CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Interrupts

One of the jobs of the OS is to tell other programs what they can and cannot do

The OS is just a big program. How can you write a program that stops another program from doing something **when that other program has control of the CPU**?

Shouldn't it be the case that anything the OS can do, so can any other program?

Hardware support!

Modern CPUs have different **privilege levels**

Most commonly referred to as **kernel mode** and **user mode**, though other names exist, and some CPUs support more than two modes

kernel mode -- processor has no limits on which operations it can perform

user mode -- hardware prevents processor from performing certain operations, including I/O and accessing certain memory areas

Older processors did not have this feature, so OSes **did** have to trust that any program they ran wouldn't bring down the system.

Side note -- Other important hardware features

Memory controller restricts access to a certain range unless the OS is making the request

Hardware timer sends control back to OS at regular intervals

Both of these rely on the existence of the user-mode vs. kernel-mode distinction

Think about ways the system could fail if these features did not exist

Switching to and from kernel mode

Hardware-enforced kernel mode is great, but doesn't do any good if a process can put itself in kernel mode

System starts in kernel mode with OS as first (major) program. When OS hands control to another program, OS must always ensure it switches to user mode

User program can request change to kernel mode, **but it cannot choose what code will be run next.** Whenever switching to kernel mode, must jump to code that is part of OS and therefore safe

Request to go to kernel mode is an *interrupt*. "Interrupt" usually refers to request generated by hardware

For example, a disk controller might notify processor it has finished reading data so processor can take appropriate action

In addition to hardware, both user software and CPU itself can issue interrupts. These are called *exceptions* or *traps*. (Terminology of interrupts vs. exceptions vs. traps is not always clear -- don't worry to much about it)

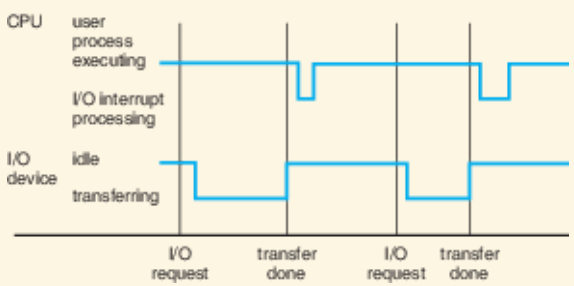
General reasons for interrupts

Interrupt -- hardware requires CPU attention (disk I/O, timer, etc.)

Trap -- CPU itself runs into special case (e.g., division by 0) and needs software support

System call -- user program is asking OS to do something program does not have privileges to do itself

I/O interrupt



I/O interrupts make the system more efficienct

Otherwise, the CPU would need to regularly poll the I/O devices to determine whether they needed attention, even if they rarely did

How are interrupts handled?

When CPU detects an interrupt, rather than continuing the current process, it

- saves the program counter
- switches to kernel mode
- jumps to OS code to handle the exception
- restores the program state, switches to user mode, and jumps back

Interrupts are **transparent to the user program**

How does CPU know where to jump?

Each interrupt comes with an ID describing which device generated it

OS keeps a table in a known spot in memory containing locations of interrupt handlers -- this is the *interrupt vector table*

Completely made up interrupt vector table:

ID	reason	location of handler
0	division by 0	`0x00208008`
1	disk finished	`0x00930004`
2	key pressed	`0x0008000A`
...	...	...

Using ID as index to table, OS jumps to specified interrupt handler

Where is PC saved?

Some architectures have a special register for this

More common now to save on a stack

Interrupt handler may also need to save other elements of program state. Key is to go back to previous program **exactly as it was** when finished

It is possible to temporarily disable interrupts

Is this available in user mode, or is it kernel mode only?

One especially important interrupt mentioned previously was timer

Without this, how would OS allow fair sharing of CPU time?

Not possible in general! Would need to trust every program on the system to willingly give up control of CPU. That method is called **cooperative multitasking**

When OS can interrupt another running program and switch control to a different program, that is **preemptive multitasking**

Recap:

- CPU has kernel mode and user mode
- User-mode process passes control to OS when there is an interrupt
- Interrupts can be signalled by hardware (I/O), CPU itself, or system calls
- Interrupts are handled by pre-defined functions in the interrupt vector table
- Interrupts are transparent to user programs
- This requires hardware support