6.033 Computer System Engineering

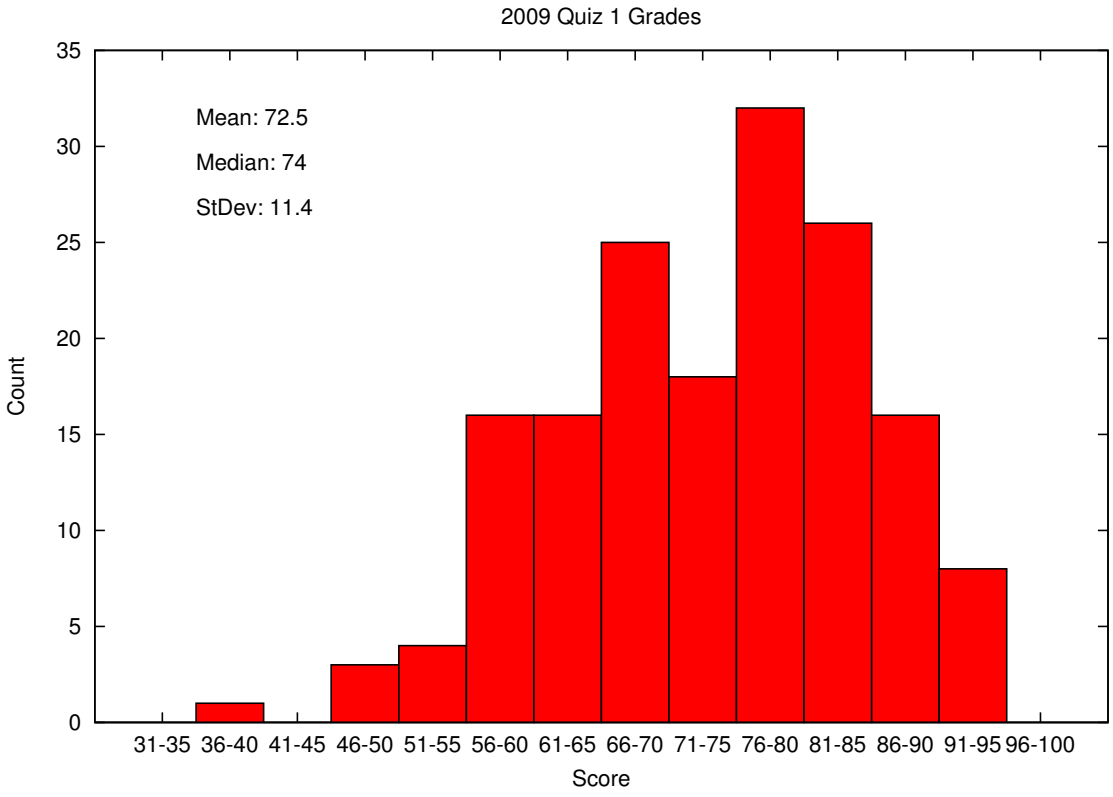
Spring 2009

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2009

Quiz I - Solutions



I Reading Questions

The following questions refer to Herbert Simon's paper, "The Architecture of Complexity" (reading #2).

1. [2 points]: Simon's notion of hierarchy organizes a collection of components by their
(Circle the BEST answer)

A. physical containment relationships

B. names and access patterns

C. strengths of interaction

Answer: C

2. [2 points]: Simon argues that systems naturally evolve in hierachical form because
(Circle the BEST answer)

A. hierarchies are inherently stable

B. hierarchies are easily described

C. a component in a structure of size N can be accessed in $logN$ steps

Answer: A

3. [6 points]: Based on the description of the X Window System in the 1986 paper by Scheifler and Gettys (reading #5), which of the following statements are true?
(Circle True or False for each choice.)

A. True / False X's client/server architecture ensures that one misbehaving client cannot interfere with other clients running on the same display.

Answer: False. For example, one client could use the window id of another client to write to its window.

B. True / False X's asynchronous protocol ensures that clients never have to wait for a network round-trip time for the server to respond to a request.

Answer: False. If the client needs some information from the server, such as a window id or the contents of a pixmap, it must wait for the server's reply.

C. True / False The X protocol always requires the server to send an exposure event to the client to redraw its window when an obscured window becomes visible.

Answer: False. The protocol allows the server to keep a copy of off-screen contents in memory as an optimization, using this to redraw exposed regions.

4. [6 points]: Based on the description of the UNIX file system in the 1974 paper by Ritchie and Thompson (reading #6), which of the following statements are true?

(Circle True or False for each choice.)

A. True / False The kernel does not allow users to create hard links to existing directories.

Answer: False. The superuser may create a link to an existing directory, for example while creating the `..` entry for a sub-directory.

Note: We also accepted True for this answer, since it was perhaps unclear whether "users" included the superuser.

B. True / False An application can ask the kernel to read any file by specifying its i-node number, as long as that i-node represents a file that the application has permissions to read.

Answer: False. No system call takes an i-number as argument.

C. True / False File names and i-node structures are stored within the data blocks of their containing directory.

Answer: False. A directory entry contains a file name and an i-number; the entry does not contain the i-node itself.

5. [6 points]: Based on the description of the MapReduce system in the 2004 paper by Dean and Ghemawat (reading #8), which of the following statements are true?

(Circle True or False for each choice.)

A. True / False If there are m map tasks, using more than m workers in the map phase may still improve performance beyond that achieved with m workers.

Answer: True. MapReduce will start multiple copies of the last few map or reduce tasks, to attempt to finish quickly despite slow or failed nodes.

B. True / False To achieve locality, map workers always execute on the same machine as the input data that they consume.

Answer: False. The master tries to place map workers with the input data, but if it can't it places them elsewhere (preferentially nearby in the network topology).

C. True / False Intermediate data passed between the map workers and reduce workers is stored in the Google File System (GFS).

Answer: False. Intermediate data is stored on the local disk of the map worker.

6. [8 points]: The following question refers to the Eraser system, by Savage et al. (reading #7).

Consider the following snippet of code, as part of a larger system:

```
Lock L1;
Lock L2;
int x;

function foo() {
  acquire(L1);
  print(x);
  release(L1);
}

function bar(int v) {
  acquire(L2);
  if (v == 0) {
    print(x);
  }
  release(L2);
}
```

The functions `foo()` and `bar()` are executed from separate threads, but Eraser never flags an error. Which of the following reasons might explain this?

(Circle ALL that apply)

A. `foo()` and `bar()` both execute, but never at the same time, so no race condition actually occurs

Answer: No. Eraser does not require race conditions to actually occur in order to flag them, since it spots errors by comparing the sets of locks that protect each variable at different points in the program.

B. `foo()` and `bar()` run concurrently, but `bar()` is always called with 1 as an argument

Answer: Yes. If execution of `bar()` never uses `x`, Eraser will not think that `L2` protects `x`.

C. Every time `foo()` or `bar()` is called, an additional lock `L3` is also held

Answer: Yes. Eraser will conclude that `L3` protects `x`.

D. The value of `x` is changed for the last time *before* either `foo()` or `bar()` are called for the first time.

Answer: Yes. Eraser will assume that `x` is a constant in this case, and doesn't need to be protected by locks.

II FaceFeeder

Inspired by Design Project 1, Ben BitDiddle decides to build a dataflow processing system for Facebook feeds. A *Facebook feed* is a stream of notifications, informing Facebook users of changes to a given friend's profile page.

Users upload programs, or *operators*, that are run by Ben's dataflow server. Operators can read from one or more feeds, and produce outputs which are themselves feeds. Users may subscribe to different feeds to receive alerts. Multiple users may subscribe to the same feed, and feed alerts are delivered asynchronously (e.g., via email.)

As an example of a FaceFeed application, one user might create an operator that combines their friends' "25 things you didn't know about me" lists into "a whole lot of things you didn't know about a whole lot of people" list. Another user might write an operator that takes in a stream of text and produces a graph showing the most common words in that stream. A third user might combine these together to produce a graph of the most common words used in his or her friends' "25 things you didn't know about me" list.

7. [8 points]: As a first approach, Ben decides to run all operators in the same address space, in a single process, with each operator running in a thread. His thread scheduler is pre-emptive, meaning that it can interrupt one thread and switch to another. Alyssa P. Hacker warns him that a single process is a bad idea, because running operators in the same process only provides soft modularity between them. Which of the following are problems that *could* arise in this design?

(Circle ALL that apply)

A. One operator might corrupt the memory of another operator.

Answer: Yes. All the software is running in the same address space, so operators can read and write each others' memory.

B. One operator might produce improperly formatted results, and send them to another operator, causing that operator to crash.

Answer: Yes. Invalid inputs can cause an operator to crash; putting operators in different threads does nothing to prevent this.

C. One operator might execute an illegal instruction, causing the process running the operators to crash.

Answer: Yes. An illegal instruction may cause a process to terminate.

D. One operator might never relinquish the CPU, preventing other operators from running.

Answer: No. Pre-emptive scheduling will periodically force such an operator to let other threads run.

8. [8 points]: Based on Alyssa's observation, Ben decides to switch to a new design where each operator runs in its own process, with its own address space, and with operators communicating only indirectly via the kernel. The OS handles scheduling of the processes, and is also pre-emptive. He claims this provides strong modularity and will prevent the problems Alyssa mentioned. Alyssa agrees that this will fix some of the problems with a single address space, but says it doesn't completely protect operators from each other. Which of the following are problems that *could* arise in this design?

(Circle ALL that apply)

A. One operator might corrupt the memory of another operator.

Answer: No. Each operator has its own address space, so it cannot directly read or write any other operator's memory.

B. One operator might produce improperly formatted results, and send them to another operator, causing that operator to crash.

Answer: Yes. Just running operators in different processes doesn't ensure that operators properly handle invalid inputs.

C. One operator might execute an illegal instruction, causing the processes running other operators to crash.

Answer: No. If an operator executes an illegal instruction, the only direct result will be that that operator's process will crash.

D. One operator might never relinquish the CPU, preventing other operators from running.

Answer: No. The operating system kernel will pre-emptively switch among the processes.

Ben decides to continue with his one-process-per-operator desgin. Because operators aren't running in the same address space, Ben needs to use a kernel structure to exchange data between them.

Ben thinks that users of feeds will often be interested in the most recent updates first. Because of this, he decides to design the system so that upstream operators first send their newest items to downstream operators. To achieve this, he uses a stack abstraction rather than a queue like the bounded buffer we studied in class.

Ben decides to add two new routines to the kernel, `put_stack` and `get_stack`, which add an item to a stack and receive an item from a stack, respectively. Adjacent operators in the data flow graph exchange data by having the upstream operator call `put_stack` and the downstream operator call `get_stack`.

His implementation of these routines is listed on the following page.

```
// add message to stack, blocking if stack is full
// stack size is N
// initially head = N
// buffer is a 0-indexed array of N message slots
// stack.lock is a lock variable associated with the stack
put_stack(stack, message):
  while true:
    if stack.head > 0:

      acquire(stack.lock)
      stack.head = stack.head - 1
      release(stack.lock)

      acquire(stack.lock)
      stack.buffer[stack.head] = message
      release(stack.lock)

      return
    else
      yield() //let another process run



//get next message from stack, blocking if stack is empty
get_stack(stack):
  while true:
    if stack.head < N:

      acquire(stack.lock)
      message = stack.buffer[stack.head]
      release(stack.lock)

      acquire(stack.lock)
      stack.head = stack.head + 1
      release(stack.lock)

      return message
    else
      yield() //let another process run
```

Notice that Ben's implementations acquires and releases `stack.lock` several times in each function. Ben claims this improves the performance of his implementation (versus an approach that acquires the lock and holds it for the duration of several operations).

Suppose that two operators, o_1 and o_2 are exchanging data via a stack s, and they perform the following sequence of operations. Here, time advances with the vertical axis, so if one operation appears above another operation, it finishes executing before the other operation begins. If two operations appear on the same line, it means they execute concurrently, and that arbitrary interleavings of their operations are possible (except, of course, that two operations cannot both be inside a critical section protected by `stack.lock`.)

o_1	o_2
put_stack(s,m_1)	
put_stack(s,m_2)	m = get_stack(s)
put_stack(s,m_3)	

9. [14 points]: Assuming `N = 4` and `head = 4` initially, after the above sequence of operations run, which of the following are possible states of the stack and the value of the `m` variable resulting from the call to `get_stack` in o_2?

(Circle ALL that apply)

A.
`m =` m_2

Stack
0: empty
1: empty
2: m_3
3: m_1

B.
`m =` m_1

Stack
0: empty
1: empty
2: m_3
3: m_2

C.
`m =` empty

Stack
0: empty
1: empty
2: m_3
3: m_1

D.
`m =` m_2

Stack
0: empty
1: empty
2: m_3
3: m_2

Answer: A, B, C. Scenario D is not possible because if `get_stack()` sees m_2, `put_stack(`m_2`)` must have already decremented `head`, so the increment in `put_stack(`m_2`)` will eliminate m_2 from the stack.

10. [6 points]: After Ben implements FaceFeed, his users create a dataflow program that consists of a long pipeline of many single-input, single-output operators. Ben runs this program in FaceFeed on a single core machine and finds that the performance isn't good enough. He decides to switch to a multi-core machine, but finds that, even though the operating system is properly scheduling his operators on different cores, he doesn't get much of a parallel speedup on this new machine. Which of the following are possible explanations for this lack of speedup?

(Circle ALL that apply)

A. One of the operators is much slower than the others, so its execution time dominates the total execution of the pipeline.

Answer: Yes.

B. All of the operators are about the same speed, so there is little opportunity for parallelism in the graph.

Answer: No. If they all take about the same time, all the cores will be kept busy, and the speedup will be roughly proportional to the number of cores.

C. One of the operators is much faster than the others, and those other operators dominate the execution time of the graph.

Answer: No. Perhaps one core will be mostly idle, but the other cores will contribute to parallel speedup.

III BeanBag.com

You're running BeanBag.com, a food delivery service. Your customers place orders over the Internet to your order server, using special client software that you supply to them. The server maintains, for each customer account, the list of items that the customer currently has on order. The order server communicates with a separate warehouse server that arranges for shipping of items.

Your order server supports three requests:

- `CHECK_ORDER(acct)`: given an account number, returns the list of items in that account's current order.
- `ADD_TO_ORDER(acct, item)`: add an item to the list of items an account has on order. Returns the item.
- `SHIP_ORDER(acct)`: directs the warehouse server to ship the account's current item list by truck to the customer. Returns the list of items that will be shipped.

Your order server is single threaded. The code for your server is given on the following page.

```
server():
  while true:
    request = RECEIVE_REQUEST()
    process_request(request)

process_request(request):
  if request.type == CHECK_ORDER:
    reply = process_check(request)
  else if request.type == ADD_TO_ORDER:
    reply = process_add(request)
  else if request.type == SHIP_ORDER:
    reply = process_ship(request)
  else
    reply = "error"
  SEND_REPLY(reply)

process_check(request):
  reply = orders[request.acct]
  return reply

process_add(request):
  orders[request.acct] = append(orders[request.acct], request.item)
  reply = "added " + request.item
  return reply

process_ship(request):
  otmp = orders[request.acct]
  orders[request.acct] = empty
  send an RPC to the warehouse server, asking for otmp to be shipped
  wait for reply from warehouse
  reply = "shipped " + otmp
  return reply
```

The order server has one CPU and keeps all its data in memory; it does not use a disk.

The initial version of the client software sends a request message to the order server when the customer clicks on the Check, Add, or Ship button, waits for a reply from the order server, and displays the reply to the customer. The client always waits for one operation to succeed before submitting the next one. Neither the client nor the order server does anything to deal with the fact that the network can lose messages.

11. [10 points]: The network between the client and order server turns out to be unreliable: it sometimes discards their messages. The network between the order server and the warehouse server is perfectly reliable. Which of the following problems might customers observe that could be caused by the network failing to deliver some messages between client and order server?

(Circle ALL that apply)

A. The client software could wait forever for a reply from the order server.

Answer: Yes. The network might drop a request message sent by the client to the order server.

B. The customer might click on the Ship button but BeanBag might never ship the order to the customer.

Answer: Yes. The network might drop the `SHIP_ORDER` message.

C. The customer might click on the Ship button, receive no reply from the order server, but still receive the items in the current order from BeanBag.

Answer: Yes. The order server might receive the `SHIP_ORDER` message, and the network might drop its reply to the client.

D. The customer might be shipped two of an item that he or she Added only one of.

Answer: No.

12. [8 points]: You modify the client software to re-send a request every five seconds until it gets a reply from the order server. You make no modifications to the servers. Which of the following problems might your modification cause?

(Circle ALL that apply)

A. The customer might be shipped two of an item that he or she Added only one of.

Answer: Yes. This will happen if the client sends an `ADD_TO_ORDER`, the order server receives it, the network drops the reply, the client re-sends the `ADD_TO_ORDER`, and the server also receives this second request.

