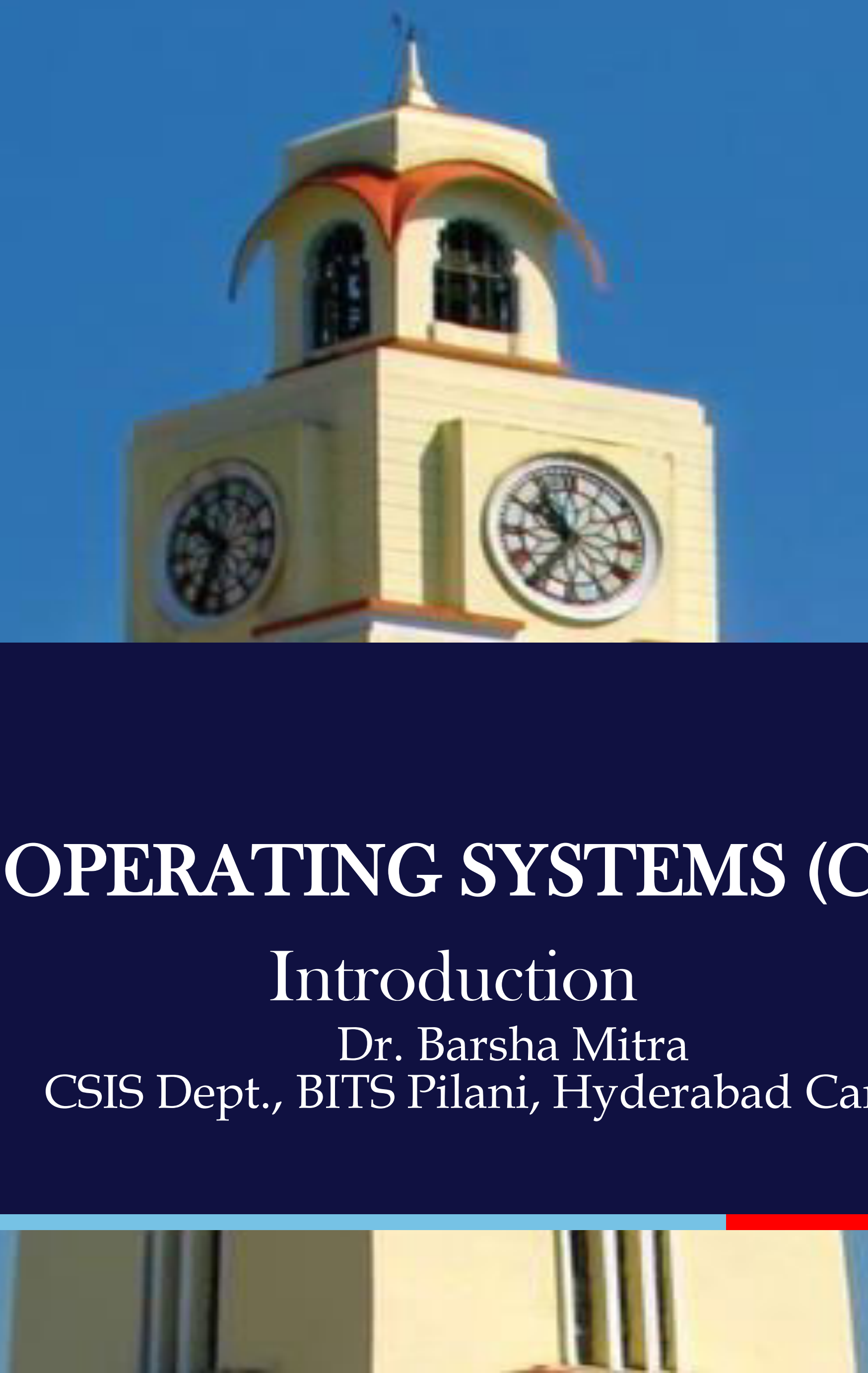


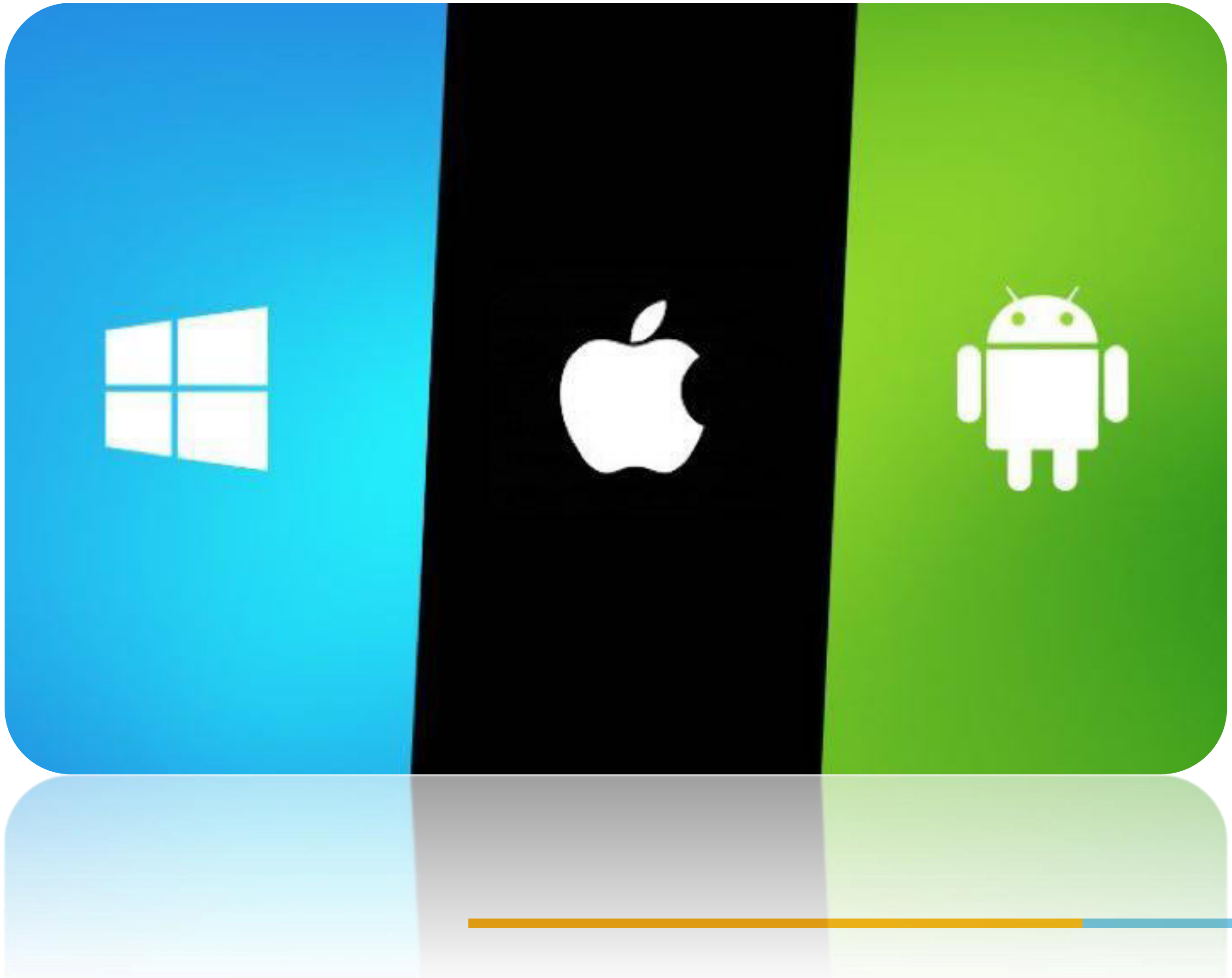
BITS Pilani
Hyderabad Campus

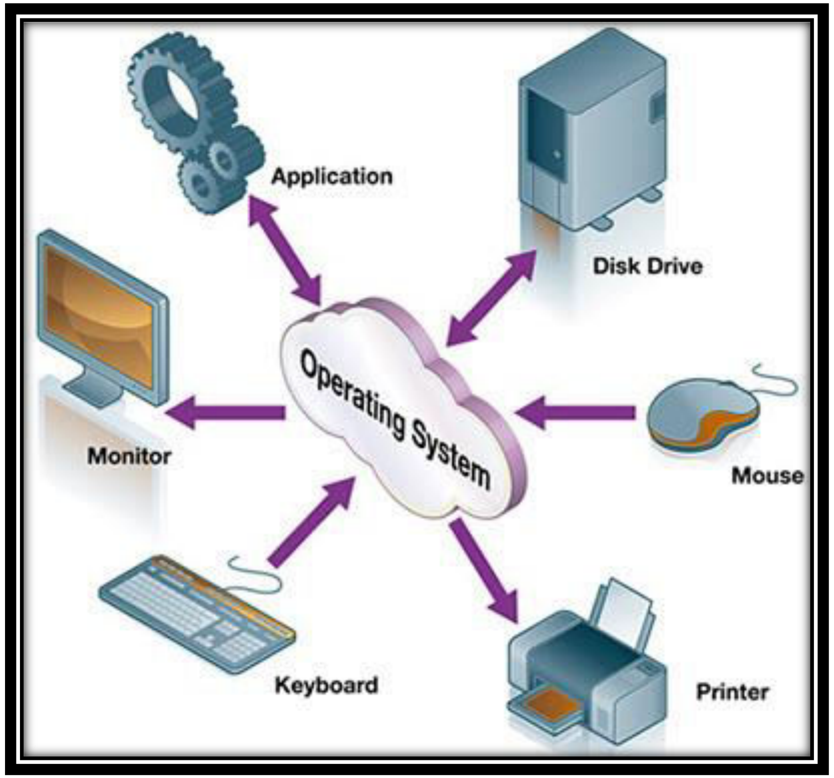
OPERATING SYSTEMS (CS F372)

Introduction

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus

What is an Operating System





Handout Overview

Objectives

- To learn about how process management is carried by the OS. This will include process creation, thread creation, CPU scheduling, process synchronization and deadlocks.
- To learn about memory management carried out by OS. This will include the concepts of paging, segmentation, swapping, and virtual memory.
- To learn how permanent storage like files and disks are managed by OS. This will include topics related to access methods, mounting, disk scheduling, and disk management.
- Hands-on experience

Handout Overview

Text Book:

T1. Silberschatz, Galvin, and Gagne, “Operating System Concepts”, 9th edition, John Wiley & Sons, 2012.

Reference Books:

R1. W. Stallings, “Operating Systems: Internals and Design Principles”, 6th edition, Pearson, 2009.

R2. Tanenbaum, Woodhull, “Operating Systems Design & Implementation”, 3rd edition, Pearson, 2006.

R3. Dhamdhere, “Operating Systems: A Concept based Approach”, 2nd edition, McGrawHill, 2009.

R4. Robert Love, “Linux Kernel Development”, 3rd edition, Pearson, 2010.

Topics to be covered

- Introduction
- OS Structures
- Processes
- Threads
- CPU Scheduling
- Process Synchronization
- Deadlocks
- Main Memory Management
- Virtual Memory
- Mass Storage
- File System Interface
- File System Implementation
- I/O Systems
- Protection

Evaluation

Component	Duration	Weightage (%)	Date & Time	Nature of Component
Mid Semester Examination	90 minutes	30%	As per Time Table	Open Book
Quiz 1	-	10%	TBA	Open Book
Quiz 2	-	10%	TBA	Open Book
Assignment		15%	TBA	Open Book
Comprehensive Examination	120 minutes	35%	As per Time Table	Open Book

