Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Operating System Engineering: Fall 2003

Quiz II Solutions

All problems are open-ended questions. In order to receive credit you must answer the question as precisely as possible. You have 80 minutes to answer this quiz.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

1 (xx/15)	2 (xx/25)	3 (xx/20)	4 (xx/25)	5 (xx/15)	Total (xx/100)

Name:

I Control-C and the shell

1. [15 points]: In UNIX when a user is running a program from the shell, the user can terminate the program by typing ctrl-C. How would you change the 6.828 kernel and its shell to support this feature? Be careful, make sure when a user types the name of the program and hits ctrl-C before the program runs that the right thing happens (i.e., don't kill the shell itself or the file system). (Sketch an implementation and define "what the right thing" is.)

There are many possible solutions; here are three:

- Add a system call to the kernel that allows the shell to indicate at any given point what environment, if any, should be killed when the user types ctrl-C. Modify the console code in the kernel appropriately so that the kernel intercepts ctrl-C (instead of passing it on as a control character to whatever environment is reading the console), and kills the indicated environment. With this solution, environments can never prevent themselves from getting killed: thus, CTRL-C acts like a "hard" or "uncatchable" Unix signal.
- Modify the kernel's console code so that the shell can intercept ctrl-C input characters even when other, "normal" input characters may be going to some other (e.g., child) environment that happens to be reading from the console. Then upon receiving this special control character from the kernel in the console input stream, the shell can decide which of its child environments should be killed. With this solution, environments once again cannot prevent themselves from getting killed, but only the *immediate* child environments of the shell can be killed this way: a (killable) child environment can easily create another, "unkillable" environment simply by forking or spawning a child of its own (a grandchild of the shell).
- Add some form of asynchronous event handling support to the kernel's console code so that environments can asynchronously accept and handle ctrl-C input characters themselves. For example, each environment might be able to register a user-mode "ctrl-C handler," along the lines of the current user-mode "page fault handler." Whenever the user hits ctrl-C, the kernel scans through the list of environments and, for each environment that has registered a ctrl-C handler (and perhaps has set a flag indicating that ctrl-C "signals" should be accepted), the kernel sets up the environment's registers and user-mode exception stack to cause the environment's registered ctrl-C handler to be invoked. Each environment can then perform whatever ctrl-C handling it deems appropriate, such as by killing itself immediately, by killing itself after "gracefully" shutting down (such as after closing files and `sync`ing outstanding data to disk), or even by ignoring the ctrl-C signal entirely. With this solution, CTRL-C is effectively a "soft" signal, which is not guaranteed to work on environments that may be either accidentally or maliciously ignoring it. (This is like the Unix behavior for ctrl-C, since Unix processes can intercept the `SIGINT` signal.)

II CPU scheduling

2. [10 points]: `primespipe` spawns many environments until it has generated a prime greater than some specified number. If you modify your shell for the 6.828 kernel to run `primespipe` in the background and then type `ls` at your shell, it can happen that the output of `ls` won't show before `primespipe` completes. Explain why.

(Keep it brief)

The kernel's scheduler uses a round-robin scheduling policy, always choosing the next runnable environment *in environment ID-space* whenever the current environment yields or runs out of its timeslice. Since newly created environments are assigned environment IDs in ascending sequence, `primespipe` creates a pathological unfairness situation. The "root" `primespipe` environment creates the first child environment, fills the pipe to that child with numbers, then yields the processor in order to wait for more space in the pipe. Since the first child environment typically has the environment ID immediately following the parent's, the child is the next to run. The child in turn creates its own grandchild environment, then reads numbers from its parent's pipe buffer and stuffs the ones that "pass" into the grandchild's pipe buffer. Upon running out of numbers, the child yields, causing the grandchild to run next, and so on. Even if the shell managed to start the `ls` environment before the `primespipe` cascade takes over, `ls` requires a few back-and-forth IPCs with the file system, and so it is very unlikely to get much of anything done before the entire `primespipe` cascade has completed. The key point is that this unfairness is created by the interaction of the *round-robin scheduling policy* with the kernel's "first-free" environment ID assignment.

