CS-323: OS Exercise Week 3 (Fall 2023)

October 2, 2023

Last name, first name: _____

1 Time- vs Space-sharing

(a) What is the difference between time- and space-sharing?

From the slides:

- time sharing: exclusive use, one at a time. Eg. multiplexed processes on a single core.
- space sharing: everyone gets a small chunk all the time. Eg. processes pinned to separate cores on a multicore processor.

(b) What would be real-world examples of time- and space-sharing?

A bartender is an example of a time-shared resource. She can serve multiple clients but will interact with only one at a time.

In contrast, the bar in itself is a space-shared resource. Each client gets a part (seat) of the bar for the duration of their presence.

2 Traps and Interrupts

(a) What is the difference between a trap and an interrupt?

Traps are synchronous and occur when the CPU executes an instruction. Interrupts refer to hardware interrupts, usually caused by something other than the CPU and are asynchronous. They can occur at any point, independent of what instruction the CPU is currently executing.

(b) Give an example of a trap event. Can you also find a second example?

The instruction used to make system calls causes a trap event. Another example could be a division by zero error. Both of these happen due to an executed instruction.

(c) Give an example of an interrupt event. Can you also find a second example?

The timer interrupt generated periodically by the CPU timer is an example, and occurs independently of the instruction currently being executed. Another example would be moving the mouse, which generates an interrupt that the CPU handles immediately by updating the mouse position.

(d) How does a running process suddenly switch to another piece of code when an interrupt happens? How does execution return afterwards? Explain the mechanism.

When an interrupt happens, the CPU will suspend the currently running process, save its state (current register values, including stack pointer and program counter) and enter kernel mode. It will then call the handler corresponding to the current interrupt on the interrupt/trap table. After this short piece of code - the handler - finishes executing, the CPU will switch back to user mode and restore the process state (registers). The suspended process will then continue executing as if nothing had happened!

Name: _____

3 Process Execution and Isolation

(a) What is the reason for having separate user and kernel execution modes on a CPU? Why not have a single mode?

Having all privileges always enabled is a security and reliability issue: Any program would be able to directly modify disk blocks, read and modify other processes' memory or gain unrestricted control of the CPU. Instead, only the operating system runs in a mode called kernel mode and is allowed unrestricted access to all instructions. Processes run in user mode and are not allowed to execute instructions that can perform the aforementioned modifications. Processes are allowed only restricted access to privileged operations via a well defined system call interface calling into the operating system. Both their memory and CPU usage is restricted and virtualized, preventing them from accessing each other's memory or taking control of the CPU directly. Disk access is limited to abstract modifications via files, instead of direct block modifications.

(b) Imagine your program makes use of a buggy system call that disables interrupts and forgets to re-enable them. How long will your program continue to execute? When would the OS regain control of the CPU?

In case interrupts are disabled, all interrupts including the timer interrupt that would allow the OS to gain back control will be disabled. The program will continue to execute until it voluntarily yields the CPU to the OS via a system call, or until it finishes executing. Using the computer will also become very difficult in the system due to the lack of interrupts: The mouse and keyboard will not respond, and the screen or connected terminals will freeze. *As a tidbit for later, even the whole system could crash if a page fault occurs while accessing memory, since it cannot be handled without an interrupt.*

(c) Modern CPUs provide hardware support for many operating system mechanisms. One example of this are the security rings (0 to 3) on x86 and x86_64 processors, which allow for executed instructions to be checked. Why do you think this kind of support exists? Can the OS not check each instruction before executing it at the software level?

Doing things at the software level usually has a significant overhead, the OS checking every instruction before execution via software would not be very easy. At the most basic level, each instruction would have to go through an if-else check, which would significantly increase overhead: doing this may require memory access and a bunch of other instructions. This is without even considering *how* the CPU may call into the OS to do the checking. Should every instruction cause a trap and a context switch? This would make execution dozens if not hundreds of times slower than executing a single instruction on CPU. Allowing the CPU to switch modes and do the checking via hardware allows things to stay very fast. As the course goes on, you will learn about more examples in which hardware is designed with OS concepts in mind to make things easier, faster and safer.