Department of Electrical Engineering and Computer Science

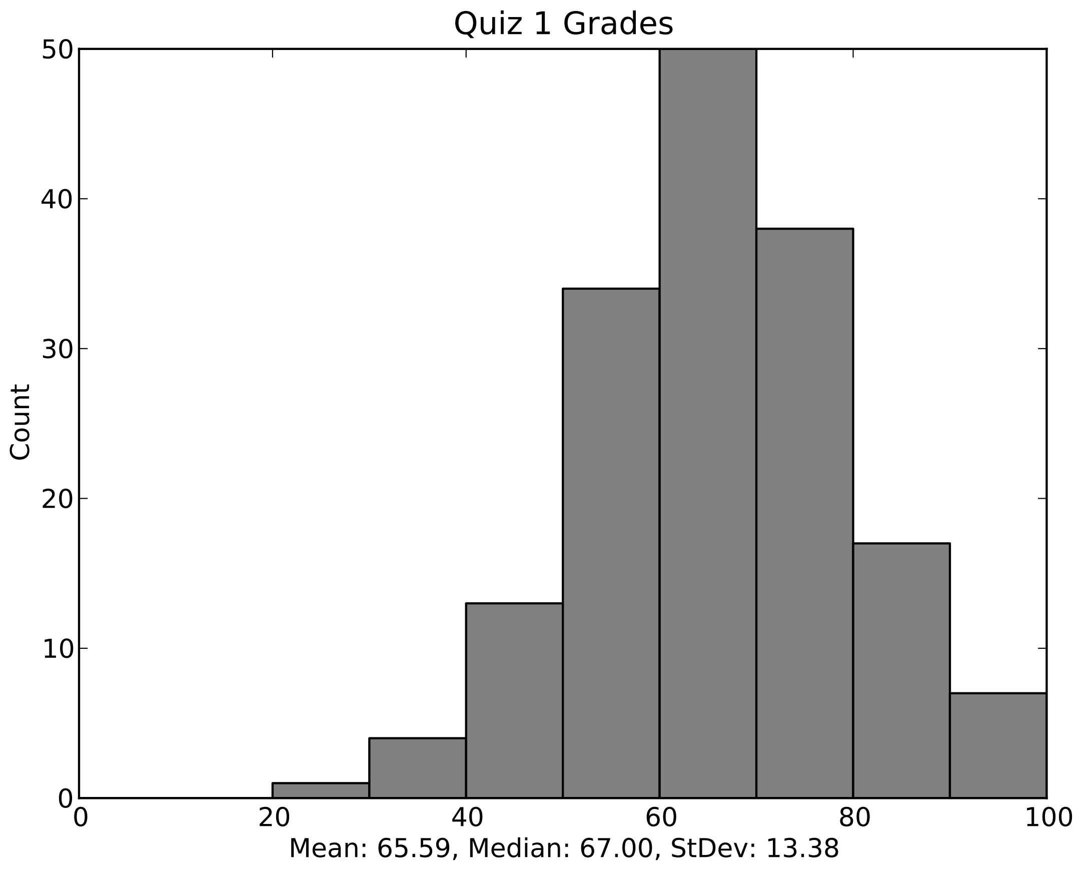
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2011

Quiz I Solutions

There are 10 questions and 12 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. [10 points]: Which of the following are true statements about the Eraser system, according to the paper by Savage *et al*?

(Circle True or False for each choice.)

A. True / False Eraser always detects and reports bugs that would cause a program to deadlock (where all of the programs threads wait concurrently trying to acquire locks).

Answer: False. Eraser does not detect deadlocks.

B. True / False If a program contains a race condition bug, Eraser needs to observe an interleaving of program threads where that bug causes a race before Eraser can report that the bug exists.

Answer: False. Eraser will report a race if the lock set is empty, even if it does not observe an interleaving that leads to a bug.

C. True / False Because Eraser modifies a binary program with its own instrumentation routines, Eraser would be an ideal tool to identify locking errors in a program which uses custom memory allocation routines, has its own special-purpose synchronization plan that does not use standard acquire and release functions, and for which no source code is available.

Answer: False. Eraser needs to track calls to memory allocation and locking routines in order to know when to update and reset the state of each memory location. Programmers can add annotations to denote non-standard operations, but this approach will not work when no source code is available.

D. True / False For each memory location in the *Exclusive* state, Eraser sets the candidate set for that location to all the locks held by the single thread that wrote to that memory location.

Answer: False. For *Exclusive* values, Eraser stores the thread identifier of the exclusive user in its shadow area instead of a candidate lock set. Eraser tracks locks held while accessing *Shared* and *Shared-Modified* values.

2. [10 points]: Which of the following statements about VMware's x86 virtual machine monitor are true, according to the paper by Agesen *et al*?

(Circle True or False for each choice.)

A. True / False The x86 architecture originally contained instructions that were non-virtualizable using trap-and-emulate virtualization.

Answer: True. The *popf* instruction in x86 was originally non-virtualizable using trap-and-emulate.

B. True / False Shadow page tables store the mappings from guest physical to host physical memory addresses.

Answer: False. Shadow page tables store the mappings from guest *virtual* memory to host physical memory addresses.

C. True / False VMware's VMM prevents the guest from accessing the memory used for the VMM's internal state by unmapping the corresponding entries in the guest's page table.

Answer: False. As stated in the paper, permission bits in page tables do not allow for execute-only mappings, which are needed for the Translation Cache. The VMM's own data must be accessible to translated code but not the original guest instructions, even though both run at the same privilege level. Thus, *memory segmentation* is used to prevent the guest from accessing the VMM's internal state.

D. True / False VMware's binary translation-based VMM translates guest kernel-mode instructions but not guest user-mode instructions.

Answer: True. There is no need to translate any guest user-mode instructions. For instance, a popf instruction in a program running on the guest OS doesn't need to be translated. An aside: some students mentioned that the VMM uses IDENT translations for guest user-mode code. If, hypothetically, the VMM did translate guest user-mode code, then this is still wrong, because jump instructions need to be modified for translated code.

E. True / False A monitor using hardware-assisted virtualization (Intel's VT-x or AMD-V) is always faster than one using binary translation.

Answer: False. Adaptive BT can perform some optimizations that hardware-assisted virtualization cannot. The relative performance of either approach to virtualization varies greatly based on the workload involved.

3. [10 points]: For v7 Unix processes as described in the paper by Ritchie and Thompson, which of the following statements are true?

(Circle True or False for each choice.)

A. True / False In Unix v7's time sharing, all processes receive an equal fraction of CPU time.

Answer: False. Unix makes no guarantees about the scheduling policy between processes, or its fairness.

B. True / False In Unix v7, a parent process and its child process can have exactly the same memory contents but execute different instructions.

Answer: True. After calling `fork`, the child process and parent process have exactly the same memory contents, but receive different return values from the fork system call (in a CPU register). If the system stores return values on the stack, it's still possible to have the child and process have identical memory contents: the program's code could first branch based on the return value, and then zero out the return value on the stack. The processes will now only differ in the program counter's value.

C. True / False In Unix v7, two child processes can communicate via a pipe only if the pipe is created by a common ancestor (where a process can be considered to be its own ancestor).

Answer: True. The only way to gain access to a pipe in Unix v7 is to create it, or to inherit a file descriptor for the pipe from a parent process.

D. True / False Two processes in Unix v7 can communicate via shared memory, regardless of whether they have a common ancestor or not.

Answer: False. Unix v7 had no shared memory.

II Client/server

4. [10 points]:

Which of these statements about client/server organization are true or false?

(Circle True or False for each choice.)

A. True / False In a client-server system, the client and the server always run on different machines.

Answer: False. The point of client-server is that there is a hard modularity boundary between the client and the server.

B. True / False A client-server system can be faster than a system using the same code to do the actual work, but in which everything runs in the same process, in spite of the added cost of communication between the client and the server.

Answer: True. If the client and the server run on different processors, there are twice as many cycles available to do the work.

C. True / False In a client-server system that uses remote procedure call, the only important difference from a similar system that uses local procedure call is that the remote calls may be substantially slower.

Answer: False. Because the client and the server can fail independently, the client will sometimes be uncertain about whether a call has actually happened or not.

D. True / False A server that wants to provide at-least-once semantics to a client must store a table of previous responses.

Answer: False. Providing at-least-once semantics can be achieved by having the client retransmit requests; a table of previous responses is required for at-most-once semantics.

E. True / False X Windows is not an example of a client-server system because the so-called X server runs on the client's machine.

Answer: False. It's confusing that the X server runs on the client machine, but it does play the role of a server: a remote agent that other programs can call on for service—in this case, painting the screen and delivering input from the user.

III Concurrency

Ben Bitdiddle wants to implement a concurrent application. Instead of doing the sensible thing and holding a lock while manipulating shared data, Ben decides to be clever. Ben's code, shown below, runs in two separate threads on a computer with two CPUs; no other threads are running on the system. There are three shared global variables, initialized before the threads start: `i` is initialized to zero, `x` is initialized to a three element array of zeros, and `x_lock` is in the unlocked state.

```
Thread 1:                              Thread 2:
   while i < 3 do:                        while i < 3 do:
      while i % 2 == 0 do:                   while i % 2 == 1 do:
         # (while i is even)                    # (while i is odd)
         yield()                                yield()
      acquire(x_lock)                        acquire(x_lock)
      x[i] = 1                               x[i] = 2
      i = i + 1                              i = i + 1
      release(x_lock)                        release(x_lock)
```

5. [10 points]: After both threads reach the end of their code segments, which of the following are possible values for the contents of the x[] array?

