



*Department of Electrical Engineering and Computer Science*

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

**6.033 Computer Systems Engineering: Spring 2010**

# **Quiz I Solutions**

## I Reading Questions

1. [9 points]: Based on your deductions from the UNIX paper by Ritchie and Thompson (reading #5), which of the following statements are true?

- A. **True / False** If you follow a shell command with `&`, the shell will make a new process in which to run the command while the shell goes on in parallel to read and execute the next command, but if you don't, the shell will not create a new process but rather run the command to completion in the shell process.

**Answer:** False. There's always a new process created, so that each command can have its own address space and be prevented from interfering with the shell.

- B. **True / False** A process can't tell whether its standard input is a file or a pipe.

**Answer:** False. You can't seek on a pipe.

- C. **True / False** The `exec` kernel call is convenient but not essential for the shell to invoke a command.

**Answer:** True. The `exec` call replaces the memory of the process with the contents of a file and then passes it the arguments. A process could do this to itself, by reserving a small amount of memory to hold ordinary user mode code that does these things.

2. [6 points]: According to the X-Windows paper by Sheifler and Gettys (reading #6), which of the following statements are true?

- A. **True / False** When two windows overlap and the bottom window is brought to the front, the server immediately draws the window using a cached image.

**Answer:** False. The client is responsible for redrawing.

- B. **True / False** Clients can send the server not just images for the server to draw, but also "higher level" primitives such as lines, rectangles, and text.

**Answer:** True.

- C. **True / False** When the user clicks on a top-level window, the X Server brings the window to the front, if it is not already the top-most window.

**Answer:** False. This is the role of the window manager, which is a client of the X server.

- D. **True / False** A client can only communicate with and draw graphics on a single server.

**Answer:** False.

**3. [5 points]:** Based on the description of the Therac-25 in the paper by Leveson and Turner (reading #4), which of the following statements are true?

- A. True / False** The hardware interlocks present in the Therac-20 were also present in the Therac-25.

**Answer:** False.

- B. True / False** A detailed fault tree analysis of the Therac-25 estimated the probability of the wrong mode being selected to be  $4 \times 10^{-9}$ .

**Answer:** False. The paper does not mention that any such analysis generated that probability.

- C. True / False** The Therac-25 software acquired locks in the wrong order, leading to disastrous consequences.

**Answer:** False. The paper does not say that any disaster was caused by incorrect lock order.

**4. [7 points]:** Based on the description in the MapReduce paper by Dean and Ghemawat (reading #8), which of the following statements are true?

- A. True / False** MapReduce guarantees that each map task is executed only once to preserve functional behavior.

**Answer:** False. It may run a task more than once if there is a failure.

- B. True / False** File renaming is used to ensure that only a single execution of a reduce task is represented in the final output.

**Answer:** True.

- C. True / False** MapReduce always schedules two instances of every task (corresponding to the GFS replicas of the input data) to guard against worker failure and stragglers.

**Answer:** False.

- D. True / False** Each map task is automatically distributed so its output is read only by a single reduce task.

**Answer:** False.

- E. True / False** Suppose that a programmer writes a map operator that has a bug that causes it to fail non-deterministically. During execution, five map tasks fail. This MapReduce job will still execute to completion.

**Answer:** True.

- F. True / False** It is possible for the master to incorrectly conclude that a reduce task has failed, even though it is still running (e.g., due to a temporary network connection failure). In this case, the master will start another reduce task, and both tasks could complete execution of the same set of reduce operations.

**Answer:** True. GFS will ensure that the output file is updated atomically, so that only one of the two reduce tasks contributes to the final output.

**G. True / False** No single machine failure will prevent a MapReduce computation from successfully completing.

**Answer:** False. The master is not replicated, so if it fails the computation cannot continue.

**5. [12 points]:** The following question refers to the Eraser system, by Savage et al. (reading #7). Suppose you have a banking application with an `Account` object protected by a lock and a function `Change()` to deposit funds into the account (a negative `Change()` is a withdrawal):

```
structure Account
  int balance initially 0
  lock acct_l initially unlocked

Account allAccounts[] // array of all accounts

procedure Change(Account a, int amount) returns int
  int newBal
  acquire(a.acct_l)
  a.balance = a.balance + amount
  newBal = a.balance
  release(a.acct_l)
  return newBal
```

`Change()` is called by `Transfer()`, which moves funds from one account to another, leaving the total balance in all of the accounts unchanged.

```
procedure Transfer(Account from, Account to, int amount)
  Change(from, 0 - amount)
  Change(to, amount)
```

More than one thread might be executing `Transfer()` at the same time. In addition, there is a thread `Total` that periodically runs the following function to add up all the account balances:

```
procedure TotalBalance() returns int
  int total = 0
  for each a in allAccounts
    total = total + Change(a, 0)
  return total
```

These are the only operations that touch an account. You should assume that the arithmetic operations do not overflow. `Change()` is never called directly; it is only called via `Transfer()` or `TotalBalance()`.

