Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Fall 2008

Quiz II Solutions

I File System Consistency

Ben is writing software that stores data in an xsyncfs file system (see the paper *Rethink the Sync* by Nightingale *et al*). He's nervous about crash recovery. To help himself test, he adds a new system call to his operating system called crash(), which powers off his computer without doing anything else. He also writes a simple "crash" command which calls his system call. His general test strategy is to perform some operations, call crash(), restart the computer and let the file system finish recovering, and then observe what files and data are on the disk.

1. **[7 points]:** First, Ben types the following commands to his shell:

```
% echo hello > foo
% crash
```

After the restart, is Ben guaranteed to see a file foo with contents "hello"? Why or why not?

Answer: Yes. xsyncfs guarantees that any file system writes that causally precede user output will be stable on disk before the user sees the output. echo's write() to foo causally precedes the second "%" prompt in the above output: echo exit()s after it write()s, and the shell wait()s for echo's exit() before it displays the prompt. Thus xsyncfs will have written echo's output to foo before it allows the second % to be displayed.

2. [7 points]: Now Ben runs the following UNIX program which he believes is equivalent to the above commands:

```
pid = fork();
if(pid == 0){
  fd = creat("foo", 0666); // create a file foo
  write(fd, "hello\n", 6);
  close(fd);
  exit(0);
}
wait(&status); // wait for pid to exit
crash();
```

After the restart, is Ben guaranteed to see "hello" in foo? Why or why not?

Answer: No. There is no user-visible output, so xsyncfs does not guarantee to send file writes to the disk. Much of the point of xsyncfs is to avoid immediately writing the disk in cases like this, to increase performance.

3. [7 points]: What would the answers be for the xv6 file system? Why?

Answer: If you used xv6, you'd see "hello" in foo after the restart. That's because the xv6 file system is synchronous: all file system operations (such as creat() and write()) wait for their modifications to be written to disk before they return.

II Virtualization

The paper *A Comparison of Software and Hardware Techniques for x86 Virtualization* states that one of the goals of virtualization is "Fidelity: Software on the VMM executes identically to its execution on hardware, barring timing effects."

The paper also says, at the end of Section 3.2, that the software virtualization technique outlined in Section 3 does not translate guest user-mode code. That is, when the guest operating system executes instructions that would cause a switch to CPL=3 on real hardware, the VMM stops translating code and allows the original guest instructions to execute directly on the hardware.

> **4. [8 points]:** Explain a way that a carefully constructed guest kernel and/or user-mode program could exploit direct execution of user-mode code to discover whether it was executing on a virtual machine. You should assume the system outlined in Section 3, running on a 32-bit x86.

Answer: One possibility is for the guest kernel to transfer to user space with interrupts turned off (IF clear in EFLAGS). The VMM will leave interrupts turned on in the real hardware, since it needs to see the real hardware interrupts; IF will only be clear in the shadow EFLAGS. User-mode code can use pushf to examine the real hardware EFLAGS. If interrupts are enabled in this situation, the software is running on a virtual machine; if interrupts are disabled, the software is running on a real machine.

III Fault Tolerance

Ben is impressed with the paper *Fault Tolerance Under UNIX*, by Borg *et al.* He builds a system that is identical – with one exception. In order to get better performance, his hardware has two busses, and every machine has a connection to both busses. When a machine needs to send a message, it selects one of the two busses at random, and sends the message on that bus.

> **5. [8 points]:** Explain why this change will cause serious problems to the correctness of the system. Give a specific example of something that will likely go wrong.

Answer: Suppose two different machines, M_1 and M_2, send messages to M_3 at the same time, but use different busses. M_3 might see the messages in one order, and M_3's backup might see them in the other order. If M_3 were to crash, its backup would restore its state incorrectly, since it would replay those two messages in a different order than M_3 saw them.

IV Bugs as Deviant Behavior

Have a look at Figure 3 of the paper *Bugs as Deviant Behavior* by Engler *et al*. Suppose first function in Figure 3 looked like:

```
ssize_t proc_mpc_write(struct file *file,
                       const char *buff){
  retval = parse_qos(buff, incoming);
}
```

That is, suppose the first seven lines of the original function were deleted.

> **6. [8 points]:** Would the checker described in Section 7.1 emit an error for this modified function? Why or why not?

Answer: No. The checker looks for pointers that the code both dereferences and treats as suspect (by passing to a "paranoid" function). There are no such pointers in this function.

It's possible that the checker's comparison of functions that are assigned to the same function pointer would catch the error even in this modified code, if some other abstractly related function passes the buff argument to a paranoid function.

V Livelock

Answer this question in the context of the paper *Eliminating Receive Livelock in an Interrupt-driven Kernel* by Mogul *et al.*

> **7. [8 points]:** Figure 6-3 shows that performance with polling and no quota is quite poor under high input load, but polling with a quota performs much better. Explain what happens without a quota that results in almost zero performance under high input load, and how a quota helps solve this problem.

Answer: Without a quota, the device driver spends 100% of the CPU time reading packets from the device and queuing them in a software output queue. This leaves no CPU time to give the queued packets to the output device hardware, so all packets (after the first queue's worth) are dropped due to overflow of the software output queue. With a quota, the polling system alternates between reading input packets and sending packets to the output device.

VI KeyKOS

Answer this question in the context of the paper *The KeyKOS Nanokernel Architecture* by Bomberger *et al*.

8. [4 points]: Suppose a machine running KeyNIX loses power while creating a new file in a directory. After the system is powered back up and executes to a quiescent state (e.g. so that any recovery code is executed), what are the possible states that the file system could be in? Circle all that apply (-1 point for each wrong answer).

A. The new file is allocated but the directory does not contain the new file.

B. The new file is not allocated but the directory contains a reference to the new (unallocated) file.

C. The new file is allocated and the directory contains a reference to the new file.

D. The new file is not allocated and the directory does not contain a reference to the new file.

Answer: **C** and **D**.

9. [7 points]: Most file system implementations worry a great deal about what happens after a crash, but the KeyNIX file system has no explicit file system recovery code. Explain what strategy KeyNIX and KeyKOS use to recover from crashes such as the one above. In particular, would the KeyNIX file system recover to a consistent state if a new file was created but not added to its parent directory by the time the machine was powered off (and if so, how)?

Answer: KeyKOS periodically checkpoints the entire state of the system to disk, including process memory and the register state of each process. If there's a crash, KeyKOS loads the most recent checkpoint into memory and continues executing from that point. The KeyNIX file system is implemented as a set of processes; these processes only write in-memory representations of files and directories, and do not directly write the disk. If there were to be a crash during the creation of a file, two scenarios are possible, both of which lead to a consistent state after recovery. The last checkpoint might have been taken early enough that the file creation hadn't started; in this case the recovered system will start out with neither file allocated nor directory modified (though perhaps whatever program eventually tried to create the file will re-execute to that point and create it again). Or the last checkpoint might have been taken after the start of creating the file, in which case the file-creation code will continue running after the recovery and create both the file and the directory entry.

VII HiStar

Answer this question in the context of the paper *Making Information Flow Explicit in HiStar* by Zeldovich *et al*.

Recall that KeyNIX maintains a global table mapping process IDs to capabilities (keys) that allow sending a message to that process's Unix keeper. When one process running on top of KeyNIX wants to send a signal to a different process, the sender's Unix keeper must check that the sender is allowed to send a signal to the recipient process (e.g. that both processes belong to the same user), before sending a "kill" message to the recipient's Unix keeper.

HiStar implements signals in a similar fashion: the Unix library locates the signal gate for the recipient process and sends a "kill" message to that gate. (Even though the Unix library has no global process ID table, it can still find the signal gate for a given process ID by enumerating that process's container to find the signal gate, since a process ID is the ID of that process's container object.)

