algorithm detect a race condition with respect to accesses to either `a[0]` or `a[1]`?

Answer: Yes. The lockset for `a[0]` will be empty, because in some cases it is accessed with `mu` held, and others with `mu0` held. Eraser's algorithm reports a possible race condition when the lockset becomes empty.

5. [6 points]: This question continues the Eraser question. Assume a multi-threaded program has one lock `mu2` and an array `a` with two locations, `a[0]` and `a[1]`. Whenever a thread is about to modify either `a[0]` or `a[1]` it acquires the same lock `mu2`.

A. Yes / No Could Eraser's lockset algorithm detect a race condition with respect to accesses to either `a[0]` or `a[1]`?

Answer: No. Eraser's lockset for both `a[0]` and `a[1]` will contain `mu2` and will never be empty for Eraser to report a race.

II Complexity

6. [10 points]: Chapter 1 of the text describes several techniques for coping with complexity: Modularity (M), Abstraction (A), Layering (L), Hierarchy (H), Design for iteration (D), and Indirection (I).

Initials:

For each of the following advantages, mark the appropriate letter (M, A, L, H, D, I) or N for "none of these" to say which technique *best* provides that advantage. For each question, there is only one best answer, but a given technique might be the best answer to more than one question.

A. M A L H D I N Helps the designers incorporate feedback in system implementations.

Answer: D.

B. M A L H D I N Helps simplify the task of debugging a complex system by letting implementers deal with smaller components

Answer: M.

C. M A L H D I N Makes it easier for designers to take advantage of delayed binding in system implementations.

Answer: I.

D. M A L H D I N Ensures that the implementation will obey the robustness principle.

Answer: A.

E. M A L H D I N If this is done correctly, it can help reduce the number of inter-module interactions in large systems.

Answer: H.

III Names

7. [5 points]: One of the examples in the first hands-on exercise asked you to notice that even though your home directory might be `/mit/YOU`, the sequence `cd /mit/6.033; cd ../YOU` when executed in the `tcsh` shell does not get you to your home directory. However, if you perform the same experiment in `bash`, it works. From the viewpoint of our name resolution discussion, which statement is correct in `bash`?

(Circle the BEST answer)

A. When executing the `cd ..` command, bash determines if any shorter symbolic links exist to the resulting directory and displays the shortest one.

B. The bash shell uses not only the current directory as its name mapping context but also some additional history of how the current directory was reached.

C. Bash uses only Unix pathnames, not inodes.

D. None of the above.

Answer: B.

IV Concurrency

In this question, you can assume that loads and stores to variables are atomic, and neither the compiler nor hardware will ever reorder instructions. `continue` transfers control flow to the beginning of the `while` loop.

Consider the following lock implementation using a variable `x` for threads numbered 1 through `n` where initially `x` = 0. Each thread's number is stored in variable `i`.

```
acquire():
  while True:
    if x != 0:
      continue  ## retry
    x = i
    if x != i:
      continue  ## retry
    return
```

```
release():
  x = 0
```

8. [10 points]: Which of the following is true for the above algorithm:
(Circle the BEST answer)

A. It does not guarantee mutual exclusion.

B. it guarantees mutual exclusion but not deadlock freedom.

C. it guarantees both mutual exclusion and deadlock freedom.

Answer: A. Two threads can both check `x != 0` and succeed. Then, thread 1 will set `x = 1` and verify that `x != 1` is false. Then, thread 2 will set `x = 2` and verify that `x != 2` is false. Both then return from `acquire`.

Consider the following lock implementation using variables `x` and `y` for threads numbered 1 through `n`, where each thread's number is stored in variable `i`, and where initially `y` = 0:

```
acquire():
  while True:
    x = i
    if y != 0:
      continue  ## retry
    y = 1
    if x != i:
      continue  ## retry
    return
```

```
release():
  y = 0
```

9. [10 points]: Which of the following is true for the above algorithm:
(Circle the BEST answer)

A. It does not guarantee mutual exclusion.

B. it guarantees mutual exclusion but not deadlock freedom.

C. it guarantees both mutual exclusion and deadlock freedom.

Answer: B. The algorithm guarantees mutual exclusion but not deadlock freedom. On an intuitive level, the algorithm guarantees mutual exclusion because once some thread passes `y=1`, all threads that excute `x=i` later will get stuck at `y!=0`, and among any collection of threads that pass the `y!=0` test concurrently and race to get to `x!=i`, only the one that wrote last in `x=i` can get past `x!=i`, and all later threads will never get past `y!=0` until this winning thread leaves the critical section and executes `release`.

To give you a sense of what it takes to rigorously prove the correctness of this lock implementation, see a complete proof at `http://web.mit.edu/6.033/2012/wwwdocs/assignments/s12_1_9.pdf`.

V Client/Server and bounded buffers

Consider the following bounded buffer code (`send` and `receive`), assuming the variables `bb.in` and `bb.out` are 64 bits, never overflow, can be read and written atomically, and neither the compiler nor hardware will ever reorder instructions:

```
send(bb, m):
  // each invocation of send has
  // its own local variables:
  //   my_send_index (64-bit int)
  while True:
    acquire(bb.lock)
    if bb.in - bb.out < N:
      my_send_index = bb.in
      bb.in = bb.in + 1
      release(bb.lock)
      bb.buf[my_send_index mod N] = m
      return
    release(bb.lock)
```

```
receive(bb):
  // each invocation of receive
  // has its own local variables:
  //   my_rec_index (64-bit int)
  //   m (message)
  while True:
    acquire(bb.lock)
    if bb.in > bb.out:
      my_rec_index = bb.out
      bb.out = bb.out + 1
      release(bb.lock)

      acquire(bb.rec_lock)
      m = bb.buf[my_rec_index mod N]
      release(bb.rec_lock)

      return m
    release(bb.lock)
```

10. [12 points]: Which of the following is true for the above implementation:
(Circle True or False for each choice.)

A. True / False The code is correct if there is one sender and one receiver executing at same time.

Initials:

Answer: False. This implementation of `send` and `receive` is broken in all cases. The `send` function increments `bb.in` and releases the lock without placing the message in the buffer, which means a concurrent `receive` can read an uninitialized message from the buffer at the `bb.in` location.

Since this code is not correct for a single sender and receiver, it is also incorrect for multiple senders or receivers.

B. True / False The code is correct if there is one sender and many receivers executing at same time.

Answer: False.

C. True / False The code is correct if there are many senders and one receiver executing at same time.

Answer: False.

D. True / False The code is correct it there are many senders and many receivers executing at same time.

Answer: False.

VI Operating systems

11. [8 points]: Circle all of the function calls that directly correspond to system calls in a Unix system (based on the Unix paper) in the following implementation of the `cp` program:

```
void cp(char *srcpath, char *dstpath) {
  int src = open(srcpath, O_RDONLY);
  int dst = open(dstpath, O_WRONLY);

  if (src < 0 || dst < 0)
    exit(-1);

  while (1) {
    char buf[1024];
    ssize_t cc = read(src, buf, sizeof(buf));
    if (cc <= 0)
      break;
    ssize_t n = write(dst, buf, cc);
    assert(cc == n);
  }

  close(dst);
  close(src);
}

int main(int argc, char **argv) {
  cp(argv[1], argv[2]);
  exit(0);
}
```

Answer: open, open, exit, read, write, close, close, exit.

12. [10 points]: Suppose we run two user-mode programs, A and B, which are independent (e.g., which do not access any common files), on a Unix OS, and we run the Unix OS as a guest in a virtual machine (VM). What can fail if a bug (such as a divide-by-zero or a random memory write) appears in different components of the system? Assume there are no other bugs. Draw an `X` in the appropriate locations in the table below:

	Program A fails	Program B fails	Guest OS kernel fails	VMM fails
Bug in program A	X			
Bug in program B		X		
Bug in the guest OS kernel	X	X	X	
Bug in the VM monitor	X	X	X	X

Answer: see above.

13. [8 points]: Suppose that you discover a bug in the implementation of `read()` that is invoked by `cp`, and you want to fix this bug.

A. True / False Fixing this bug will require modifying the `cp` program.

Answer: False. The code for `read` is in the OS kernel.

B. True / False Fixing this bug will require modifying the OS kernel.

Answer: True. The code for `read` is in the OS kernel.

Initials: