System Calls, fork(), exec()
Event Driven Kernel,
Multi-tasking OS

Abhijit A. M.
abhijit.comp@coep.ac.in

Credits: Slides of “OS Book” ed10.

Building an “OS”

- E.g. Debian
  - A collection of thousands of applications, libraries, system programs, ... and Linux kernel !
  - Linux kernel is at the heart, but heart is not a human without the body !
  - Job of “Debian Developers”
    - Collect the source code of all things you want
    - Create an “Environment” for compiling things : bootstrap challenge
    - Compile everything
    - Ensure that things work “with each other”
      - Very difficult! Versions, dependencies !
    - “Package” things (e.g. make .deb files)

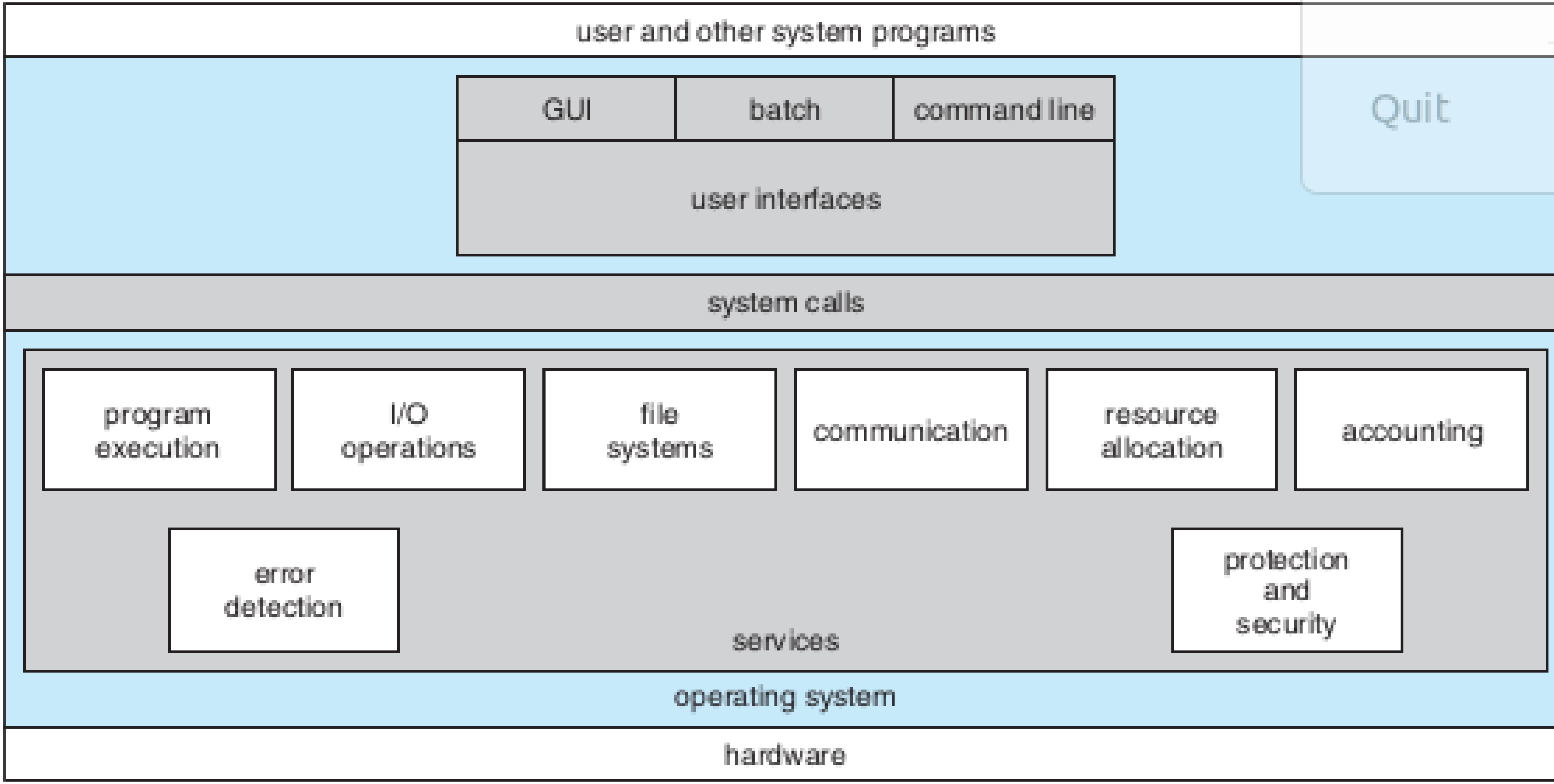


Figure 2.1 A view of operating system services.

Components of a computer system Abstract view

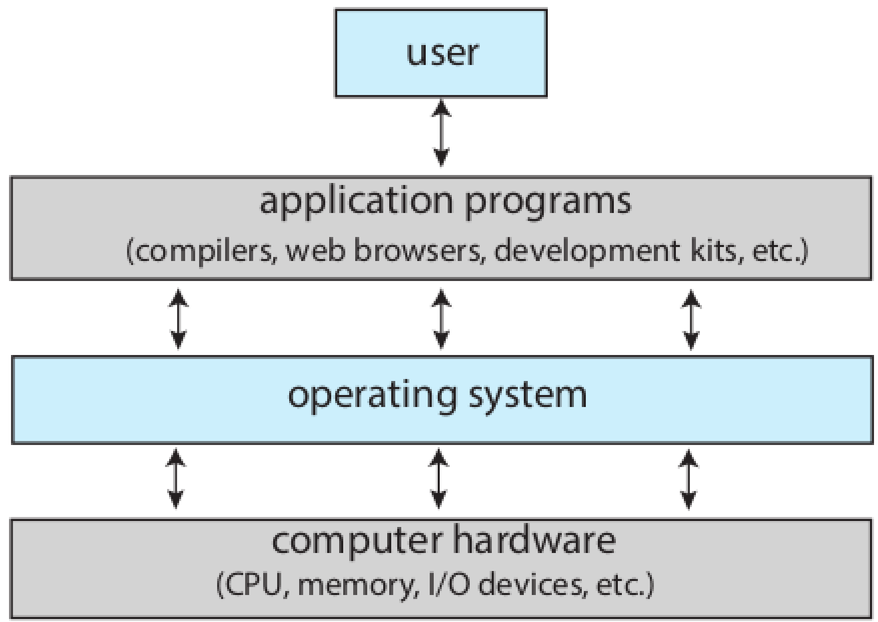
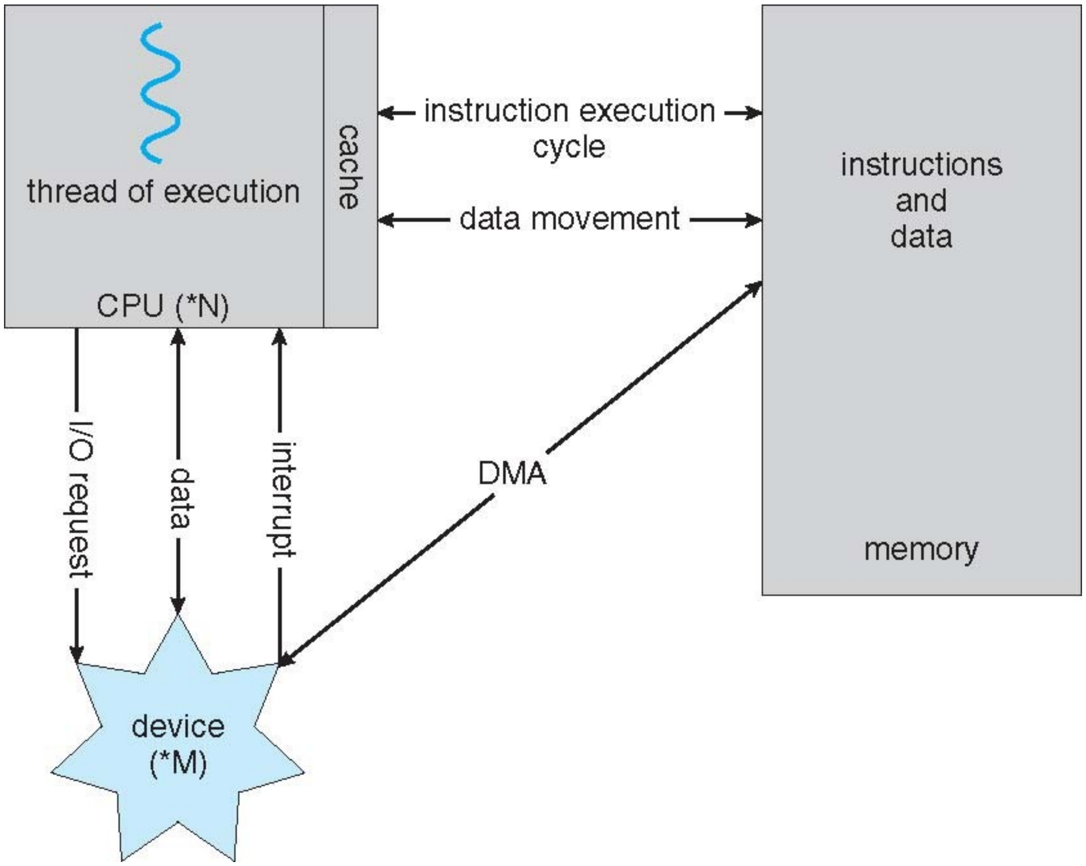