B. The customer might Add some items to the order and get replies, then click on Ship, and get a reply with an empty item list.

Answer: Yes. This could happen if the order server receives the `SHIP_ORDER` but the network drops the reply. If the client's re-sent `SHIP_ORDER` arrives at the server, the customer's order list will be empty.

C. The customer might Add some items to the order and get replies, then click on Ship, get a reply with the correct item list, and then receive two distinct shipments, each with those items.

Answer: No. `process_ship()` clears the customer's order cart, so it is not possible to receive two shipments for the same items.

You decide that you need higher performance, so you convert the code to use a threading package, with pre-emptive scheduling. The order server starts up a new thread to serve each request. In order to avoid races your new code holds a lock when manipulating customer orders.

New or modified lines are in **bold**.

```
Lock lock;
server():
  while true:
    request = RECEIVE_REQUEST()
    create_thread(process_request, request)

process_request(request):
  if request.type == CHECK_ORDER:
    reply = process_check(request)
  else if request.type == ADD_TO_ORDER:
    reply = process_add(request)
  else if request.type == SHIP_ORDER:
    reply = process_ship(request)
  else
    reply = "error"
  SEND_REPLY(reply)
  exit_thread()

process_check(request):
  acquire(lock)
  reply = orders[request.acct]
  release(lock)
  return reply

process_add(request):
  acquire(lock)
  orders[request.acct] = append(orders[request.acct], request.item)
  reply = "added " + request.item
  release(lock)
  return reply

process_ship(request):
  acquire(lock)
  otmp = orders[request.acct]
  orders[request.acct] = empty
  release(lock)
  send an RPC to the warehouse server, asking for otmp to be shipped
  wait for reply from warehouse
  reply = "shipped " + otmp
  return reply
```

Your friend predicts that threading will not help performance, since your order server has only one CPU.

You measure the total throughput and per-request latency with the new and the old order server using a collection of machines running simulated clients. Each generates a sequence of `CHECK_ORDER`, `ADD_TO_ORDER`, and `SHIP_ORDER` requests, issuing a new one as soon as the server replies to the previous request. Each client machine generates requests for a different account. You measure throughput in total requests served per second. You should assume that it takes zero time to switch between threads, that the time to acquire a lock takes no time beyond waiting for the current holder (if any) to release it, and that function calls and returns take zero time.

13. [10 points]: It turns out your friend is not correct. Which of the following are true about the throughput of the threaded server, compared to the original server?

(Circle ALL that apply)

A. The average per-request latency is lower.

Answer: Yes. See the answer to D.

B. Simultaneous requests in the server have to wait for each other to release the lock, leading to lower throughput.

Answer: No. While it's true that simultaneous requests have to wait for each other, that was true of the non-threaded version as well.

C. A thread for one client can be in `process_check()` while a thread for another client is in `process_add()`, leading to higher total throughput.

Answer: No. The lock prevents any significant overlap between different operations, and in any case there is only one CPU so throughput of these compute-bound operations cannot be higher.

D. A thread for one client can be in `process_check()` while a thread for another client is in `process_ship()`, leading to higher total throughput.

Answer: Yes. `process_ship()` may spend a long time waiting for the warehouse server, but not using CPU and not holding the lock. Other operations can execute during this time.

At your friend's insistence you replace the order server with a shared-memory multiprocessor, and you measure its performance as above. You find that performance has not increased significantly beyond your single-processor deployment.

14. [6 points]: You hope to increase performance still further on your new multiprocessor. Which of the following would have the greatest positive effect on performance, while preserving correctness?

(Circle the BEST answer)

A. Have a separate lock for each request, so that, for example, `process_check()` calls `acquire(check_lock)` and `process_add()` calls `acquire(add_lock)`.

Answer: This isn't the best answer because it would not preserve correctness in cases where multiple clients send requests for the same customer account.

B. Have a separate lock for each account, so that, for example, `process_check()` calls `acquire(locks[request.acct])`.

Answer: This is the best answer. The per-account locks preserve correctness, and allow operations for different accounts to proceed concurrently.

C. Delete all the acquires and releases.

Answer: This doesn't preserve correctness where two operations are running concurrently for a given customer account.

D. Delete the acquires and releases, and add an `acquire(lock)` at the start of `process_request()`, and a `release(lock)` just before the `exit_thread()`.

Answer: This doesn't improve performance, since only one operation can execute at a time.

End of Quiz I