3. [5 points]: Describe a solution to the problem described in the previous question.

(Keep it brief; no pseudocode required)

- A quick-and-dirty solution that would make this unfairness problem go away but would probably leave many others: Simply make the round-robin scheduling policy look for runnable environments in *descending* rather than ascending order of environment ID. Then `ls` and the file server will each typically get one chance to run for each new `primespipe` instance.
- A more general solution: maintain an explicit *queue of runnable environments* from which the scheduler picks the next environment to run instead of just scanning by environment ID. As long as newly-runnable (including newly-created) environments are always added to the *tail* of the queue and the scheduler always picks the environment at the *head* as the next environment to run, environment creation cannot be used as a way to hog the CPU.
- Implement a Unix-style interactive scheduling policy in which processes that have not run for a while or have not used much CPU time receive an elevated priority.
- Implement a hierarchical or group-based scheduling policy where an environment's children receive a share of the CPU time originally allotted to its parent, so that the entire group of `primespipe` environments collectively get only as much CPU time as the `ls` or file server environments do individually.

Name:

Let's assume we modify the 6.828 kernel to build a high-performance Web server. We add a driver for ethernet card in in the Web server environment. Interrupts from the card are directly delivered to the Web server's environment, and the interrupt handler executes on a separate stack in the Web server's environment (in a similar style as the 6.828 kernel sends page faults to environments.) The Web server's interrupt handler runs with interrupts disabled, processes the received interrupt completely (including any TCP/IP and HTTP processing), returns all resources associated with handling the interrupt, and re-enables interrupts. Web pages in the cache are served straight from the interrupt handler. CGI scripts and pages not in the cache are handled by the Web server on the main stack (not in the interrupt handler).

4. [10 points]: Does this implementation suffer from receive livelock? If so, sketch a sequence of events that will result in receive livelock. If not, explain why. (If you need to make any more assumptions about the architecture of the system, be sure to state them explicitly.)

Yes, the implementation does suffer from receive livelock. Suppose HTTP requests arrive at a sufficient rate that by the time the web server's interrupt handler is finished with the interrupt processing for one request, the Ethernet card has already received another request, causing the interrupt to be triggered again immediately as soon as the web server re-enables interrupts. Since cached pages are served straight from the interrupt handler, the web server will be able to continue serving them. Pages that miss in the cache and dynamic CGI pages must be handled by the web server *outside* of the interrupt handler, however. As long as the web server keeps getting a continuous load of interrupts it will *never* get out of its network interrupt handler and get around to processing any of these cache misses or CGI requests. Thus, the web server is livelocked, making no forward progress at all on many of the requests it is supposed to be serving.

III File systems and reliability

Modern Unixes support the `rename(char *from, char *to)` system call, which causes the link named "from" to be renamed as "to". Unix v6 does not have a system call for rename; instead, rename is an application that makes use of the `link` and `unlink` system calls.

5. [5 points]: Give an implementation of rename using the `link` and `unlink` system calls, and briefly justify your implementation.

```
int rename(char *from, char *to) {
     /* 1 */ unlink(to);
     /* 2 */ link(from, to);
     /* 3 */ unlink(from);
}
```

Some editors use `rename` to make a new version of a file visible to user atomically. For example, an editor may copy "x.c" to "#x.c", make all changes to "#x.c", and when the user hits save, the editor calls `sync()` followed by `rename("#x.c", "x.c")`.

6. [5 points]: What are the possible outcomes of running `rename("#x.c", "x.c")` if the computer fails during the rename library call? Assume that both "#x.c" and "x.c" exist in the same directory but in different directory blocks before the call to `rename`.

