Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2007

Quiz II Solutions

The average was 80.

I Virtual Machines

The x86 SIDT and LIDT instructions read and write the interrupt descriptor table register (IDTR). Recall that the IDTR specifies the address of the interrupt descriptor table, which controls where the CPU jumps when a fault or hardware device interrupt occurs. LIDT loads the IDTR (modifies the register) while SIDT just reads the register so the executing code can examine its value. LIDT is privileged: it can only be executed at CPL 0. SIDT is not a privileged instruction: it can be executed at CPL 3.

1. [6 points]: A user program in JOS or xv6 could use SIDT to examine the IDTR. Is this a bad thing? Why or why not?

Answer: It is not bad. The IDTR value is a constant in JOS and xv6, so there's little to be learned from it.

Suppose you want to make a virtual machine monitor (VMM) for the x86, along the same lines as the Disco paper. Remember that, for efficiency, Disco directly executes guest operating system instructions when possible. It executes them with ordinary user privilege, so that the guest kernel traps to the VMM when the guest tries to execute privileged MIPS instructions.

2. [6 points]: The x86 will trap to the VMM if the guest kernel tries to use LIDT to load an address a into the IDTR. Can the VMM just itself execute LIDT a and return to the guest? Why or why not? You can assume that a is a valid linear address that really does refer to the base of the guest kernel's desired IDT, and that the IDT entries all refer to valid code locations within the guest kernel.

Answer: No. Hardware interrupts and traps should enter the VMM, not the guest kernel, so that the VMM can decide whether the guest kernel should see them. Thus the VMM must supply its own IDT.

3. [6 points]: It turns out that SIDT causes problems for an x86 VMM. Explain why.

Answer: SIDT lets the guest see the real contents of the IDTR, which point to the VMM's IDT. The guest is expecting to see a pointer to the guest's IDT. The fact that SIDT does not trap when executed at CPL 3 means the VMM doesn't get a chance to intervene.

II File System Naming

Look at xv6's `_namei()` in fs.c on sheet 43. The job of `_namei()` is to turn a pathname, such as `/usr/rtm/quiz.tex`, into a pointer to an in-core i-node. The while() loop processes each element of the pathname in turn, descending the directory tree. `_namei()` locks each directory while it is looking at it; for example, it locks the inode of `/` while it is searching for `usr`. It unlocks the directory at end of the loop. These are not spin locks; instead these locks set the inode's `I_BUSY` flag, and waiting processes sleep instead of spinning.

Suppose process P1 has called `_namei()` for `/usr/rtm/quiz.tex` and is about to execute `ip = next` at line 4379 after finding `rtm` in `/usr`. Thus the old `ip` refers to `/usr`, and `next` refers to `rtm`. `/usr/rtm` has only one file in it: `quiz.tex`. A clock interrupt forces a switch to a different process, which happens to be a root shell that runs

```
rm /usr/rtm/quiz.tex
rmdir /usr/rtm
mkdir /usr/zzz
echo hello > /usr/zzz/quiz.tex
```

All of these commands complete successfully; there are no errors. Then P1's `_namei()` continues.

4. [6 points]: Could the file system allocate `/usr/zzz` the same disk inode (i.e. same i-number) that `/usr/rtm` was using before the `rmdir`? Why or why not?

Answer: No. `_namei()` still has a pointer to the in-core inode of `/usr/rtm`, and has not called `iput()`, so the `ref` reference count is non-zero. Thus `iput()` will not free the inode when `rmdir` finishes. Thus the `mkdir` won't allocate the same inode.

5. [6 points]: What will P1's `_namei()` do after it continues? Will it find the original quiz.tex? Is there a possibility it could instead find the new quiz.tex? Or will something else happen? Explain why.

Answer: `_namei()` will return 0. It will search the original inode of `/usr/rtm` and, since `rm` deleted `/usr/rtm/quiz.tex`, `_namei()` won't find any file.