Figure 1.1 Abstract view of the components of a computer system.

Von Neumann architecture



- Processor
  - Fetch
  - Decode
  - Execute

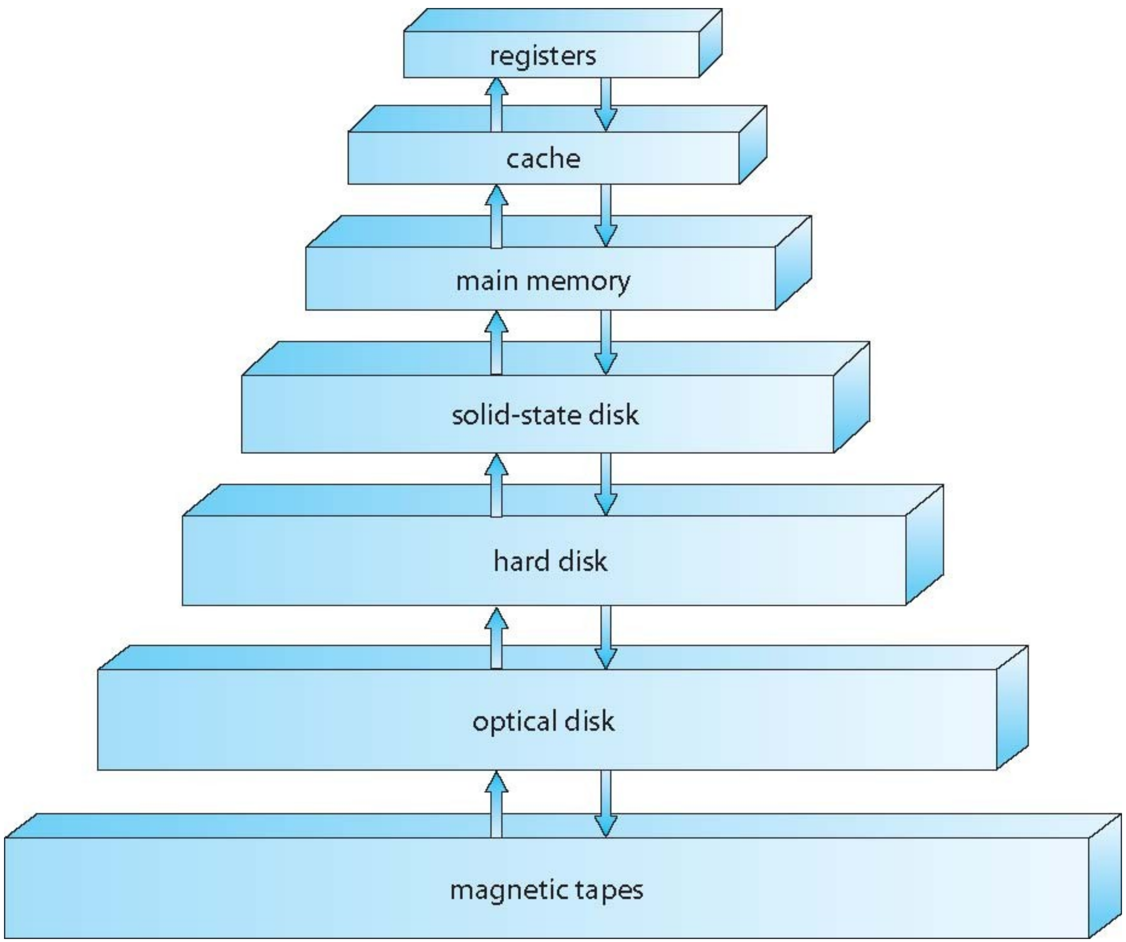
  Repeat

A von Neumann architecture

A key to “understanding”

- Everything happens on processor
- Processor is always running some instruction
- We should be able to tell possible execution sequences on processor

Memory hierarchy



Memory hierarchy

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

System Calls

- Services provided by operating system to applications
  - Essentially available to applications by calling the particular software interrupt application
    - All system calls essentially involve the “INT 0x80” on x86 processors + Linux
    - Different arguments specified in EAX register inform the kernel about different system calls
- The C library has wrapper functions for each of the system calls
  - E.g. open(), read(), write(), fork(), mmap(), etc.

Types of System Calls

- File System Related
  - Open(), read(), write(), close(), etc.
- Processes Related
  - Fork(), exec(), ...
- Memory management related
  - Mmap(), shm_open(), ...
- Device Management
- Information maintainance – time,date
- Communication between processes (IPC)
- Read man syscalls

https://linuxhint.com/list_of_linux_syscalls/

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File Manipulation	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device Manipulation	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	write()
Information Maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communication	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

Code schematic

```
int main() {
    int a = 2;
    printf("hi\n");
}
-----
C Library
-----
int printf("void *a, ...) {
    ...
    write(1, a, ...);
}
int write(int fd, char *, int len) {
    int ret;
    ...
    mov $5, %eax,
    mov ... %ebx,
    mov ..., %ecx
    int $0x80
    __asm__("movl  %eax, -4(%ebp)");
# -4ebp is ret
    return ret;
}
```

```
-------user-kernel-mode-boundary----
//OS code
int sys_write(int fd, char *, int len) {
    figure out location on disk
    where to do the write and
    carry out the operation,
    etc.
}
```

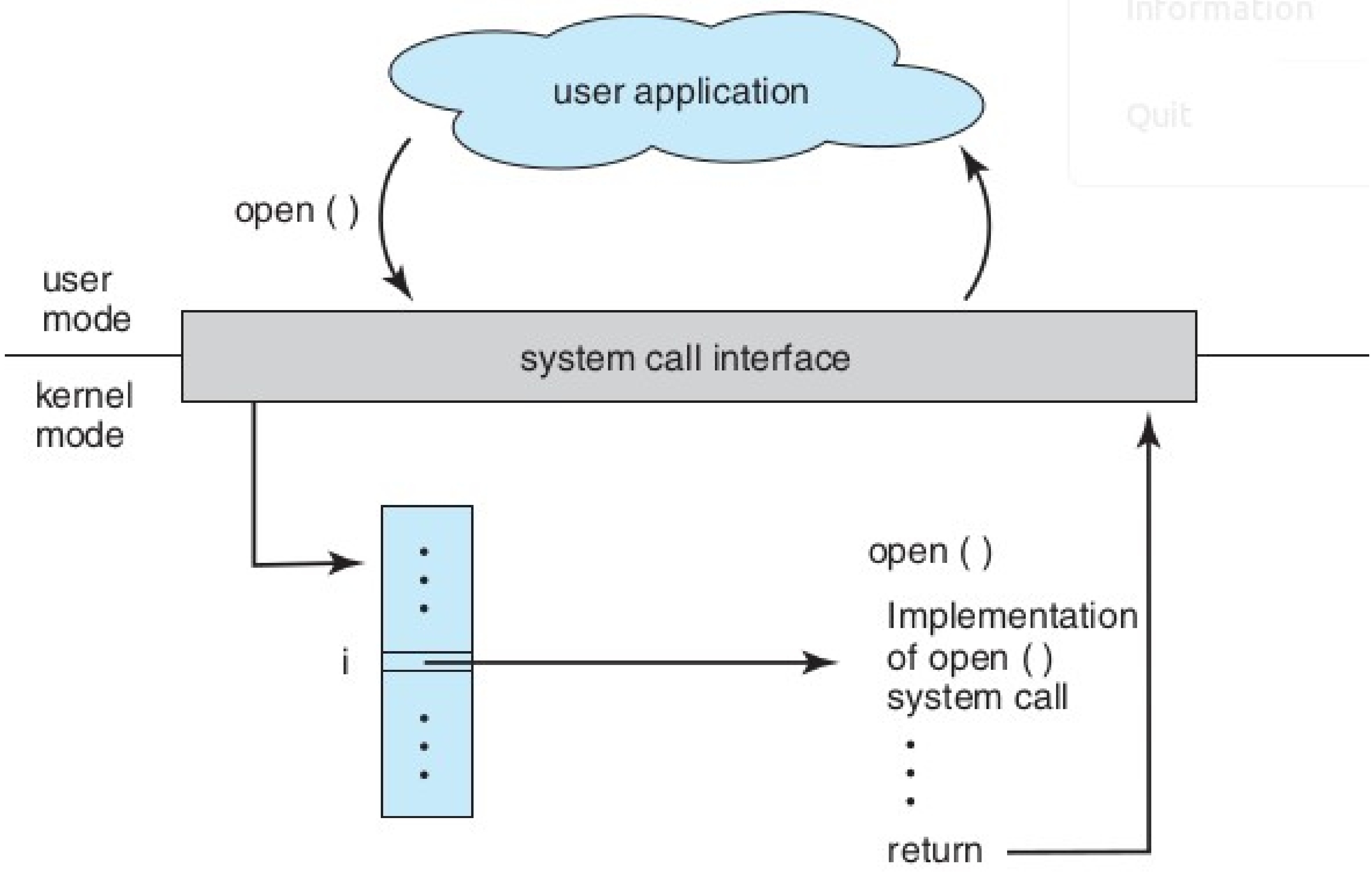


Figure 2.6 The handling of a user application invoking the `open()` system call.

Process

- A program in execution
- Exists in RAM
- Scheduled by OS
    - In a timesharing system, intermittantly scheduled by allocating a time quantum, e.g. 20 microseconds
- The “ps” command on Linux

Process in RAM

- **Memory is required to store the following components of a process**
  - **Code**
  - **Global variables (data)**
  - **Stack (stores local variables of functions)**
  - **Heap (stores malloced memory)**
  - **Shared libraries (e.g. code of printf, etc)**
  - **Few other things, may be**

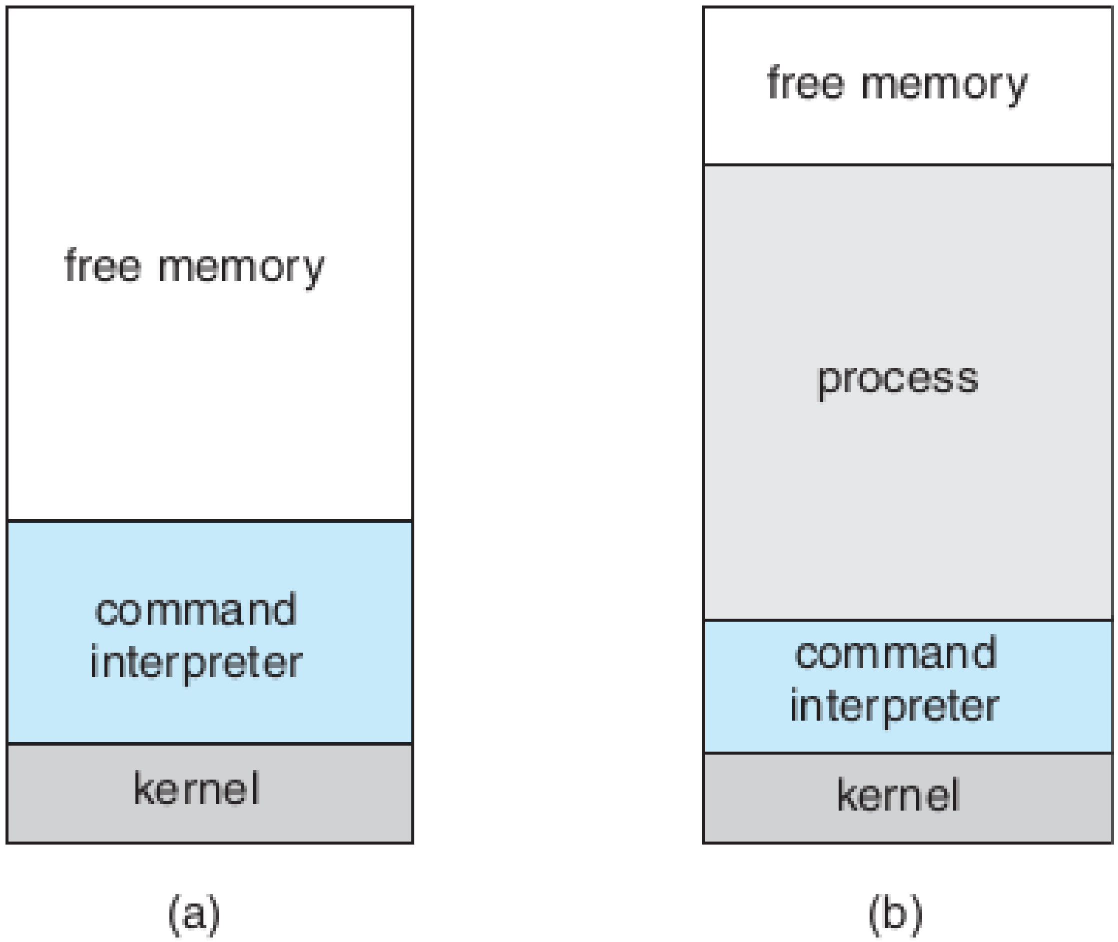
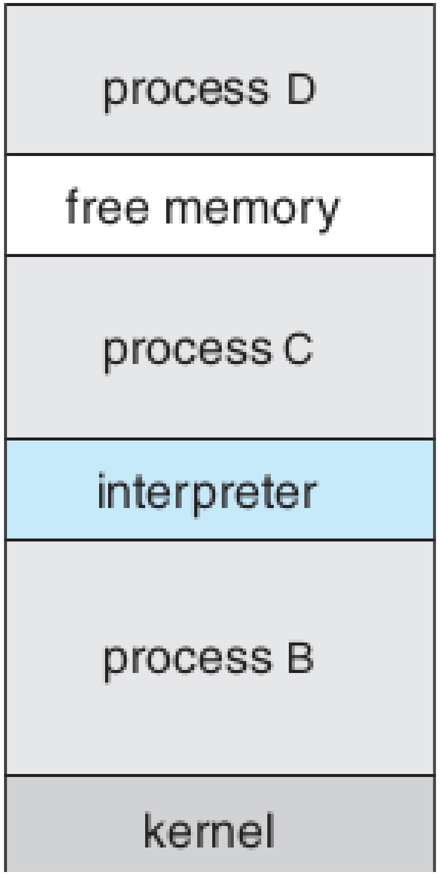


Figure 2.9 MS-DOS execution. (a) At system startup. (b) Running a program.

MS-DOS: a single tasking operating system
Only one program in RAM at a time, and only one program can run at a time



A multi tasking system
With multiple programs loaded in memory
Along with kernel
(A very simplified conceptual diagram. Things are more complex in reality)

fork()

- A running process creates it’s duplicate!
- After call to fork() is over
  - Two processes are running
  - Identical
  - The calling function returns in two places!
  - Caller is called parent, and the new process is called child
  - PID is returned to parent and 0 to child

exec()

- **Variants: execvp(), execl(), etc.**
- **Takes the path name of an executable as an argument**
- **Overwrites the existing process using the code provided in the executable**
- **The original process is OVER ! Vanished!**
- **The new program starts running overwritting the existing process!**

Let's Understand fork() and exec()

```
#include <unistd.h>

int main() {

        fork();

        printf("hi\n");

        return 0;

}
```

```
#include <unistd.h>

int main() {

        printf("hi\n");

        execl("/bin/ls", "ls", NULL);

        printf("bye\n");

        return 0;

}
```

A simple shell

```
#include <stdio.h>
#include <unistd.h>
int main() {
        char string[128];
        int pid;
        while(1) {
                printf("prompt>");
                scanf("%s", string);
                pid = fork();
                if(pid == 0) {
                        execl(string, string, NULL);
                } else {
                        wait(0);
                }
        }
}
```

Shell using fork and exec

- Demo
- The only way a process can be created on Unix/Linux is using fork() + exec()
- All processes that you see were started by some other process using fork() + exec() , except the initial *"init" process*
- *When you click on "firefox" icon, the user-interface program does a fork() + exec() to start firefox; same with a command line shell program*
- The "bash" shell you have been using is nothing but an advanced version of the shell code shown during the demo
- See the process tree starting from "init"
- Your next assignment

The boot process

- BIOS/UEFI
  - The firmware. Runs "Automatically".
  - CPU is hardwired to start running instructions stored a a fixed address.
  - The Motherboard manufacturers, ensure that the BIOS/UEFI is at his address
- Boot loader
  - BIOS reads the code from sector 0 of the "Boot device", this is called "Boot Loader", and loads it in RAM
  - PC changed to Boot Loader code
  - GRUB is an example
- kernel
  - The Boot Loader shows the choice of running an OS, user selects a choice
  - The Boot loader loads the code of kernel from disk into RAM
  - PC changed to kernel code
- Kernel initializes hardware, it's own data structures, etc etc
- Init created by kernel by Hand
- Kernel schedules init (the only process)
- Init fork-execs some programs (user mode)
  - Now these programs will be scheduled by OS
- Init -> GUI -> terminal -> shell
  - One of the typical parent-child relationships

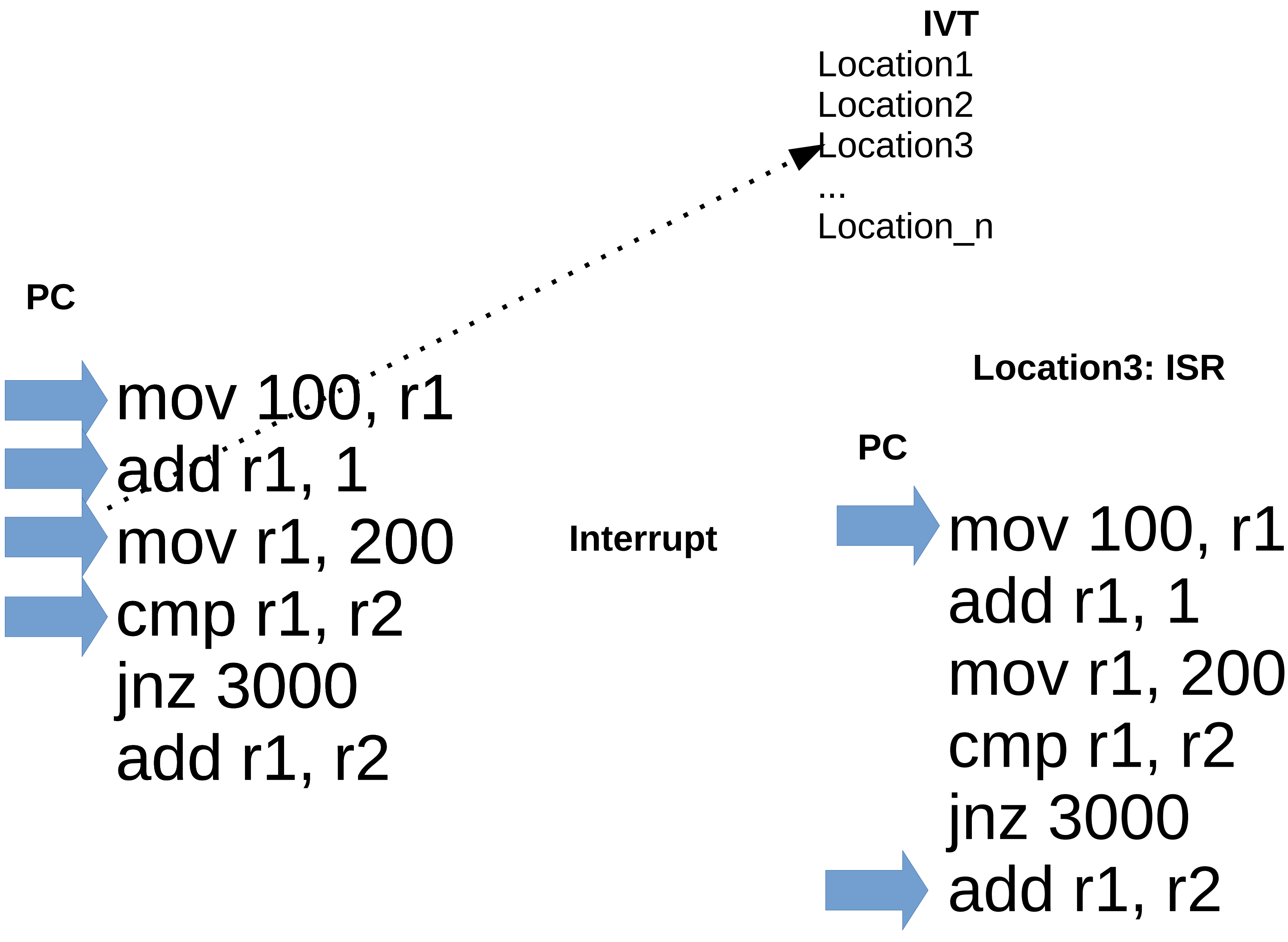
Event Driven kernel
Multi-tasking, Multi-programming

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 ?(in a few minutes)


IVT
Location1
Location2
Location3
...
Location_n
PC
mov 100, r1
add r1, 1
mov r1, 200
cmp r1, r2
jnz 3000
add r1, r2
Interrupt
Location3: ISR
PC
mov 100, r1
add r1, 1
mov r1, 200
cmp r1, r2
jnz 3000
add r1, r2

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!
  - If there is an IVT in the hardware, OS sets up the IVT also!
  - 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

- Timesharing happens 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
- How is this done?
  - kernel sets up the “timer register”
  - kernel changes PC to address of a process (now process runs)
    - Timer register decremented automatically in hardware
  - When timer is “up”, a hardware interrupt occurs!
  - Now OS runs again! This OS code is called “scheduler”

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. Exec() does this.
  - 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**
  - **Timer interrupt used to achieve this.**

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 (given by IVT)**
  - **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 kernel code occupies all memory locations corresponding to the PC values related to all the above events, at the time of boot! It also sets up the IVT if there is one.**

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 privileged 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 jumping to kernel code, using IVT!

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 !

Code schematic

```
int main() {
    int a = 2;
    printf("hi\n");
}
-----
C Library
-----
int printf("void *a, ...) {
    ...
    write(1, a, ...);
}
int write(int fd, char *, int len) {
    int ret;
    ...
    mov $5, %eax,
    mov ... %ebx,
    mov ..., %ecx
    int $0x80
    __asm__("movl  %eax, -4(%ebp)");
# -4ebp is ret
    return ret;
}
```

```
-------user-kernel-mode-boundary----
//OS code
int sys_write(int fd, char *, int len) {
    figure out location on disk
    where to do the write and
    carry out the operation,
    etc.
}
```

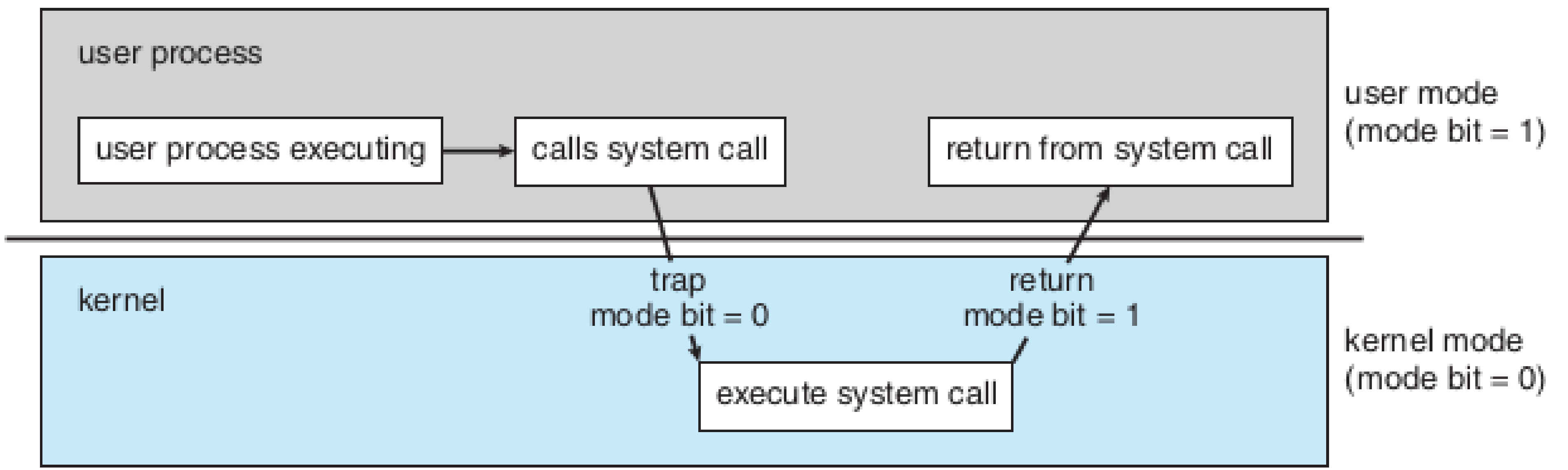


Figure 1.10 Transition from user to kernel mode.

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

What runs on the processor ?

- **4 possibilities.**