Because the effects of the calls in our rename implementation are not persistent unless the corresponding block writes go out to disk, it is *not* correct simply to consider the failure happening before or after each call. Instead, we need to think about the possible block writes that can happen before the failure.

The rename implementation above causes three block writes, all possibly delayed (written with `bdwrite`). First, the directory entry for "to" is removed from its block. Second, the directory entry for "to" is added back to its block, but with the i-number for "from". Third, the directory entry for "from" is removed from its block.

The possible outcomes are:

- Nothing done: no changes written to disk yet.
- The first write went to disk: "from" remains as before, but "to" doesn't exist at all. (Looks like rename failed just after line 1 above.)
- The first and second writes went to disk: "from" remains as before, but now "to" points at the same file as "from". (Looks like rename failed just after line 2 above.)
- The first, second, and third writes went to disk: "from" is gone, and "to" points at the new file. (Looks like rename succeeded.)
- The first and third writes went to disk, *but not the second*: both "from" and "to" are gone! (Cannot be explained by examination of the rename implementation!)

Name:

Modern BSD UNIX implements `rename` as system call. The pseudocode for `rename` is as follows (assuming "to" and "from" are in different directory blocks):

```
int rename (char *from, char *to) {
   update dir block for ''to'' to point to ''from'''s inode // write block
   update dir block for ''from'' to free entry // write block
}
```

BSD's Fast File System (FFS) performs the two writes in `rename` synchronously (i.e., using `bwrite`).

7. [5 points]: What are the possible outcomes of running `rename("#x.c", "x.c")` if the computer fails during the rename system call? Assume that both "#x.c" and "x.c" exist in the same directory but in different directory blocks before the call to `rename`.

- Nothing done: state remains as-is before the `rename`.
- Only first ("to") update done: "from" and "to" both refer to the same file.
- Everything done: `rename` completed successfully.

Because of the atomic change of the "to" entry in this implementation, we never get into a state where "to" does not point to anything. Because the first write must go out before the second, we cannot lose the new copy of the file. Many common Unix applications today depend on this level of `rename` atomicity in the file system in order to ensure reasonable robustness against failures.

Assume the same scenario but now using FFS with soft updates.

8. [5 points]: How does using FFS with soft updates change this scenario?

In terms of robustness, soft updates do not change the file system's semantics at all: exactly the same situations are possible as above for FFS with synchronous writes, because the writes must go out in the same order. The difference is in *performance*: with soft updates, the file system does not have to perform all meta-data writes synchronously, and therefore can achieve much greater performance benefits from caching without compromising robustness.

Just to be precise, it's worthy of note that soft updates *can* potentially change the file system's failure semantics in ways that are theoretically *observable* to applications; just not in ways that create *file system metadata inconsistencies*. For example, suppose an application creates a file A somewhere in the file system, then creates a file B *somewhere else* in the file system that happens to involve completely unrelated directory and inode blocks. If a failure occurs during or shortly after these files are created, with soft updates the application could (on system restart) observe that file B was successfully created but A wasn't, because the soft updates algorithm imposes no ordering dependency between them. With synchronous writes, however the application would never observe that B as having been successfully created unless A had also been successfully created.

Name:

IV Virtual machines

A virtual machine monitor can run a guest OS without requiring changes to the guest OS. To do so, virtual machine monitors must virtualize the instruction set of a processor. Consider implementing a monitor for the x86 architecture that runs directly on the physical hardware. This monitor must virtualize the x86 processor. Unfortunately, virtualizing the x86 instruction set is a challenge.

9. [5 points]: Using the instruction "mov %cs, %ax", give a code fragment that shows how a guest operating system could tell the difference between whether it is running in a virtual machine or directly on the processor, if the virtual machine monitor is not sufficiently careful about how it virtualizes the x86 architecture.

