DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033—Computer System Engineering Handout 13—March 12, 1995

Quiz #1 Solutions

180 students took the quiz.
The median grade (half got better, half got worse) was 80 points.

Score Frequency:

41-45 ***** (5)
46-50 *** (3)
51-55 ******* (7)
56-60 ***** (5)
61-65 ************** (14)
66-70 ************ (12)
71-75 ***************** (17)
76-80 ****************************** (30)
81-85 ***************************** (29)
86-90 ****************** (18)
91-95 ************************** (26)
96-100 ************** (14)

For Official Use Only

Problem	Your Score	Maximum Score
1		18
2		20
3		18
4		20
5		24
Total		100

Name: ______________________________

1. *(18 points)* For each of the following statements, circle whether it is True or False, and briefly explain. An answer without an explanation will receive no credit. And if you give a good enough explanation you may be able to get credit even if you say False to a point we thought was True, or vice-versa. *But keep it brief!*

1a. The paper "On building systems that will Fail", written by Corbató, was on creating planned catastrophes. TRUE/FALSE

FALSE. The paper was on building complex systems, anticipating that failures will occur, and how to cope with problems.

1b. If one created a graph of personal friendships, one would have a hierarchy. TRUE/FALSE

FALSE. No, mutual friends would make it more of a "web" or a "complete graph".

1c. One of the mistakes that the Therac-25 programmers made was to not reuse software from earlier radiation therapy machines. TRUE/FALSE

FALSE. They DID reuse software from the previous machines, but removed hardware interlocks.

1d. Firewalls completely prevent any bug in a server from affecting clients. TRUE/FALSE

FALSE. An incorrect answer from the server would still affect the client. Firewalls attempt to keep such effects to a minimum.

1e. Modularity and abstraction control propagation of effects.TRUE/FALSE

TRUE. They keep bugs within a module/abstraction layer, and thus control propogation.

1f. Variable delay is an intrinsic problem of isochronous networks.TRUE/FALSE

FALSE. Isochronous networks have equal delay by definition. (ISO=equal, chronous=time)

Name: ______________________________

2. Ben Bitdittle is designing a virtual memory system, VM, for his new microkernel system Pico. In particular, he is interested in what page size to use for his VM system and whether VM should be implemented inside the Pico kernel or outside.

2a. *(5 points)* Ben's computer has a disk, which can be accessed in 20 msec. Once accessed Ben's computer can blast data to the disk at 10 Mbyte/s. Give a formula for the time to write an *n*-byte page to disk, in milliseconds.

The formula is $20 + n * 10^{-4}$ msec

Other formulas which define M and K as 2^{20} and 2^{10} respectively also received full credit.

2b. *(5 points)*Using the formula compute how long it takes to write a 1-Kbyte page and how long it takes to write an 10-Kbyte page to disk.

The time for a 1-Kbyte page = 20.1 msec

The time for a 10-Kbyte page = 21.0 msec

Other answers that were based on the formula given in 2a received full credit.

2c. *(5 points)*Given the results of b) Ben decides to use a 10-Kbyte page size.Why might Ben be better off choosing a smaller page size?

Ben might better off using smaller page size, because with smaller pages more pages fit in memory. If multiple applications are using only a small parts of a page, then fewer page faults will be incurred and less memory will be wasted than with using big pages.

2d. *(5 points)*Suppose that crossing the kernel boundary and delivering a message takes about 100 cycles on Ben's 200-Mhz computer. Should Ben handle page faults in the kernel or in a trusted server?

The cost for doing the LRPC from the kernel to a trusted server is neglible compared to the time for dealing with a page fault. Therefore, the VM system might as well be implemented in a trusted server to improve modularity of the system.

Name: ____________________________________

3. Two threads, A and B, execute a function with the following code (using the syntax from Birrell's thread paper), but always at different times (i.e., only one of the threads is in the function at a given time).

```
lock mutex_a do
    lock mutex_b do
    ...
    end
end

lock mutex_b do
    lock mutex_a do
    ...
    end
end
```

3a. *(9 points)*Assuming that no other code in the program ever acquires more than one lock at a time, can the program deadlock? (if yes, give an example, if not, argue why not)

No, because there is no possibility that two or more threads each holds a lock at the same time time that another threads needs. Call the code fragments in the problem "our_func()". If our_func() gets one or both locks before some other thread, say thread C, it will run to completion, because, by assumption, thread C can only lock at most one lock at a time (and thus cannot have a lock needed by "our_func()"). If thread C gets the lock first, our_func() may be delayed, but C will finish and eventually release the lock (we know that C proceeds because it could only be waiting for one lock, which in this case we assumed it has gotten). Either way, some thread is able to run to the point of releasing its lock(s). Threads A and B cannot conflict, since only one is in our_func() at once.

3b. *(9 points)*Now, assume that the two threads *can* be in the code fragment above at the same time, can the program deadlock? (if yes, give an example, if not, argue why not)

Yes. Deadlock can occur if thread A acquires mutex_a at the beginning of the function and thread B acquires mutex_b in the middle. Both will hold a lock needed by the other thread, so neither will make forward progress. Neither lock will ever be released, so the threads are deadlocked.

Name: ____________________________________

4. Threads in a new multithreaded WWW browser periodically query a nearby WWW server to retrieve documents. On average, a browser's thread performs a query every *N* instructions. Each request to the server incurs an average latency of *T* milliseconds before the answer is received.

4a. *(5 points)*For *N* = 2,000 instructions and *T* = 1 msec, what is the smallest number of such threads that would be required to keep a single 100 MIPS processor (i.e., executing one hundred million instructions every second) 100% busy? Assume the context switch between threads is instantaneous, and the scheduler is optimal.

*50 threads [= (1msec * 100 MIPS) / 2000 instructions] Adding one more (51) was acceptable -- most who did this added a bonus thread for a scheduling role, and mentioned that. .*

4b. *(5 points)*But context switches are not instantaneous. Assume that a context switch takes *C* instructions to perform. Recompute the answer to part a) for *C* = 500 instructions.

*40 threads [= (1msec * 100 MIPS) / (2000 + 500 instructions) Again, a bonus thread (41) for overhead was acceptable if qualified. Assuming that a context switch requires 2C=1000 total instructions (ie., to get in, and get out) gives you 33.33333 threads. A few people did this, and it's a reasonable alternate answer.*

4c. *(5 points)*What property of the application threads might cause the answers of parts a) and b) to be incorrect? That is, why might more threads be required to keep the processor running the browser busy?

Within a WWW browser, application threads will tend to be interdependent (perhaps blocking on disk, network, or display), and the application i/o will tend to be bursty. Any "average" latencies are thus subject to wide variance, and the browsing process will afford many idle moments.

4d. *(5 points)*What property of the actual computer system might make the answers of a) and b) gross overestimates?

A nice bullet list of general system vicissitudes was fine; e.g.,

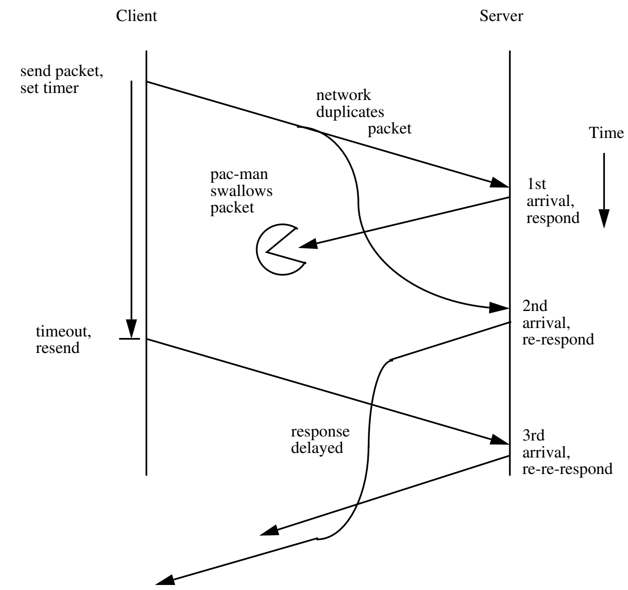
- *paging*
- *multiprocessing overhead, sub-optimal scheduling*
- *i/o bursts in audio visual subsystems*
- *network delays (percolation through internet, load on remote servers)*

Name: ____________________________________

5. Birrell and Nelson's RPC, as described in their paper "Implementing Remote Procedure Calls", guarantees *at-most-once* semantics (*i.e.*, the RPC is carried out at most one time, but possibly none at all).

5a. *(8 points)*Give a scenario (*i.e.*, a sequence of events) in which the same request may arrive twice at the server?

The timing diagram below shows several things that could produce this effect: the network may duplicate the request or the network may swallow or delay the server's response to the request. In either of the latter two cases, the client will timeout and resend the request, causing it to arrive twice at the server.



Name: ____________________________________

5b. *(8 points)*Ben gets really excited about RPC and implements an RPC package himself. Like Birrell and Nelson, he starts a thread for each new request at the server. The thread performs the operaton specified in the request, sends a reply, and terminates. After measuring the RPC performance, Ben discovers that the creation of the thread and context switching to it is a major part of the cost of performing an RPC. After thinking about it, Ben comes up with a brilliant optimization: instead of creating a new thread for each new request, Ben's optimized RPC package executes the RPC on the current running thread. To his surprise the server deadlocks with this new optimization. What is happening? (Hint: think about locks that the current thread might be holding.)

To "execute the RPC on the current running thread" must mean that the system pre-empts the currently running thread from whatever it was doing and puts it to work on the newly arrived RPC request. If the pre-empted thread happened to be holding a lock at the instant of pre-emption, and the new RPC request requires it to try to seize that same lock, the thread will deadlock on itself. More complicated scenarios involving other threads are also possible, but they all involve the new RPC request somehow waiting for release of the lock held by the pre-empted thread.

5c. *(8 points)*Frustrated with optimizing his RPC package, Ben decides to use a brute force solution: he buys a much faster network. The latency for a sending packet from A to B on the new network is twice as low. Ben measures the performance of small RPCs (*i.e.,* each RPC contains only a couple of bytes) on the new network. To his surprise the performance is hardly better. What might be the reason that his RPCs are not twice as fast?

The latency introduced by RPC comprises network transit times, time in the client stub that marshalls arguments, time in the server stub that unpacks the arguments, time in the network layers that add and remove headers, and time in context switches in both the client and the server. So making the network faster will reduce only one, possibly unimportant, part of the total RPC latency. (The called procedure at the far end also doesn't get any faster when the network speeds up, but presumably Ben is smart enough to realize that time in the application program isn't part of the RPC-induced latency.)