**A. True / False** When run with the program above, Eraser will not issue any warnings.

**Answer:** True. The code always protects each account's balance with that account's lock.

**B. True / False** If one replaced the call to `Change(a, 0)` in `TotalBalance()` with `a.balance`, Eraser would not issue any warnings.

**Answer:** False. Eraser would complain because sometimes a balance variable would be protected by a lock, and sometimes not.

- C. True / False** If one or more threads call `Transfer()`, then after all the transfers have completed the sum of the account balances is the same as before they started.

**Answer:** True. Each call of `Transfer(from, to, amount)` adds `amount` to `from.balance` and subtracts `amount` from `to.balance`, leaving the total unchanged. `Change()` waits for a lock on the balance that it updates, so no update will be lost even if multiple threads call `Transfer()` with the same account.

- D. True / False** If the `Total` thread runs while other threads are executing `Transfer()`, then each call to `TotalBalance()` will return the same value.

**Answer:** False. There is no lock held for the entire work of `Total`, so for example, if `Total` looks at “from” before a `Transfer(from, to, amount)` starts and at “to” after the `Transfer` is done, it will see a total balance that is too small by “amount.”

- E. True / False** If no other thread calls `Change()` during the time that a single call of `TotalBalance()` is running, then any two calls of `TotalBalance()` will give the same result.

**Answer:** False. A call to `Transfer()` might make one of the calls to `Change()`, then a `TotalBalance()` might run, and then the call to `Transfer()` might make the other call to `Change()`.

## II Zed and Ned Wrestle With Threads

Zed is running a server on the Internet that keeps people informed of the latest known locations of important entities such as Elvis, Bigfoot, and the Loch Ness monster. Zed's server accepts one RPC:

```
whereis(entity)
```

The server uses a file system like the one described in the UNIX paper. It maintains one file per entity. Each file contains the entity's last reported location; Zed updates the files manually. Here's what Zed's server code looks like:

```
procedure dispatch()
  while true:
    wait for an RPC request from any client
    parse the request to extract the entity
    result = whereis(entity)
    send RPC reply containing result

procedure whereis(string entity)
  fd = open(entity) // for reading
  location = read(fd)
  close(fd)
  return location
```

Zed's server computer initially has one CPU and one disk. His software is a single program with one thread. The server's operating system queues incoming RPC requests that arrive while `dispatch()` is busy with a previous request. Zed's system does not cache files or disk blocks. Each call to `read()` has to wait for the disk to seek and rotate, which you should assume takes a fixed total time of 10 milliseconds. `open()` and `close()` do not involve any disk activity. Parsing an RPC takes 10 milliseconds of CPU time; none of the other activities in the server takes a significant amount of CPU time. The Internet delivers messages between client and server in zero time, and has infinite throughput.

**6. [6 points]:** 4 clients send `whereis()` RPCs to the server at the same time. What is the average of the latencies perceived by the four clients?

**(Circle the best answer.)**

- A. 5 milliseconds
- B. 10 milliseconds
- C. 20 milliseconds
- D. 30 milliseconds

E. 50 milliseconds

**Answer:** 50 milliseconds. The four requests take 20, 40, 60, and 80 milliseconds, averaging 50.

Zed's friend Ned has taken 6.033 and advises him that using threads can help improve performance. Zed changes his server to use a pre-emptive threading system that switches in round-robin among runnable threads 10,000 times per second. Zed's only change is in `dispatch()`, which he modifies so that it starts a new thread for each RPC request:

```
procedure dispatch()
  while true
    wait for an RPC request from any client
    allocate_thread(do_dispatch) // create and run thread

procedure do_dispatch()
  parse the request to extract the entity
  result = whereis(entity)
  send RPC reply containing result
  destroy this thread
```

The main loop of `dispatch()` does not wait for the thread that it creates for each request. Thus, if multiple requests arrive in quick succession, `dispatch()` will create multiple threads.

**7. [6 points]:** Again, 4 clients send `whereis()` RPCs to the server at the same time. What is the average of the latencies perceived by the four clients?

**(Circle the best answer.)**

- A. 12.5 milliseconds
- B. 20 milliseconds
- C. 35 milliseconds
- D. 65 milliseconds
- E. 100 milliseconds

**Answer:** 65 milliseconds. The pre-emptive thread system interleaves the four requests' parsing, so that they all finish at the same time, after 40 milliseconds. Thus the four requests complete after 50, 60, 70, and 80 milliseconds, averaging 65.

