Interrupts

Motivation

- When a user process is running on the CPU, how does OS stop it?
- When a user process accesses disk, how does OS verify that user has permission?
- When a user process crashes, how does OS regain control?

Topics

- Kernel Mode verses User Mode
- Interrupt Types
- IDT setup
- Interrupt lookup
- HW Interrupt
- System Calls
- Exceptions
- Examples

Kernel Mode verses User Mode

- Kernel Mode
  - Most privileged
  - Access to entire file system, memory space, all hardware
  - OS runs in this mode
- User Mode
  - Least privileged
  - Access to resources (like files and memory) that belong to current user
  - User processes run in this mode

Kernel Mode verses User Mode

- Can be implemented with 1 bit in a CPU register, register is only accessible in Kernel mode (means user mode programs cannot modify this bit)

- What mode is process in?
  - 0=Kernel,
  - 1=User

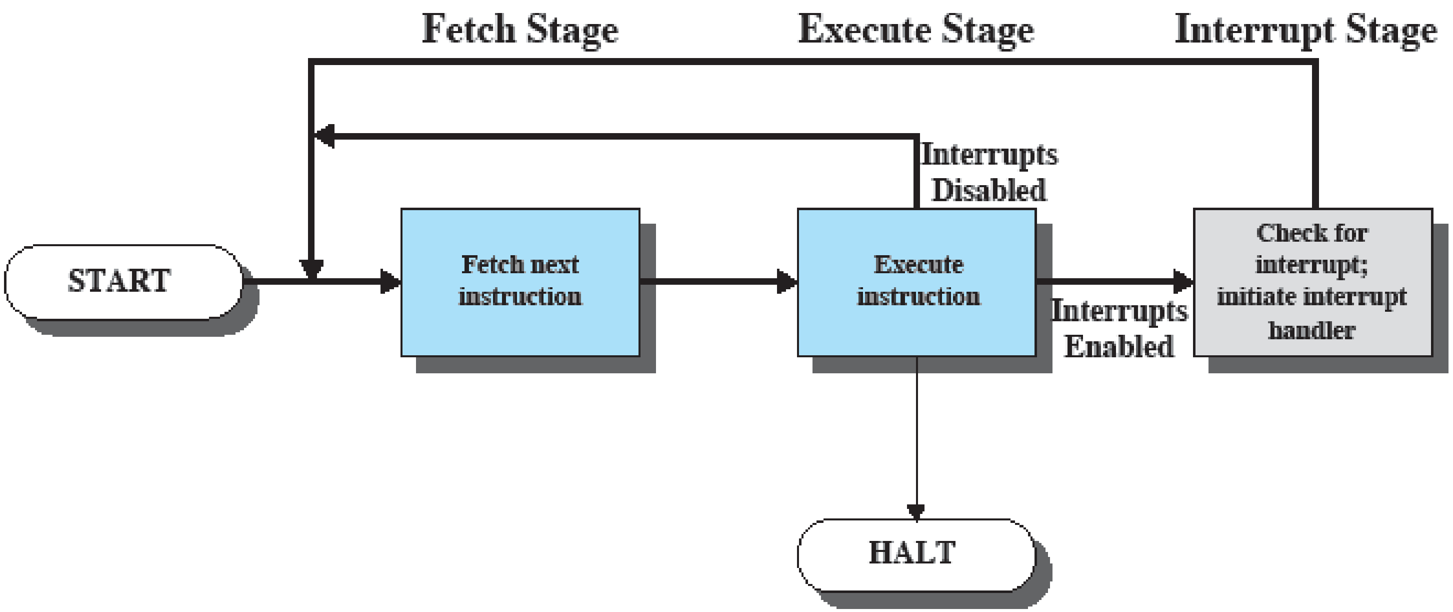
- Set/Reset when handling interrupts (more coming)

Interrupt Types

- Hardware
  - Raised by hardware devices
  - Can occur at any time (Asynchronous)
  - Examples: timer (process switch), I/O signals
- System calls:
  - Software interrupts (Synchronous)
  - Raised by user programs to invoke OS functionality
- Exceptions
  - Generated by processor as a result of illegal action (Synchronous)
  - Faults: recoverable (page fault)
  - Aborts: difficult to recover (divide by 0)

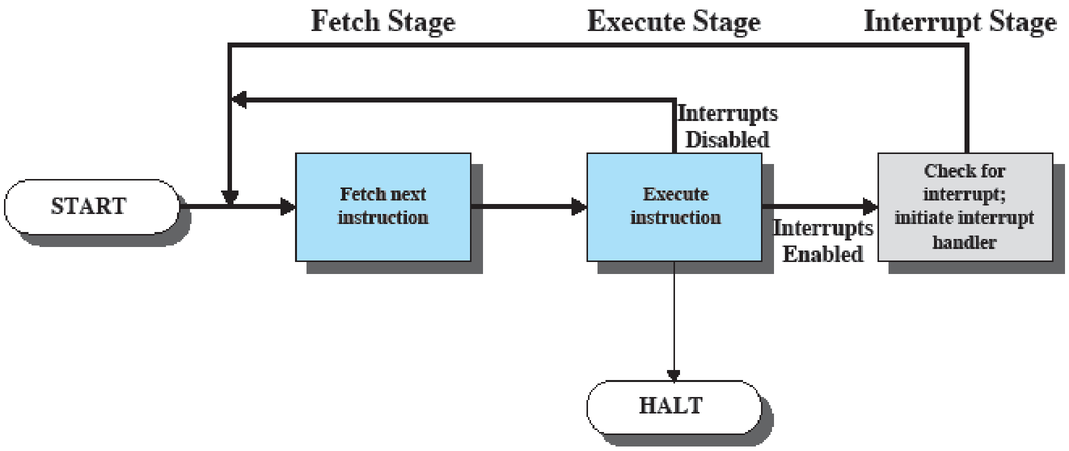
Interrupt servicing

- Remember this cycle is for 1 machine instruction (assembly)
- NOT a high level language (like C++)
- Uses lookup table to find correct interrupt handler (**coming soon!**)
- Works for both synchronous and asynchronous interrupts



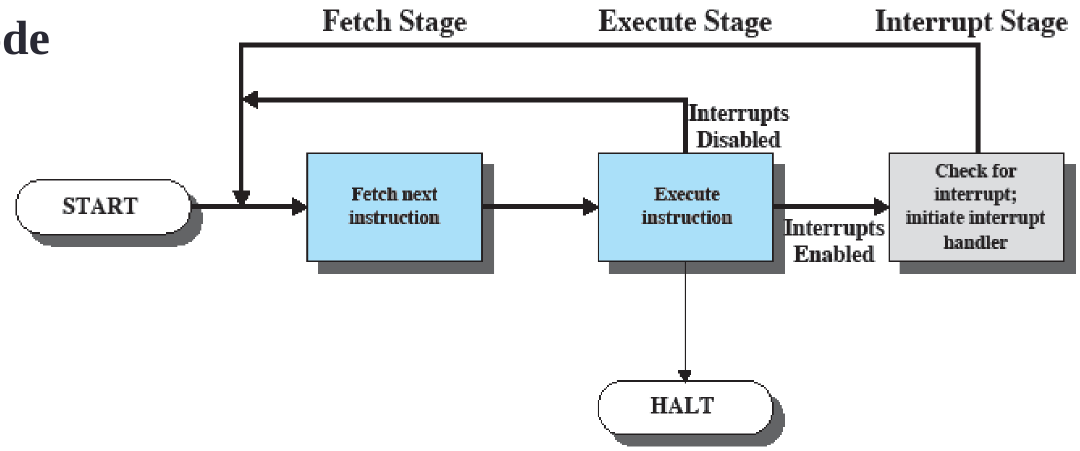
Interrupt servicing – pseudo code

```
while (fetch next instruction) {
    run instruction;
    if (interrupt occurred) {
        save process state     // user mode
        find and jump to OS-provided interrupt handler // kernel mode
        restore original process state from kernel stack
    }
    // user mode
}
```



Interrupt servicing – better pseudo code

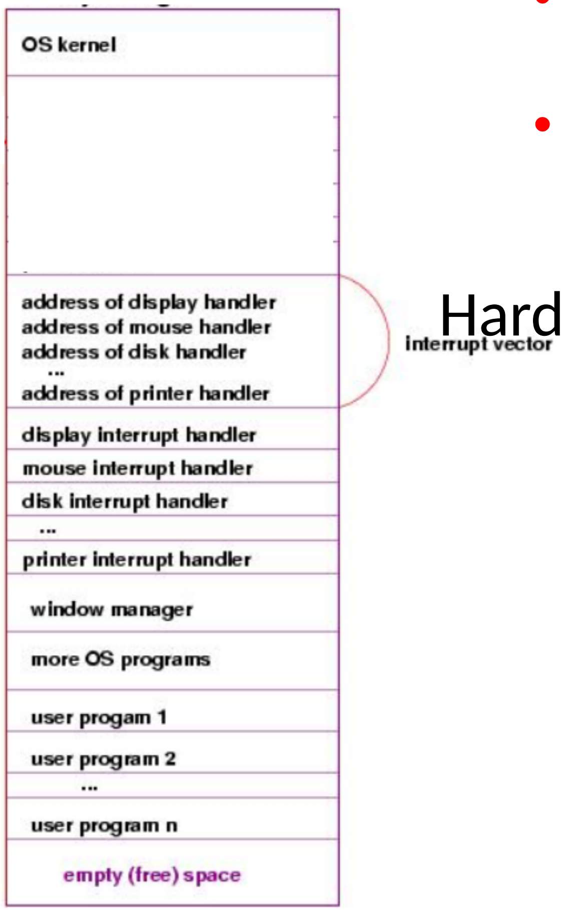
```
while (fetch next instruction) {
    run instruction;
    if (interrupt occurred) {
        save process state     // user mode
        find and jump to OS-provided interrupt handler // kernel mode
        OS chooses next process to run
        if  new process
                load its state from mem
        else
            restore original process state from kernel stack
    }
    // user mode
}
```



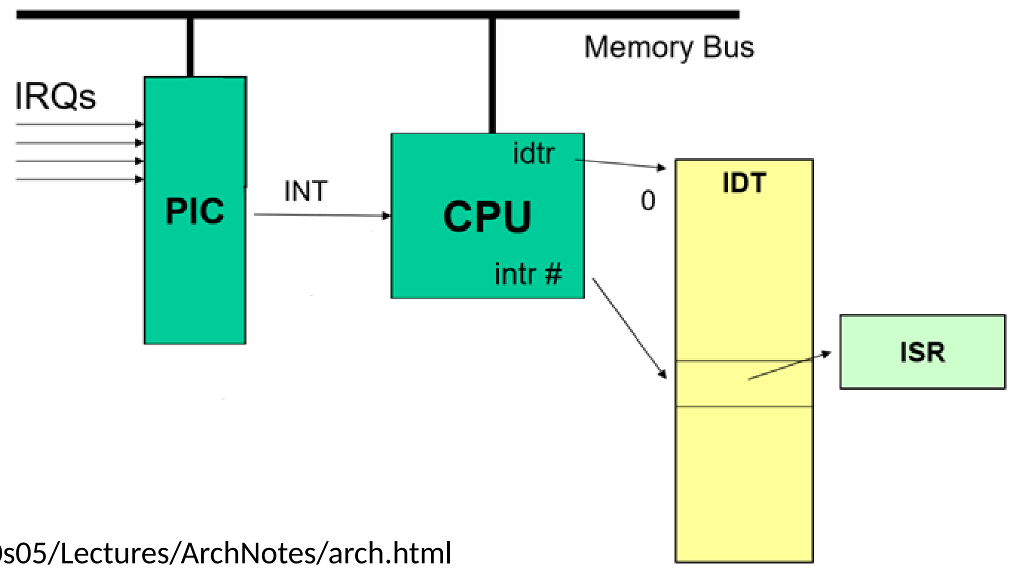
Interrupt Descriptor Table (IDT) setup

- OS sets up IDT at boot, its base pointed to by IDT register (IDTR) in CPU
- Each entry is address of an interrupt handler (known as Interrupt Service Routine (ISR))
- Each ISR handles a particular type of interrupt.

IDT Memory Layout

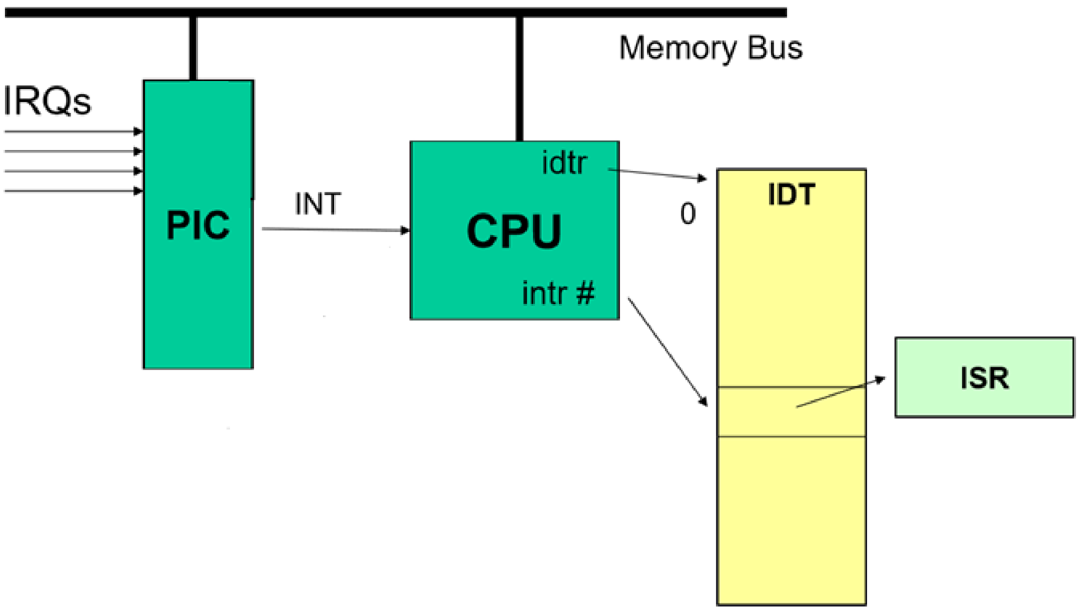


Hardware Interrupt Sequence



Diagrams from http://people.cs.ksu.edu/~schmidt/300s05/Lectures/ArchNotes/arch.html

Hardware Interrupt
Asynchronous interrupt, can happen anytime



- Intr# generated by CPU
- Save current process state
- In kernel mode – look up Intr# in IDT (idtr + sise_of(row)*intr)
- Jump to, and run ISR
- Either return to running process (reload its state including Instruction Pointer (IP) or load and run new one

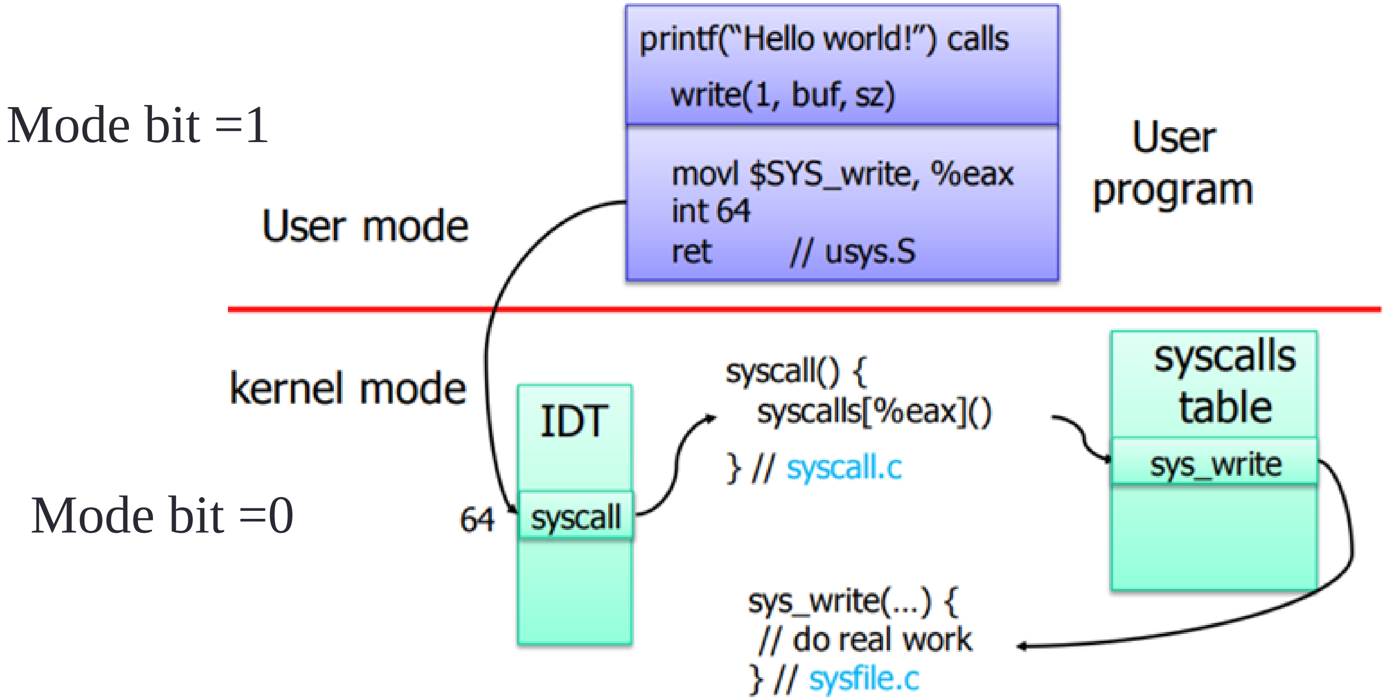
System Calls

- Applications cannot perform privileged operations themselves
  - Otherwise they could access devices they should not (DMA controller to access other processes memory), write files they don't have permission too, elevate their privileges, etc.
- Instead, they ask OS to do so on their behalf by issuing system calls
- OS verifies permissions, then parameters then fulfills request.

System Call Types

- **Process Control -** process creation, process termination etc.
  - Ex. fork(), exit(), wait()
- **File Management-** file manipulation such as creating a file, reading a file, writing into a file etc.
  - Ex. open(), read(), write(), close() //to file
- **Device Management -** device manipulation such as reading from device buffers, writing into device buffers etc.
  - Ex. read(), write() //to console
- **Information Maintenance -** handle information and its transfer between the operating system and the user program.
  - Ex. getpid(), alarm(), sleep()
- **Communication -** interprocess communication. They also deal with creating and deleting a communication connection.
  - Ex pipe()

System Call
synchronous interrupt (you know its coming)



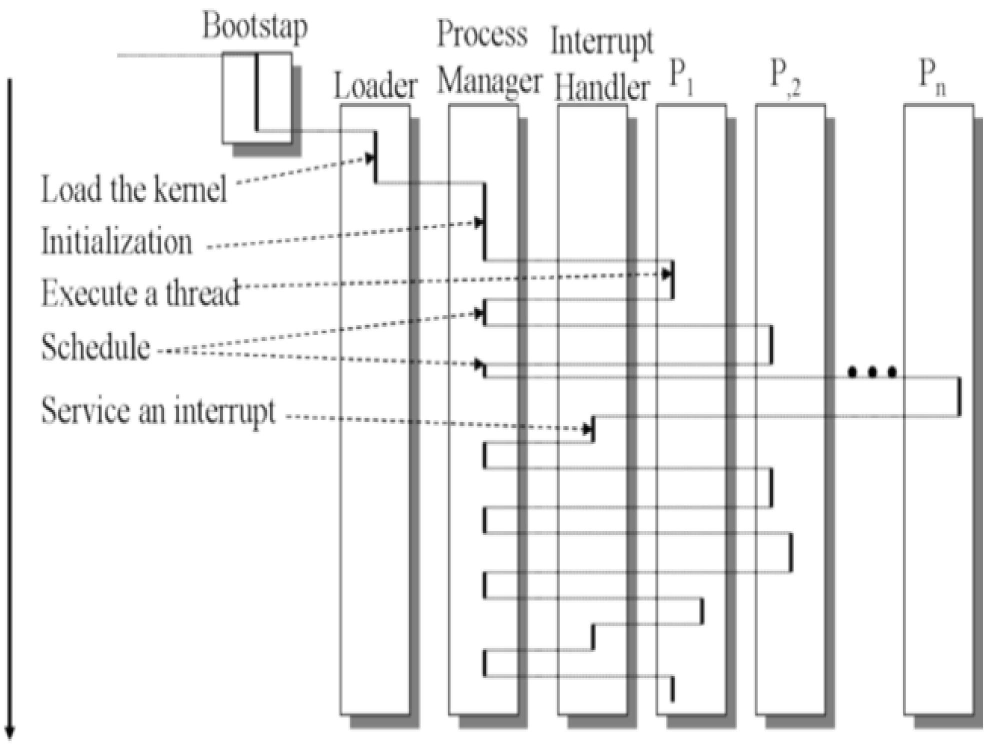
- Save what system call (Sys_write) in eax reg
- int 64 synchronous interrupt
- In kernel mode – look up interrupt 64 in IDT (syscall)
- Use eax to index into syscalls table to get to sys_write(…)
- run sys_write()
- Either return to running process or load and run new one

Exceptions

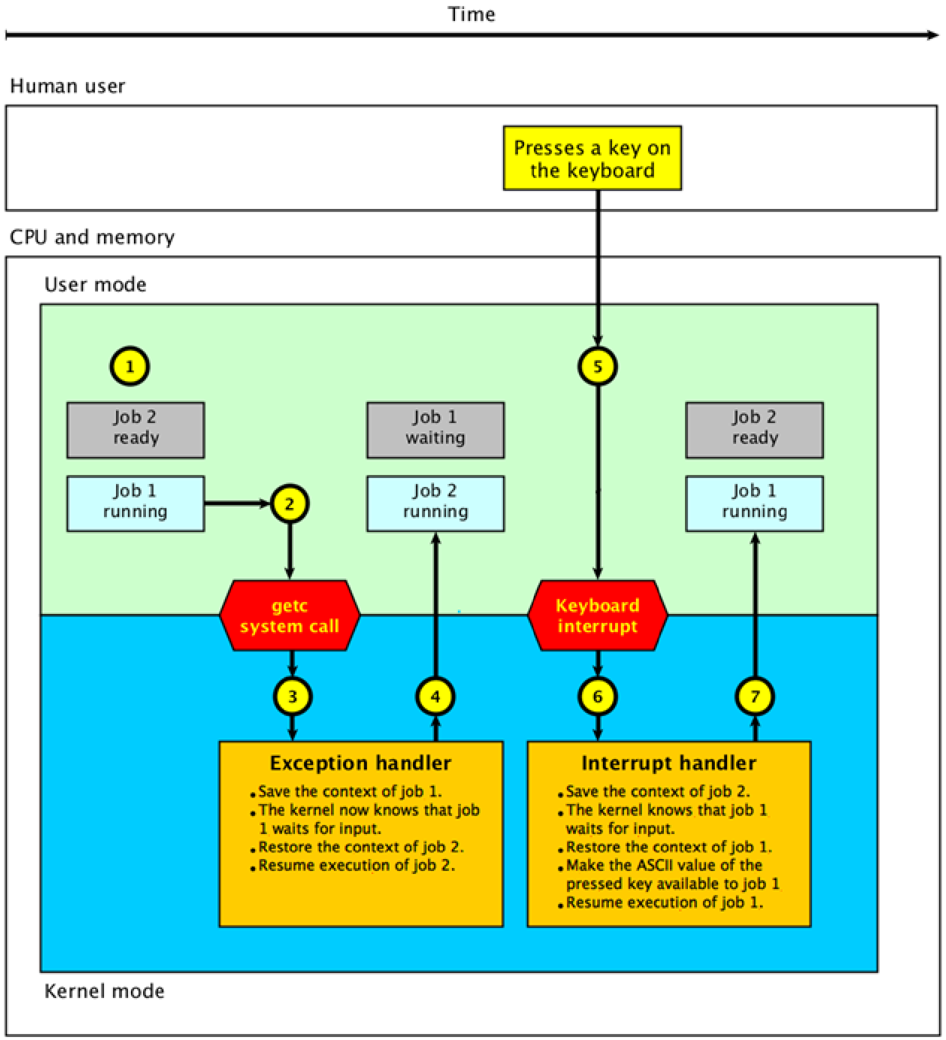
- Divide by 0, dereference Null pointer, page fault
- Throws exception interrupt
- Transition to Kernel mode to handle interrupt type as defined in IDT

- Some are recoverable;
  - Page fault-OS blocks current process, request page from mem, loads another process. When page available OS interrupted again and page is loaded and page faulting process can run again
- Some are not
  - Divide by zero, OS logs error, terminates current process, loads a new process

System operation with Interrupts



Example - 2 programs, HW and system call interrupts



1. Job 2 is ready to run and job 1 is running.

2. Job 1 uses the getc system call to read a character from the keyboard.

3. The system call uses a **system call exception** to enter the kernel.
   - The kernel saves the context (all register values) of job 1.
   - The kernel now knows that job 1 waits for input and changes its state from **running** to **waiting**.
   - The kernel restores the context of job 2.

4. The kernel resumes execution of job 2 and changes its state from **ready** to **running**.

5. The human user presses a key on the keyboard.

6. The key-press causes a keyboard **interrupt** which is handled by the kernel.
   - The kernel saves the context of job 2 and changes its state from **running** to **ready**.
   - The kernel knows that job 1 waits for the getc system call to complete.
   - The kernel restores the context of job 1 and changes its state from **waiting** to **running**.
   - The ASCII value of the pressed key is made available to job 1.

7. The kernel resumes execution of job 1.

Content from http://www.it.uu.se/education/course/homepage/os/vt18/module-1/system-call-design/

Summary

- Kernel verses user mode, when and how they are entered
- IDT and ISRs
- Types of interrupts (HW, System call, exception)
- Examples of both hardware and software interrupts

References

- This presentation was developed with the aid of the OSTEP text and the following websites
- http://faculty.salina.k-state.edu/tim/ossg/Introduction/sys_calls.html
- http://www.cs.columbia.edu/~junfeng/13fa-w4118/lectures/l06-trap.pdf
- http://www.it.uu.se/education/course/homepage/os/vt18/module-1/system-call-design/