(List a sequence of x86 assembly instructions)

```
movw    %cs, %ax
andw    $3, %ax
jnz     running_on_vm
```

The bottom two bits of the CS register on the x86 represent the processor's current privilege level, 0 meaning kernel mode and 3 meaning user mode. The guest operating system can therefore tell whether it is running in user or kernel mode simply by testing those two bits of CS. Since the "mov %cs, %ax" instruction is *not* privileged, the virtual machine monitor has no easy way to trap that instruction and emulate the *correct* behavior—i.e., to return a CS register value with 0 in the low two bits, indicating that the guest OS is running in "virtualized" kernel mode, even though the guest OS code is *actually* running in user mode as far as the physical hardware is concerned.

10. [5 points]: Describe a solution for how to virtualize the instruction "mov %cs, %ax" correctly. What changes do you need to make to the monitor?

(Give a brief description; no pseudocode required)

The virtual machine monitor first scans through all code in the guest OS before allowing it to run, replacing each non-virtualizable instruction such as the one above with a one-byte "`int $3`" instruction, and remembering the original instruction replaced at each point in the code using a table maintained elsewhere by the monitor. When the guest OS hits one of these "`int $3`" instructions, the virtual machine monitor intercepts the interrupt, scans its table of replaced instructions for that EIP address, emulates the correct behavior for the replaced instruction, and increments EIP past the space reserved for it.

Name:

The monitor might emulate a load of CR3 as follows:

```
// addr is a physical address
void
emulate_lcr3(thiscpu, addr)
{
  Pte *fakepdir;

  thiscpu->cr3 = addr;
  fakepdir = lookup(addr, oldcr3cache);
  if (!fakepdir) {
    fakedir = page_alloc();
    store(oldcr3cache, addr, fakedir);
    // CODE MISSING:
    // May wish to scan through supplied page directory to see if
    // we have to fix up anything in particular.
    // Exact settings will depend on how we want to handle
    // problem cases.
  }
  asm("movl fakepdir,%cr3");
  // Must make sure our page fault handler is in sync with what we do here.
}
```

11. [15 points]: Describe a solution for handling a guest OS that executes instructions stored in its data segment (e.g., `user_icode` in the 6.828 kernel).

The virtual machine monitor must use some protection mechanism to ensure that:

- Any attempt by the guest OS to "jump into" and run newly loaded or modified code will trap into the monitor, giving the monitor a chance to scan the code and replace non-virtualizable instructions as described for question 10.
- *After* a page of physical memory containing code has been scanned and modified, any attempts by the guest OS to *read or write* that page of memory (treating it once again as "data" instead of jumping into it as "code") will once again trap into the monitor, allowing the monitor to undo its modifications or simulate the correct read or write to the code page as if the page contained the guest OS's original, unmodified code.

There are different ways of achieving the above protection characteristics on the x86, but none of them are simple due to such practical complications as the lack of support for "execute-only" page mappings in the x86 architecture. Clever use of segment registers, privilege levels, and/or more intelligent guest OS code analysis and rewriting can help.

Name:

V Feedback

Since 6.828 is a new subject, we would appreciate receiving some feedback on how we are doing so that we can make corrections next year. (Any answer, except no answer, will receive full credit!)

12. [1 points]: Did you learn anything in 6.828, on a scale of 0 (nothing) to 10 (more than any other class)?

13. [1 points]: What was the best aspect of 6.828?

14. [1 points]: What was the worst aspect of 6.828?

15. [2 points]: If there is one thing that you would like to see changed in 6.828, what would it be?

16. [2 points]: How should the lab be changed?

17. [1 points]: How useful was the v6 case study, 0 (bad) to 10 (good)?

18. [1 points]: How useful were the papers, 0 (bad) to 10 (good)?

19. [2 points]: Which paper(s) should we definitely delete?

20. [2 points]: Which paper(s) should we definitely keep?

Name:

21. [1 points]: Rank TAs on a scale of 0 (bad) to 10 (good)?

22. [1 points]: Rank the professor on a scale of 0 (bad) to 10 (good)?

End of Quiz II

Name: