Event Driven kernel
Multi-tasking, Multi-programming

Earlier...

- **Boot process**
  - **BIOS -> Boot Loader -> OS -> “init” process**
- **CPU doing**

```
for(;;) {
    fetch from @PC;
    deode+execute;
    PC++/change PC
}
```

Understanding hardware interrupts

- Hardware devices (keyboard, mouse, hard disk, etc) can raise “hardware interrupts”
- Basically create an electrical signal on some connection to CPU (/bus)
- This is notified to CPU (in hardware)
- Now CPU’s normal execution is interrupted!
  - What’s the normal execution?
  - CPU will not continue doing the fetch, decode, execute, change PC cycle !
  - What happens then?

Understanding hardware interrupts

- **On Hardware interrupt**
  - **The PC changes to a location pre-determined by CPU manufacturers!**
  - **Now CPU resumes normal execution**
    - **What's normal?**
    - **Same thing: Fetch, Decode, Execute, Change PC, repeat!**
    - **But...**
  - **But what's there at this pre-determined address?**
  - **OS! How's that ?**

Boot Process

- **BIOS runs “automatically” because at the initial value of PC (on power ON), the manufacturers have put in the BIOS code in ROM**
  - **CPU is running BIOS code . BIOS code also occupies all the addresses corresponding to hardware interrupts.**
- **BIOS looks up boot loader (on default boot device) and loads it in RAM, and passes control over to it**
  - **Pass control? - Like a JMP instruction**
  - **CPU is running boot loader code**
- **Boot loader gets boot option from human user, and loads the selected OS in memory and passes control over to OS**
  - **Now OS is running on the processor !**

Hardware interrupts and OS

- **When OS starts running initially**
  - **It copies relevant parts of it's own code at all possible memory addresses that a hardware interrrupt can lead to!**
  - **Intelligent, isnt' it?**
- **Now what?**
  - **Whenever there is a hardware interrupt – what will happen?**
  - **The PC will change to predetermined location, and control will jump into OS code**
- **So remember: whenever there is a hardware interrupt, OS code will run!**
- **This is "taking control of hardware"**

Key points

- **Understand the interplay of hardware features + OS code + clever combination of the two to achieve OS control over hardware**
- **Most features of computer systems / operating systems are derived from hardware features**
  - **We will keep learning this throught the course**
  - **Hardware support is needed for many OS features**

Time Shared CPU

- **Normal use: after the OS has been loaded and Desktop environment is running**
- **The OS and different application programs keep executing on the CPU alternatively (more about this later)**
  - **The CPU is time-shared between different applications and OS itself**
- **Questions to be answered later**
  - **How is this done?**
  - **How does OS control the time allocation to each?**
  - **How does OS choose which application program to run next?**
  - **Where do the application programs come from? How do they start running ?**
  - **Etc.**

Multiprogramming

- **Program**
  - **Just a binary (machine code) file lying on the hard drive. E.g. /bin/ls**
  - **Does not do anything!**
- **Process**
  - **A program that is executing**
  - **Must exist in RAM before it executes. Why?**
  - **One program can run as multiple processes. What does that mean?**

Multiprogramming

- **Multiprogramming**
  - A system where multiple processes(!) exist at the same time in the RAM
  - But only one runs at a time!
    - Because there is only one CPU
- **Multi tasking**
  - Time sharing between multiple processes in a multi-programming system
  - The OS enables this. How? To be seen later.

Question

- **Select the correct one**

  1) A multiprogramming system is not necessarily multitasking

  2) A multitasking system is not necessarily multiprogramming

Events , that interrupt CPU's functioning

- **Three types of "traps" : Events that make the CPU run code at a pre-defined address**
  - **1) Hardware interrupts**
  - **2) Software interrupt instructions (trap)**
    - **E.g. instruction "int"**
  - **3) Exceptions**
    - **e.g. a machine instruction that does division by zero**
    - **Illegal instruction, etc.**
    - **Some are called "faults", e.g. "page fault", recoverable**
    - **some are called "aborts", e.g. division by zero, non-recoverable**
- **The OS code occupies all memory locations corresponding to the PC values related to all the above events!**

Multi tasking requirements

- Two processes should not be
  - Able to steal time of each other
  - See data/code of each other
  - Modify data/code of each other
  - Etc.
- The OS ensures all these things. How?
  - To be seen later.

But the OS is “always” “running”
“in the background”
Isn’t it?

Absolutely No!

Let’s understand
What kind of
Hardware, OS interplay
makes
Multitasking possible

Two types of CPU instructions and two modes of CPU operation

- CPU instructions can be divided into two types
- Normal instructions
  - mov, jmp, add, etc.
- Privileged instructions
  - Normally related to hardware devices
  - E.g.
    - IN, OUT # write to I/O memory locations
    - INTR # software interrupt, etc.

Two types of CPU instructions and two modes of CPU operation

- CPUs have a mode bit (can be 0 or 1)
- The two values of the mode bit are called: User mode and Kernel mode
- If the bit is in user mode
  - Only the normal instructions can be executed by CPU
- If the bit is in kernel mode
  - Both the normal and privileges instructions can be executed by CPU
- If the CPU is “made” to execute privileged instruction when the mode bit is in “User mode”
  - It results in a illegal instruction execution
  - Again running OS code!

Two types of CPU instructions and two modes of CPU operation

- **The opearting system code runs in kernel mode.**
  - **How? Wait for that!**
- **The application code runs in user mode**
  - **How? Wait !**
  - **So application code can not run privileged hardware instructions**
- **Transition from user mode to kernel mode and vice-versa**
  - **Special instruction called “software interrupt” instructions**
  - **E.g. INT instruction on x86**

Software interrupt instruction

- **E.g. INT on x86 processors**
- **Does two things at the same time!**
  - **Changes mode from user mode to kernel mode in CPU**

    + Jumps to a pre-defined location! Basically changes PC to a pre-defined value.
    - **Close to the way a hardware interrupt works. Isn't it?**
  - **Why two things together?**
  - **What's there are the pre-defined location?**
    - **Obviously, OS code. OS occupied these locations in Memory, at Boot time.**

Software interrupt instruction

- **What's the use of these type of instructions?**
  - **An application code running INT 0x80 on x86 will now cause**
    - **Change of mode**
    - **Jump into OS code**
  - **Effectively a request by application code to OS to do a particular task!**
  - **E.g. read from keyboard or write to screen !**
  - **OS providing hardware services to applications !**
  - **A proper argument to INT 0x80 specified in a proper register indicates different possible request**

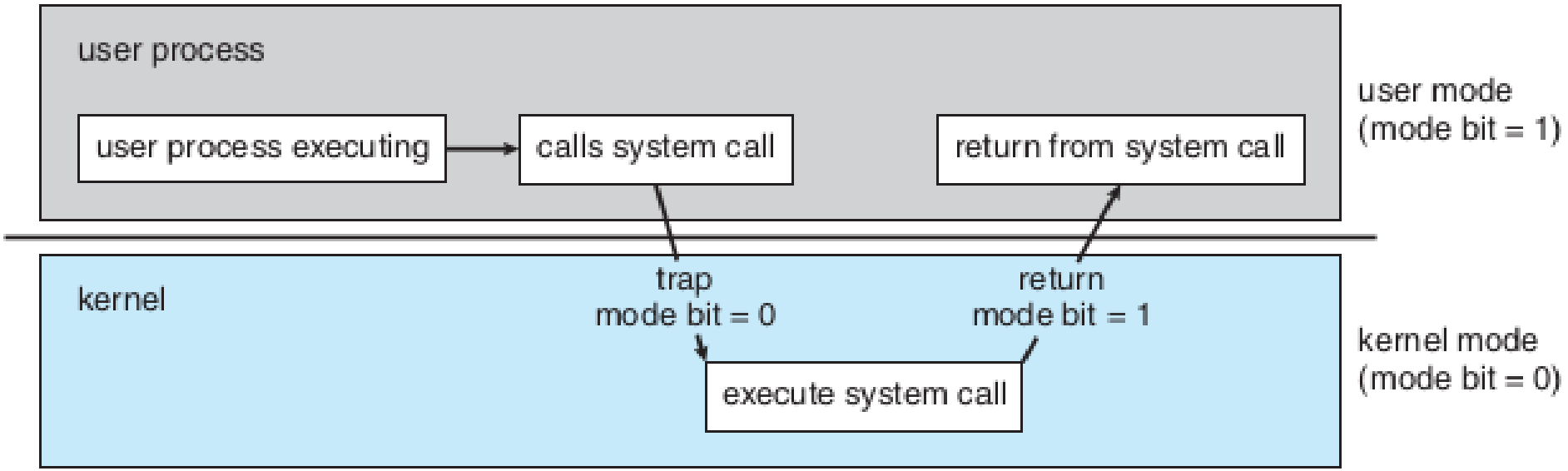

Figure 1.10 Transition from user to kernel mode.

Software interrupt instruction

- **How does application code run INT instruction?**
  - **C library functions like printf(), scanf() which do I/O requests contain the INT instruction!**
  - **Control flow**
    - **Application code -> printf -> INT -> OS code -> back to printf code -> back to application code**

Example: C program

```
int main() {
    int i, j, k;
    k = 20;
    scanf("%d", &i);  // This jumps into OS and returns back
    j = k + i;
    printf("%d\n", j); // This jumps into OS and returns back
    return 0;
}
```

Interrupt driven OS code

- **OS code is sitting in memory , and runs intermittantly . When?**
  - **On a software or hardware interrupt or exception!**
  - **Event/Interrupt driven OS!**
  - **Hardware interrupts and exceptions occur asynchronously (un-predictedly) while software interrupts are caused by application code**

Interrupt driven OS code

- **Timer interrupt and multitasking OS**
  - **Setting timer register is a privileged instruction.**
  - **After setting a value in it, the timer keeps ticking down and on becoing zero the timer interrupt is raised again (in hardware automatically)**
  - **OS sets timer interrupt and “passes control” over to some application code. Now only application code running on CPU !**
  - **When time allocated to process is over, the timer interrupt occurs and control jumps back to OS (hardware interrupt mechanism)**
  - **The OS code that runs here (the ISR) is called “scheduler”**
  - **OS can now schedule another program**

What runs on the processor ?

- **4 possibilities.**