Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2010

Quiz I

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

Some questions are harder than others and some questions earn more points than others—you may want to skim all questions before starting.

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!

Write your name in the space below. Write your initials at the bottom of each page.

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

CIRCLE your recitation section number:

10:00	1. Lampson/Kushman	
11:00	2. Jones/Rieb	3. Rudolph/Kushman
12:00		4. Rudolph/Rieb
1:00	5. Gifford/Post	6. Jones/Spicer
2:00	7. Gifford/Spicer	8. Lampson/Post

Do not write in the boxes below

1-4 (xx/27)	5 (xx/12)	6-11 (xx/36)	12-13 (xx/25)	Total (xx/100)

Name:

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?

(Circle True or False for each choice.)

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.

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

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

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

(Circle True or False for each choice.)

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.

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.

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.

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

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?

(Circle True or False for each choice.)

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

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

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

Initials:

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

(Circle True or False for each choice.)

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

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

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.

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

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.

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.

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

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()`.

(Circle True or False for each choice.)

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

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

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.

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

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.

Initials:

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.

Initials:

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

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

Initials:

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

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

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

Initials:

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

Initials:

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 6), 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.

Initials:

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: _____

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:_____

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:_____

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:_____

Initials:

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.

Initials:

13. [10 points]: Which of the following statements about Ben's RPC implementation are true? **(Circle True or False for each choice.)**

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

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.

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.

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()`.

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

End of Quiz I

Please double check that you wrote your name on the front of the quiz,
and circled your recitation section number.

Initials:

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 execute kernel call is convenient but not essential for the shell to invoke a command.

Answer: True. The execute 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 4×10^{-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 millisconds, 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 BBBE). 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

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2012

Quiz I

There are 13 questions and 8 pages in this quiz booklet. Answer each question according to the instructions given. You have **50 minutes** to answer the questions.

Some questions are harder than others and some questions earn more points than others—you may want to skim all questions before starting.

For true/false and yes/no questions, you will receive 0 points for no answer, and negative points for an incorrect answer. We will round up the score for every *numbered* question to 0 if it's otherwise negative (i.e., you cannot get less than 0 on a numbered question).

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!

Write your name in the space below. Write your initials at the bottom of each page.

THIS IS AN OPEN BOOK, OPEN NOTES, OPEN LAPTOP QUIZ, BUT DON'T USE YOUR LAPTOP FOR COMMUNICATION WITH OTHERS.

Pad : can Google

CIRCLE your recitation section number:

10:00	1. Rudolph/Grusecki		
11:00	2. Rudolph/Grusecki	3. Abelson/Gokce	4. Katabi/Joshi
12:00		5. Abelson/Gokce	6. Katabi/Joshi
1:00	7. Shavit/Moll	8. Szolovits/Fang	
2:00	9. Shavit/Moll	10. Szolovits/Fang	

Do not write in the boxes below

1-5 (xx/27)	6-7 (xx/15)	8-10 (xx/32)	11-13 (xx/26)	Total (xx/100)
12	10 KP	22 OM	26 LL	70 NZ

Name: Michael AS Plasmeier

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.

B. True / False The parable of the watchmakers.

C. True / False Thermodynamic considerations about entropy.

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

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.

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.

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.

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.

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

Initials: MEP

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

(6)

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

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. oh yeah

C. True / False One of the advantages of multitasking is that it makes the system more responsive to user inputs. No - hanging

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`.

(0)

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

No - non standard locking algorithm would throw Eraser off

Paper: It does not do multiple locks

X

(6)

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]`?

No - its a seperate lock

(yeah) → Paper: again no multiple locks

Initials: MEP

II Complexity

H is made up of M, A, L

5

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). 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.

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

what feedback?

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

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

Web: wait to make connection

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

Web: conser in send, liberal in accept

1 **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.

Book: module - can be replaced
abstraction: see interfaces
layer: builds add. interfaces

III Names

5

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)

Verify

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.

does it

Web: Bash does keep history

C. Bash uses only Unix pathnames, not inodes.

Pretty sure

D. None of the above.

Initials: MEP

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.

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.

Initials: MEP

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.

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

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

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

Initials: MEP

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);
}
```

Initials: MEP

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 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:

10 Correction Mark clever q, straightforward

	Program A fails	Program B fails	Kernel fails	VM monitor fails
Bug in program A	X			
Bug in program B		X		
Bug in the kernel	X	X	X	
Bug in the VM monitor	X	X	X	X

correction virtual OS

TA: Ideal VMM

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.

8

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

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

End of Quiz I

Please double check that you wrote your name on the front of the quiz, and circled your recitation section number.

Initials: MEP

1st round 40 min left
2nd 25 min left
3rd 2 min
w/book goes fast w/ book!

Department of Electrical Engineering and Computer Science

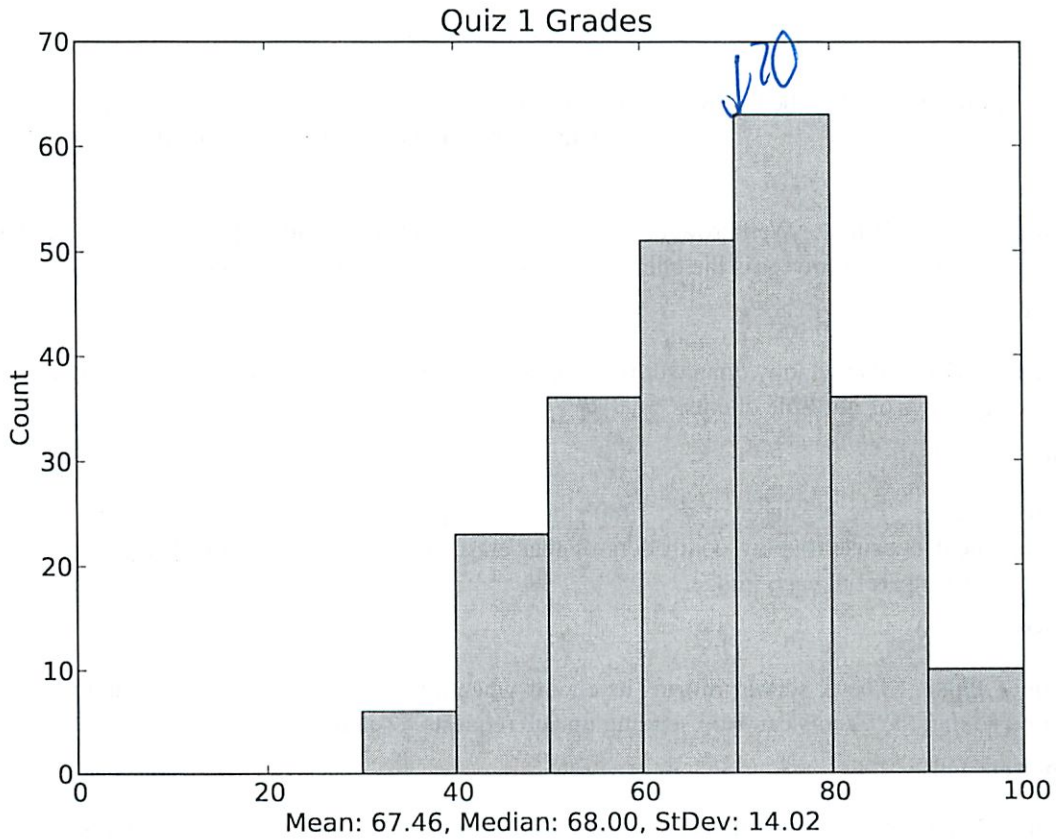
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

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.

Initials:

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)

Initials:

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.

Initials:

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.

Initials:

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: