CSE 120: Homework #1 Solutions

Fall 2014

1. **For each of the three mechanisms that supports dual-mode operation — privileged instructions, memory protection, and timer interrupts — explain what might go wrong without that mechanism, assuming the system only had the other two.**

   Without privileged instructions: Without privileged instructions, any process could execute any instruction (e.g., instructions for interacting with I/O devices, masking interrupts, manipulating virtual memory state, etc.). As a result, a buggy or malicious process could violate memory protection (access data in the address space of another process or the operating system), I/O protection (issue requests to disk for someone else's data), fairness (turning off interrupts to gain full control over the CPU), etc.

   Without memory protection: Similar to above, without memory protection any process can potentially access the data of any other process (of any user), as well as the data maintained by the operating system. [Early personal computers lacked such memory protection, and any application could potentially disrupt other applications or the operating system with stray pointers.]

   Without timer interrupts: Fundamentally, the timer interrupt is the essential mechanism that enables the operating system to regain control over the CPU on a regular basis. Without timer interrupts, a buggy or malicious process could run indefinitely on the CPU and deny the CPU to other processes and the operating system (e.g., an infinite loop).

2. **Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computers? Give arguments both that it is and that it is not possible.**

   By a secure operating system, we mean that a user program is not able to corrupt the kernel, prevent it from running, crash the system, violate memory protection, etc.

   There are several possible approaches for implementing a secure operating system. One is to write a simulator for a dual-mode processor (one with a privileged mode) and to run the operating system on top of this emulator. A slight variant of this is to run user code only in an emulator; this is the approach you use in Nachos.

   Another is to rely upon safe languages. Only executing user programs written in a safe language such as Java will work, since the language properties guarantee that arbitrary writes to memory aren't allowed and the instruction set is limited. Note here that the user doesn't get to choose arbitrary machine code to run—the code run is the result of just-in-time compiling the Java code. Similar to this is only running code which has been produced by a trusted compiler which is known to check memory addresses and not output privileged instructions.

   It is not enough to have the kernel scan the user code for bad instructions before executing it, if the code may have been produced by an arbitrary compiler. First, bad instructions may not be easy to identify—for example, they might be a write instruction which happens to overwrite

critical data, but which is not easily identified as such. Directly executing user code is also dangerous because the user code may do something indirect such as overwriting its code or writing out new code to a new page of memory and them jumping to it, and this code would then not be checked for security.

Similarly, asking the processor to check with the OS before executing each instruction doesn't quite work. I don't know of a processor which has a mode quite like this. Even if one did, that would essentially amount to having a mode bit—there has to be some way for the processor to know to check user code but not the operating system code itself. To make this work, you can use a full emulator (as described earlier).

3. **Which of the following instructions should be privileged? (Also give a one-sentence explanation for why.)**

   a) Set value of timer
   b) Read the clock
   c) Clear memory
   d) Turn off interrupts
   e) Switch from user to monitor mode

   Set value of timer: Yes, otherwise the user program can manipulate it such that the OS never gains control

   Read the Clock: No, as a user can't really do anything harmful by simply reading the clock.

   Clear Memory: Yes, since a user program shouldn't be able to clear arbitrary memory. (Exception: No, if interpreted as simply clearing memory belonging to the process.)

   Turn off interrupts: Yes, same reasoning as for setting the value of the timer.

   Switch from user to monitor mode: Yes, since otherwise a user program could simply switch to kernel mode to execute instructions it wouldn't otherwise be able to, and defeat security.

4. **For each of the following system calls, give a condition that causes it to fail: `open`, `read`, `fork`, `exec`, `unlink`.**

   `open`: File does not exist.

   `read`: Invalid file descriptor.

   `fork`: No more processes (OS out of memory).

   `exec`: Program file does not exist; file exists, but does represent a valid executable file (e.g., it is a spreadsheet); file exists, but the user does not have permission to execute it (e.g., the file does not have the exec bit set, or the user does not have permission to access the file); ...

   `unlink`: File does not exist; user does not have permission to access the file.

5. **List two challenges an OS faces when passing parameters between user and kernel mode. Describe how an OS can overcome them.**

   Safety: The OS needs to verify that data passed to the kernel is valid and safe to use. User

should not be able to panic the kernel by making syscalls.

Large Data: The OS sometimes needs to move a large amount of data — more than will fit in registers — to or from userspace (for example, read/write syscalls). This challenge is overcome by passing pointers to buffers containing the data, instead of passing the data itself.

Address Translation: The OS needs to translate pointers passed as arguments to data in the user address space so that the data is accessible in the kernel address space.

6. **The Java runtime provides a set of standard system libraries for use by programs. To what extent are these libraries similar to the system calls of an operating system, and to what extent are they different?**

   Java libraries are similar to OS syscalls because they both provide interfaces that allow the user to do things they couldn't do otherwise (read from file, etc). Java libraries are different from OS syscalls because Java libraries are not actually privileged — Java libraries just invoke OS syscalls.

7. **Suppose the hardware interval timer only counts down to zero before signalling an interrupt. How could an OS use the interval timer to keep track of the time of day?**

   The operating system can program the interval timer to go off after some short time interval, say 10 ms. Each time the timer interrupt fires, the OS resets the timer, and increments a counter. By multiplying the counter by the timer interval, the operating system can measure the amount of time which has progressed. If the operating system knows what time it booted (say it has a clock which can tell it this, or it asks the user, or checks the network), it can determine the exact time of day.

   It is not sufficient to simply program the timer to a large value and check to see the amount of time remaining, since the operating system will also need to use the timer for other purposes, such as interrupting user programs for context switching.

8. **Suppose you have to implement an operating system on hardware that supports interrupts and exceptions but does not have a trap instruction. Can you devise a satisfactory substitute for traps using interrupts and/or exceptions? If so, explain how. If not, explain why.**

   (Background) Processes execute the trap instruction to invoke system calls in the operating system. The trap instruction ensures a controlled and protected transition from user-level to kernel-level, enabling user-level processes to execute code in the operating system on their behalf.

   Exceptions like divide by zero or invalid instruction also cause a controlled and protected transition from user-level to kernel-level. If the underlying hardware does not provide a trap instruction, an operating can use exceptions instead as a hack. For example, an operating system can have the convention that executing an invalid instruction will take the place of a trap instruction since executing an invalid instruction will cause an exception that immediately transfers control to the operating system.

   Additional details (not needed in an answer) are how to specify which system call to invoke

and how to pass arguments. By convention, for instance, the system call number and arguments can be placed on the process stack. In verifying all arguments, the operating system can also distinguish between using an invalid instruction for a system call and the process actually executing an invalid instruction unintentionally.

9. **Suppose you have to implement an operating system on hardware that supports exceptions and traps but does not have interrupts. Can you devise a satisfactory substitute for interrupts using exceptions and/or traps? If so, explain how. If not, explain why.**

   In this situation, it would not be possible to use exceptions or traps to substitute for interrupts. Exceptions and traps are synchronous, and interrupts are inherently asynchronous; exceptions and traps happen as a result of process doing something, whereas interrupts are external events that happen at unpredictable times. Going back to the timer interrupt, for example, we would not be able to interrupt a process that has complete control over the CPU (e.g., an infinite loop) using exceptions or traps.

   Polling for I/O devices only partially solves the problem since, e.g., polling cannot replace the timer interrupt needed to prevent one process from having complete control over the CPU.

---

voelker@cs.ucsd.edu