Zed is still looking for ways to improve performance. He upgrades his server so that it has eight CPUs with shared memory. His threading system will use multiple CPUs if there are multiple runnable threads.

**8. [6 points]:** Again, 4 clients send `whereis()` RPCs to the server at the same time. What is the average of the latencies perceived by the four clients?

**(Circle the best answer.)**

- A. 5 milliseconds
- B. 12.5 milliseconds
- C. 20 milliseconds
- D. 35 milliseconds
- E. 50 milliseconds

**Answer:** 35 milliseconds. After 10 milliseconds, all of the requests have finished parsing. Then they must take turns waiting for the disk, so they take 20, 30, 40, and 50 milliseconds, averaging 35.

Zed decides to add a second disk to his 8-CPU server. He puts the file for Elvis on one disk, and the file for Bigfoot on the other disk. Requests for each entity only use the one disk that the entity is stored on.

**9. [6 points]:** Four clients send `whereis()` RPCs to the server at the same time. One client sends a request for Elvis; the other three send requests for Bigfoot. What is the average of the latencies perceived by the four clients?

**(Circle the best answer.)**

- A. 12.5 milliseconds
- B. 17.5 milliseconds
- C. 20 milliseconds
- D. 27.5 milliseconds
- E. 35 milliseconds

**Answer:** The Elvis request and one of the Bigfoot requests run entirely in parallel, since each has its own CPU and disk. The other two Bigfoot requests must wait for the disk. So the requests take 20, 20, 30, and 40 milliseconds, averaging 27.5.

Zed adds a cache to his system. The cache keeps a copy of the information about the entity most recently read by `whereis()`. The cache can only hold a single entity's information. Zed's caching code looks like this:

```
lock cache_lock
string cache_entity
string cache_content
int hits = 0
```



```
int misses = 0

procedure whereis(string entity)
  acquire(cache_lock)
  if cache_entity == entity
    val = cache_content
    hits = hits + 1
  else
    fd = open(entity) // for reading
    location = read(fd)
    close(fd)
    val = location
    cache_content = val
    cache_entity = entity
    misses = misses + 1
  release(cache_lock)
  return val
```

Zed's code keeps track of the number of cache hits and misses to help him understand the performance of his system. As before, the server has two disks with Elvis and Bigfoot on different disks, the server has 8 CPUs, and `dispatch()` creates a new thread for each request.

**10. [6 points]:** Four clients send `whereis()` RPCs to the server at the same time. One client sends a request for Elvis; the other three send requests for Bigfoot. Before these requests, the `hits` and `misses` counters started with value zero, and `cache_entity` was neither Elvis nor Bigfoot. What values for the `hits` counter are possible after the server has answered all four requests? Circle True for each value of `hits` that is possible, and False for each value that is not possible.

- A. True / False 0
- B. True / False 1
- C. True / False 2
- D. True / False 3
- E. True / False 4

**Answer:** 1 and 2. There are four possible orders in which Elvis and Bigfoot requests check the cache (EBBB, BEBB, BBEB, and BBEB). These involve 2, 1, 1, and 2 hits respectively.

**11. [6 points]:** What is the **shortest** time that it could take for all four RPCs to finish in the previous question's scenario?

(Circle the best answer.)

- A. 10 ms
- B. 20 ms
- C. 30 ms
- D. 40 ms
- E. 50 ms

**Answer:** 30 milliseconds. All four requests finish parsing in 10 milliseconds. The minimum number of misses is two (from the previous question). The lock in the caching code causes disk requests to proceed one at a time even though there are two disks. So the minimum total time is 30 milliseconds.

### III Bank of Ben

Ben Bitdiddle is building a server to store and manipulate bank account balances. His server provides several routines:

```
int balances[NUM_ACCOUNTS] // array of accounts

procedure get_balance(account) returns int
    return balances[account]

procedure transfer(account1, account2, amount)
    balances[account1] = balances[account1] - amount
    balances[account2] = balances[account2] + amount
    return amount
```

Clients issue RPCs to the server to invoke `get_balance()` and `transfer()`. Ben uses the multi-threaded RPC `dispatch()` routine used in Ned's server above for processing these requests (page 7), except that it calls `get_balance()` or `transfer()` rather than `whereis()`.

To demonstrate his server, Ben writes a graphical user interface (GUI) client that connects to the server and performs `get_balance()` or `transfer()` operations in response to user-supplied commands.

Ben's server and GUI are written in a language like C that allows a buggy program to write anywhere in its memory. Ben's machine has one processor with one core.

Ben runs his GUI and server in separate address spaces on the same machine.

**12. [15 points]:** Ben is concerned that his code might be slow and incorrect, so he comes to you for help. Below, Ben proposes several modifications to his banking application. For each choice, tell Ben whether it would:

- (a) Enforce modularity by making it less likely that bugs in the GUI affect the internal operation of `get_balance()` and `transfer()`.
- (b) Improve throughput without introducing additional sources of incorrect results.
- (c) Eliminate sources of incorrect results in the presence of multiple simultaneous client threads (e.g., GUI instances).
- (d) None of the above

For each of the following proposed modifications, indicate which of the above effects (a—d) the modification would produce. Indicate only the one best answer.

- A.** Proposed modification: Cache the results of RPC calls to `get_balance()` in the GUI, while still running `transfer()` calls on the RPC server. This takes the form of a new client-side RPC stub for `get_balance()`:

```
procedure get_balance_stub(acct) returns int:
    if (acct not in cache)
        cache[acct] = result of sending get_balance(acct) to RPC server
    return cache[acct]
```

Effect of modification: **Answer:** d. This modification might improve performance, but it adds a new source of errors, because one client might cache a stale balance that another client subsequently asks the server to update.

- B.** Proposed modification: Modify the operating system kernel to maintain the account balances and add system calls that the client makes to ask the kernel to perform the `get_balance()` and `transfer()` operations. Assume that system calls take a significant amount of time, and that kernel routines may be pre-empted (forced to yield).

Effect of modification: **Answer:** b. This modification will reduce the total number of system calls, and thus increase throughput (each RPC requires at least two system calls to exchange messages).

- C.** Proposed modification: Place a lock around the reads and writes of balances in the server:

```
procedure transfer(account1, account2, amount) returns int
    acquire(balance-lock)
    balances[account1] = balances[account1] - amount
    balances[account2] = balances[account2] + amount
    release(balance-lock)
    return amount

procedure get_balance(account) returns int
    int bal
    acquire(balance-lock)
    bal = balances[account]
    release(balance-lock)
    return bal
```

Effect of modification: **Answer:** c. These locks eliminate a race that might occur if two clients sent transfer RPCs involving the same account.

- D. Proposed modification:** Run the account server on a separate machine from the client threads. Assume that RPCs between machines take long enough that this doesn't improve performance.

Effect of modification: **Answer:** a. Placing the client and server on separate computers might help prevent some client bugs from affecting the server, for example if the client allocates too much memory or consumes too much CPU time.

After running his server for a few days, Ben observes that sometimes clients (e.g., GUI instances) hang because RPC requests are never responded to. He suspects the problem is with the custom RPC sending and receiving code he added to the custom operating system he built for his banking application. His send/receive code is as follows:

```

structure rpcRequest
    string procedure // operation to perform in server
    string args // arguments
    string result

rpcRequest msgs[N] // array of up to N RPC requests that need to be processed
int numMsgs initially 0
lock bufferLock initially unlocked
condition rpcDone // a condition variable, as described in lecture

procedure rpc_send(rpcRequest m)
    m.result = null
    acquire(bufferLock)
    msgs[numMsgs] = m
    numMsgs = numMsgs + 1
    wait(rpcDone, bufferLock)
    release(bufferLock)
    return m.result

procedure rpc_handler()
    while (true) // repeat forever
        acquire(bufferLock)
        if (numMsgs > 0)
            m = msgs[numMsgs-1]
            m.result = execute m.procedure(m.args) in server
            notify(rpcDone)
            numMsgs = numMsgs-1
        release(bufferLock)

```

`rpc_handler` runs in a separate thread inside the operating system kernel, looking for RPC messages to dispatch. A client thread calls `rpc_send` to send an RPC request and wait for the server's response. There is a single instance of each of the variables `msgs`, `numMsgs`, `bufferLock`, and `rpcDone` inside the kernel.

**13. [10 points]:** Which of the following statements about Ben's RPC implementation are true?

- A. True / False** `rpc_handler()` will execute all RPCs as long as fewer than N RPC requests are outstanding at a time.

**Answer:** True.

- B. True / False** `rpc_send()` will correctly return results from all RPCs as long as fewer than N RPC requests are outstanding at a time.

**Answer:** False. The `notify()` for any of the RPCs will wake up all the waiting `rpc_send()`s, causing all but the intended one to incorrectly return null.

- C. True / False** `rpc_send()` will correctly return results from all RPCs as long as only one client has an outstanding RPC request at a time.

**Answer:** True.

- D. True / False** If only one call to `rpc_send()` is ever made, that call may wait forever because it may miss the `notify` from `rpc_handler()`.

**Answer:** False. As explained in lecture, calling `wait()` and `notify()` while holding the lock avoids lost notifies.

- E. True / False** The code is likely to deadlock because `rpc_send()` calls `wait()` while holding a lock.

**Answer:** False. As explained in lecture, `wait()` releases the lock.

## End of Quiz I Solutions