(Circle True or False for each choice.)

A. True / False x=2,1,1
Answer: False

B. True / False x=2,2,2
Answer: False

C. True / False x=2,1,2
Answer: True

D. True / False x=2,2,1
Answer: False

Initially Thread 2 is the only one that can proceed to acquire the lock, and will write a 2 to $x[0]$. As soon as it stores a 1 into i, Thread 1 can pass through to try to aquire the lock, but it cannot succeed until Thread 2 releases the lock. Thread 2 then proceeds to spin in the inner while loop until Thread 1 changes i again. Thread 1 writes a 1 into $x[1]$, and updates i, and then Thread 2 can procees to update $x[2]$.

Thread 1 may write to $x[3]$, off the end of the array, if it checks for $i < 3$ before Thread 2 sets i to 3 the final time through the loop, but this was not listed among the choices to consider.

Now suppose that thread 1 instead runs the following variation below. Thread 2 runs the same code as before (shown below for convenience), and global variables are initialized in the same way as before.

```
Thread 1:
   a = 0
   while i < 3 do:
      while i % 2 == 0 do:
         # (while i is even)
         yield()
         a = i
      acquire(x_lock)
      x[a] = 1
      i = a + 1
      release(x_lock)
```

```
Thread 2:
   while i < 3 do:
      while i % 2 == 1 do:
         # (while i is odd)
         yield()
      acquire(x_lock)
      x[i] = 2
      i = i + 1
      release(x_lock)
```

6. [10 points]: After both threads reach the end of their code segments, which of the following are possible values for the contents of the x[] array?

(Circle True or False for each choice.)

A. True / False x=2,1,1
Answer: True. Suppose that the threads proceed as in the previous section, until the point where Thread 2 writes a 2 into $x[2]$ and sets i to 3. At this point, Thread 1 can break out of the inner while loop. When it does so, a can hold either 2 or 3, depending on whether Thread 2 updated i between when Thread 1 executes `a = i` and checking $i < 3$ or at some other time. If a holds 2, Thread 1 will overwrite $x[2]$ with a 1, resulting in a final value of 2,1,1.

B. True / False x=2,2,2
Answer: False. No interleaving is possible where Thread 2 writes to $x[1]$.

C. True / False x=2,1,2
Answer: True. Same interleaving as with the initial code segments

D. True / False x=2,2,1
Answer: False. No interleaving is possible where Thread 2 writes to $x[1]$.

E. True / False one or both threads might not reach the end of the code segments.
Answer: True. There are two possible ways the threads might not reach the end of their code segments:

- Thread 1 sets a to i each time it spins waiting for i to change. When Thread 2 sets $x[0]$ to 2 and i to 1, Thread 1 will break out of the inner while loop with a set to either 0 or 1. If a is 0, Thread 1 will acquire the lock, set $x[0]$ to 1, and set i to 1 (as it already is). At this point, Thread 1 loops forever in the body of the outer while loop, and thread 2 loops forever in the body of the inner while loop.
- Thread 1 could write to $x[3]$ the final time through the outer while loop, as described earlier. If $x[3]$ is an invalid memory location, Thread 1 could cause a fault and be killed on this memory access before reaching the end of the code segment.

7. [10 points]: Having come to his senses, Ben wants to repair his code so that it doesn't take him nearly this long to understand its behavior. Which of the following options would make it easier to reason about the concurrency in Ben's code, while not introducing new errors?

(Circle True or False for each choice.)

A. True / False Remove acquire() and release() to avoid reasoning about locks.
Answer: False. This would make the code harder to reason about, not easier.

B. True / False Move acquire() and release() around the entire outer `while` loop.
Answer: False. This would introduce a new bug: the threads would never complete their code segments.

C. True / False Acquire and release the lock around every single access to `i`.
Answer: False. This would make the code harder to reason about, not simpler.

D. True / False Acquire the lock before the outer `while` loop and release it only around the `yield` call.
Answer: True. This approach would serialize the threads around the calls to yield, which makes reasoning easier, and does not introduce any new bugs, since each thread can still make progress

IV Operating systems

In many modern Unix systems the kernel code and a user program share a single address space on Intel x86 processors, yet the kernel is protected from malicious programs. The x86 has a bit `U/K` that specifies whether the processor is in user mode or kernel mode. Each x86 page table entry contains the physical page address, along with several bits, including:

- A present bit (P), indicating whether the virtual page should be accessible.
- A writable bit (W), indicating whether writes to the virtual page are allowed.
- A user bit (U), indicating whether this page should be accessible from both kernel and user modes (if the U bit is set), or whether it should be accessible only from kernel mode (if the U bit is not set).

8. [10 points]: Which of the following statements must be true to protect the kernel's data from either reads or writes by user programs, while allowing the program to read and write its own data, and allowing the kernel to operate?

(Circle True or False for each choice.)

A. True / False The kernel must set the U and P bit in the page table entries for text (i.e., code) pages of the user program.

Answer: True. If this isn't done, the user program will not be able to execute its own code.

B. True / False The kernel must clear the U bits and set the P bits in the page table entries for text (i.e., code) pages of the kernel program.

Answer: True. The P bit needs to be set so that the kernel will be able to execute its own code. But the U bit must not be set, so that the user process can not read the kernel's code.

C. True / False The kernel must program the hardware to set the U/K bit to K when entering the kernel.

Answer: True. If the U/K bit is not set to K, the kernel will not be able to operate because it cannot access its code and data, as these pages do not have the U bit set.

D. True / False The kernel must program the hardware so that the user program can enter only at carefully-chosen addresses.

Answer: True. User code must be allowed to enter kernel execution at certain well-defined entry points (gates), such as the system call or trap handlers. If it could enter at any arbitrary address, it would violate the isolation between the user process and kernel.

E. True / False The kernel should not execute load instructions for user addresses.

Answer: False. The kernel is permitted to access user memory. For example, it needs to do so when the user process makes a `write` system call, in order to read the data that needs to be written to a file.

F. True / False The kernel should not execute store instructions to user addresses.

Answer: False. Same as above. Similarly, when the user process makes a `read` system call, the output is written to the user process's memory.

9. [10 points]: Which of the following statements relating to the complexity of Linux, as compared to the v7 version of Unix system as described in the paper by Ritchie and Thompson are true?

(Circle True or False for each choice.)

A. True / False Linux has many more developers than v7 Unix.

Answer: True. Linux has thousands of developers. v7 Unix was developed by a small group at Bell Labs.

B. True / False Linux has many more lines of code than v7 Unix.

Answer: True. v7 Unix was around 10,000 lines of code; the Linux kernel contains millions of lines of code.

C. True / False Only Linux uses client/server to enforce modularity within kernel (and not v7 Unix).

Answer: False. Neither use a client/server approach. Both use a monolithic kernel design rather than a microkernel.

D. True / False Only Linux uses virtual memory to isolate user processes (and not v7 Unix).

Answer: False. Both v7 Unix and Linux use virtual memory to enforce modularity between processes.

E. True / False Only the Linux kernel supports multiple processes (and not v7 Unix).

Answer: False. Both v7 Unix and Linux support multiple processes.

V Virtual machines

A guest OS kernel executes a single process, using the page table shown below on the left, stored in guest physical page 4. The virtual machine monitor uses the table shown below on the right to translate guest physical pages to host physical pages.

Virt Page	Phys Page	P	W	U
0	5	1	1	1
1	9	0	1	1
2	11	1	0	1
3	4	1	1	0
4	7	1	1	0

Guest phys page	Host phys page
4	2
5	6
6	4
7	3
11	8

10. [10 points]: Fill in the two shadow page tables below that the VMM should use when running the VM. The VMM loads the *user* shadow PT when running VM code with the guest U/K bit set to user, and the VMM loads the *kernel* shadow PT when running VM code with the guest U/K bit set to kernel. The VMM performs no binary rewriting for any guest instructions that read or write memory. You can use a dash (—) to indicate values that don't matter. (Do not rely on demand-filling these shadow page tables whenever possible.)

User shadow PT, stored in host phys. page 12

Virt Page	Phys Page	P	W	U
0	6	**1**	**1**	**1**
1	**—**	**0**	**—**	**—**
2	8	**1**	**0**	**1**
3	**—**	**0**	**—**	**—**
4	**—**	**0**	**—**	**—**

Kernel shadow PT, stored in host phys. page 13

Virt Page	Phys Page	P	W	U
0	6	**1**	**1**	**1**
1	**—**	**0**	**—**	**—**
2	8	**1**	**0**	**1**
3	**2**	**1**	**0**	**1**
4	**3**	**1**	**1**	**1**

Answer: The P bit for virtual page 1 has to be 0 in both shadow PTs, because the page is not present in the guest's PT. The W bit for virtual page 2 has to be 0 in both shadow PTs, because the page is not writable in the guest's PT. The U bits have to be set for all pages that are accessible to the VM, because the guest VM runs with the physical U/K bit set to user, regardless of the guest U/K bit value. Virtual page 3 maps to guest physical page 4, which maps to host physical page 2. Virtual page 4 maps to guest physical page 7, which maps to host physical page 3. Virtual pages 3 and 4 should have their P bits set to 0 in the user shadow PT, because these pages should only be accessible to the guest kernel. The W bit for virtual page 3 should be 0 (in the kernel's shadow PT) to trace writes to that page, since that page contains the guest's PT.