> **10. [7 points]:** The HiStar Unix library is not trusted; anything the Unix library can do, ordinary program code can also do. As a result, the Unix library cannot be trusted to check that signals are only sent to processes owned by the same user. How does HiStar prevent a malicious user program from sending signals to other users' processes?

Answer: HiStar relies on the signal gate's clearance to restrict the set of threads that can send a signal to a process. Each process's signal gate has a clearance of $\{u_w 0, 2\}$, where the u_w category corresponds to the user that owns that process. This means only threads that either own u_w or have a label of $\{u_w 0, ...\}$ can invoke this gate, and one process cannot send signals to processes of other users whose categories it does not own.

11. [7 points]: Can KeyNIX be re-designed to prevent a compromised Unix keeper (controlled by an attacker) from sending signals to other users' processes? Sketch out a design that would run on KeyKOS, or describe why it would be difficult to implement using capabilities.

Answer: One possible design would be to keep the keys to Unix keepers in a per-user process table, rather than in a single global table. Unix keepers would maintain a key to the process table corresponding to the user they're running as, and would be able to send signals to processes in that table. Unix keepers of processes running as root would have a key to a top-level table of keys for every user's process table, allowing root to send signals to anyone's process. When a root process calls setuid to run as a particular user, its Unix keeper would delete its key for the top-level table, and only keep a key to the table for that user. A non-root process would not be able to send signals to processes of other users, since it would have no way to get the relevant key.

An alternative design would be to create a separate key representing each user in the system (the key could refer to anything, e.g. an empty segment), and pass this key as part of the "kill" message sent to any Unix keeper. Each Unix keeper would have a key for the user it's running as, and on receiving a "kill" message, it would use Keybits to compare the key it received with the key for the user it's running as, and only accept the signal if the keys matched. Care would have to be taken for signals sent by root. If a Unix keeper running as root were to include root's key in signals sent to non-root processes, those processes' Unix keepers could save that key and use it to send their own signals to other processes. To avoid this problem, keepers running as root would have to use the correct user's key when sending signals, perhaps using some other table to ensure they don't divulge one user's key to a Unix keeper of another user.

VIII RCU

Answer this question in the context of the paper *Read-Copy Update* by McKenney *et al*.

12. [7 points]: Illustrate a specific problem that could occur if the reference-counted linked list search code in Figure 4 did not obtain a read lock on list_lock. Be sure to indicate what would go wrong as a result.

Answer: `search()` might be about to use `p->next` for some element, while at the same time another thread is calling `delete()` for the same element. A third thread might re-use the freed element's memory for another purpose, overwriting `p->next`. If `search()` now dereferences `p->next`, it may crash or interpret an inappropriate piece of data as a list element.

13. [7 points]: The RCU version of the linked list search code in Figure 8 does not use locks. Describe the steps taken by RCU to ensure that the problem seen in the above question does not occur in this case.

Answer: In Figure 9, RCU's `kfree_rcu()` does not immediately free the list element; it defers freeing it until all threads have voluntarily given up the CPU. Since `search()` in Figure 8 doesn't voluntarily context switch, it can safely use any list element without fear that it will be concurrently freed and reused.

14. [8 points]: Suppose you want to use RCU for a linked list in the xv6 kernel. Explain how to detect quiescence in xv6. To maximize read performance, you may not use any additional code (such as locks) around read accesses to the linked list.

Answer: xv6 has involuntary context switches: if interrupts are enabled (which they usually are in the kernel), a timer interrupt can cause a switch to a different process. This means that any RUNNING or RUNNABLE thread that's in the kernel might hold a reference to an RCU-protected object; let's call them "suspect" threads. Thus, when an object is passed to `kfree_rcu()`, RCU must defer the free until every thread that is suspect at that time has voluntarily given up the CPU. In xv6, this means waiting for each such thread to call `sleep()` or return to user space or exit. One possibility is to associate, with each thread, a count of the number of voluntary context switches it has performed. `kfree_rcu()` would tag the object with each suspect thread's count. It would free the object after all of those threads' counts had changed.