III File System Atomicity and Recovery

Suppose a program on xv6 creates and writes a file:

```
fd = open("file", O_CREATE | O_WRONLY);
write(fd, "hello", 5);
close(fd);
```

The following is an abbreviated list of the disk writes that would result:

```
write file inode
write directory entry
write block free bitmap
write data block
write file inode
```

These disk writes are synchronous: xv6 waits for the disk to finish each before it starts the next. xv6 uses synchronous writes to enforce a particular order, much as described in Section 2.1 of the Soft Updates paper.

You can assume that individual disk writes are atomic with respect to crashes: if there's a crash or power failure during a write, the write either completes successfully, or does nothing. You should assume that there is no file system checker (i.e. there is nothing like UNIX's `fsck`).

The above sequence writes the file inode twice, which sounds inefficient. Suppose xv6 delayed the first file inode update, and combined it with the second, to save one write:

```
write directory entry
write block free bitmap
write data block
write file inode
```

> **6. [6 points]:** Suppose there's a power failure during the above sequence. What inconsistency could exist in the on-disk file system after the failure as a result of eliminating the first inode write? How might that inconsistency eventually cause the file system to return the wrong data for a `read()`?

Answer: The directory will contain the i-number of an inode that is marked free (type zero). A future creation of a different file might use the same inode, causing the first file to be the same as the second.

Suppose xv6 were modified to omit updating the block free bitmap right away, and rather only update the bitmap every 30 seconds in order to increase efficiency. The write order would then be:

```
write file inode
write directory entry
write data block
write file inode
```

7. [6 points]: Suppose there's a power failure just after the above sequence. What inconsistency could exist in the on-disk file system after the failure as a result of eliminating the free bitmap write? How might that inconsistency eventually cause the file system to return the wrong data for a `read()`?

Answer: The file will contain a block that is on the free list. A future file write might allocate the same block and change its content, thus incorrectly modifying the first file.

IV XFI Memory Protection

The XFI paper describes a system that ensures that a module can only read, write, and execute inside its own memory. XFI would be useful for Linux loadable kernel modules, which the kernel loads from a file into kernel memory and executes with full privilege (CPL 0). Without something like XFI, a buggy or malicious loadable kernel module could wreck the kernel by writing into its memory. XFI could be used to make sure that a loadable kernel module only used its own memory, and did not read or write any of the kernel's data structures.

Below is a copy of Figure 2 of the XFI paper. XFI has inserted code that checks that the target of the `call` really is the start of a function inside the module.

```
   EAX := 0x12345677
   EAX := EAX + 1
   if Mem[EBX - 4] != EAX, goto CFIERR
   call EBX
   ...
   0x12345678
L: push EBP
```

8. [6 points]: Why is it absolutely necessary for XFI to check that the module is jumping to the start of a function? Why can't XFI just enforce that the module is jumping somewhere inside its own memory – i.e. just prevent the module from jumping into the kernel?

Answer: When XFI checks that the module's code doesn't contain instructions that write outside the module's memory, it only looks at instructions that could be reached from the beginnings of functions. If the module could jump anywhere in its memory, it could jump into the middle of a legitimate instruction, thus executing an instruction that XFI did not check. The module could also jump directly to a load or store instruction, bypassing XFI's checks.

You could view XFI as providing a software version of the memory protection provided by the x86's pagetables. You could even imagine using XFI instead of the x86's pagetables in order to protect the kernel against user processes. That is, you might be able to re-design JOS to run user environments with CPL zero (full privilege), but use XFI to make sure environments only read and wrote their own memory. Environments would use function calls into the kernel instead of INT system calls; XFI is powerful enough to check that environments only call approved system call functions in the kernel.

9. [6 points]: Explain two different ways that JOS uses x86 pagetables to do things that XFI (as described in the paper) does not support.

Answer: Copy on write fork and direct transfer of file pages from the file server. XFI doesn't provide the level of indirection in addressing that pagetables provide.

V L3, Microkernels, and fast IPC

The L3 paper (Improving IPC by Kernel Design) describes a number of techniques to achieve fast inter-process communication in a micro-kernel on an x86.

10. [6 points]: Explain why the L3 paper worries much more about avoiding TLB misses than about avoiding data cache misses.

Answer: For both the TLB and the CPU data cache, you will get expensive misses if you touch too many different areas in memory. Thus much of what the paper is doing is reducing the number of distinct areas of memory that the IPC code uses. The paper concentrates on the TLB because it has fewer entries than the data cache (32 vs 512 on the Intel i486).

An IPC sending process in L3 waits until the receiving process is ready to receive (i.e. in the receive system call) before sending the message (see the start of Section 5.1).

11. [6 points]: Explain why L3's high performance depends on the receiver being ready.

Answer: If the receiver is ready, the sending process can directly enter the receiver's user space, carrying along the IPC message in the registers. This direct entry also eliminates the expense of scheduling (putting the sender to sleep and calling the scheduler).

VI Bugs as Deviant Behavior

Recall Section 3.1 of the Bugs as Deviant Behavior paper, which aims to detect possible dereferences of null pointers by looking for inconsistencies between the programmer's pointer checks and dereferences. The paper's checker flags both the examples in 3.1 as bugs.

Assume there is no inter-procedural information – the checker only looks at the current function.

Consider the following code:

```
fn1(int x, int *p){
  if(x){
    printf("p -> %d\n", *p);
  }
  if(p == 0)
    panic("p is null");
}
```

12. [6 points]: Will a checker like the one in 3.1 flag the above code as having inconsistent use of pointers? Why or why not?

Answer: Yes. There is a path through `fn1()` that dereferences `p` and then checks if `p` is null. `p`'s belief set will be "not null" after the dereference, which contradicts the "null" and "not null" implied by the check.

Now consider this code:

```
fn2(int x, int *p){
  if(x){
    printf("p -> %d\n", *p);
  } else {
    if(p == 0)
      panic("p is null");
  }
}
```

13. [6 points]: Will a checker like the one in 3.1 flag the above code as having inconsistent use of pointers? Why or why not?

Answer: No. There is no path through the code that has both a derefence and a check.

VII OKWS

14. [6 points]: Section 6.1 (4) of the OKWS paper mentions that okld does no message parsing. Why is that important to the security of the system?

Answer: Since attackers cannot arrange to send data to okld (even indirectly), they won't have the opportunity to exploit bugs that might exist in okld's message handling code. okld runs as root (unlike the rest of OKWS), so it's particularly important that it not be exposed to attacks.

VIII Scalable Synchronization

Suppose you are running xv6 on a machine with 10 CPUs and shared memory. After reading the Scalable Synchronization paper you're worried that xv6 might have bad spin-lock performance, and you wonder if it might benefit from the techniques in the paper.

15. [6 points]: Would a more scalable spin-lock be likely to improve the performance of `pipewrite()` on sheet 52? Why or why not?

Answer: Probably not. In most cases pipes have just a single reader and a single writer, so even with many CPUs there is not likely to be much contention for any given pipe's lock.

Here's a copy of the paper's Algorithm 5, the MCS list-based queuing lock:

```
procedure acquire_lock(L : ^lock, I : ^qnode)
  I->next := nil
  predecessor : ^qnode := fetch_and_store(L, I)
  if predecessor != nil
    I->locked := true
    predecessor->next := I
    repeat while I->locked // <-- XXX

procedure release_lock(L : ^lock, I : ^qnode)
  if I->next = nil
    if compare_and_swap(L, I, nil)
      return
    repeat while I->next = nil
  I->next->locked := false
```

Suppose 10 CPUs try to acquire the same MCS lock at the same time. Each releases the lock immediately after it successfully acquires it. The amount of time it takes all 10 CPUs to finish acquiring and releasing the lock is mostly determined by the number of cache misses. A CPU will incur a cache miss if it reads a location that was last written on a different CPU, or if it writes a location that is currently in the cache of a different CPU.

You can assume that each CPU's `qnode` structure starts out locally cached on just that CPU. You can assume that variables and data that are only used by one CPU incur no cache misses. You can assume that `predecessor` is non-nil for 9 of the 10 calls to `acquire_lock()`.

16. [6 points]: How many cache misses will be incurred by the line marked XXX, summed over all the CPUs, in the process of all 10 CPUs each acquiring and releasing the lock? Explain your answer.

Answer: Nine. Each waiting CPU will see one cache miss on its `I->locked`, when its predecessor sets `locked` to false at the end of `release()`.

End of Quiz