Handout Overview

- Chamber Consultation
- Notices
- Make-up Policy

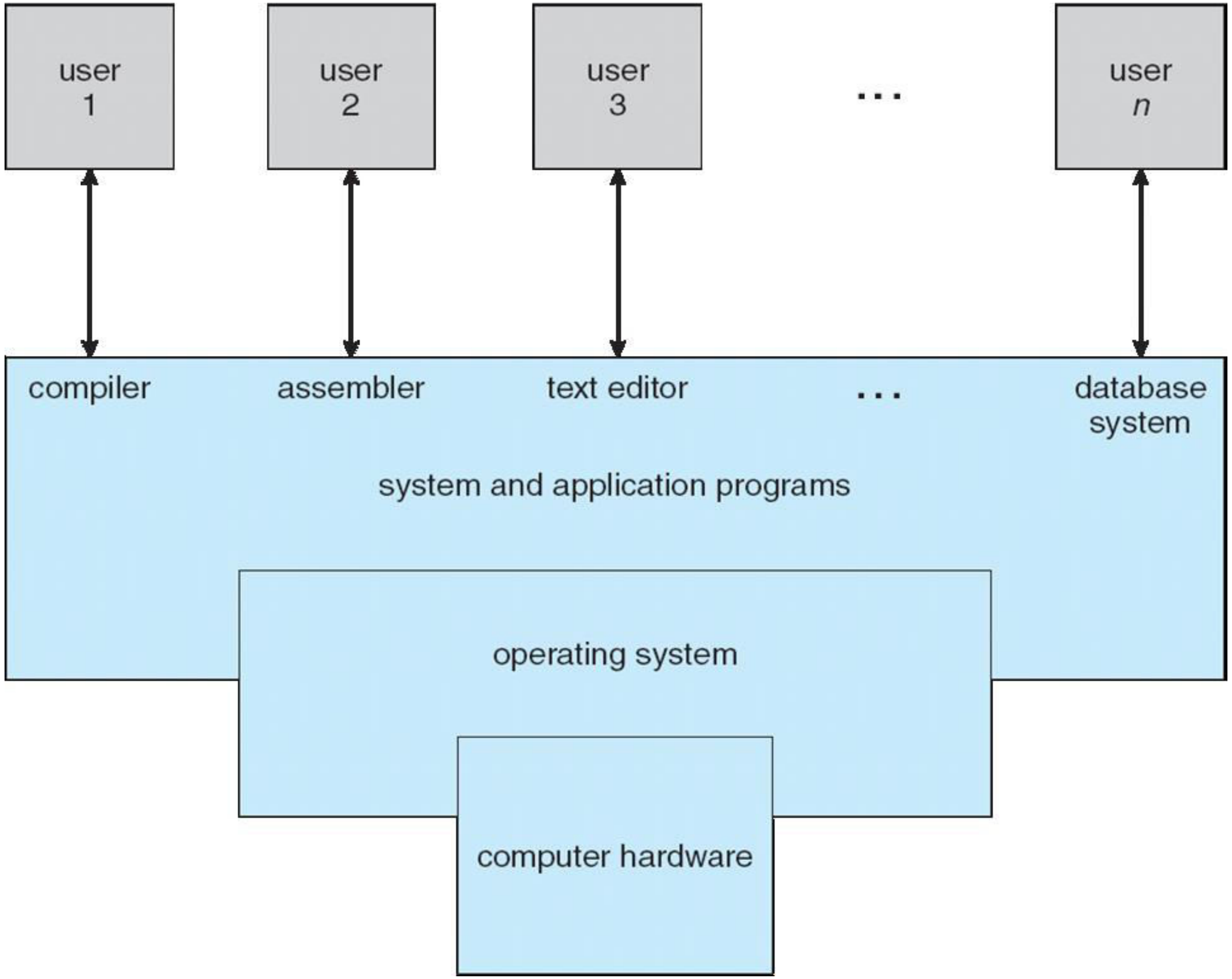
Introduction

- program that manages computer’s hardware
- acts as an intermediary between computer user and computer h/w
- mainframe operating systems
- personal computer (PC) operating systems
- operating systems for mobiles

Computer System Architecture

- Hardware – provides basic computing resources
  - CPU, memory, storage, I/O devices
- Operating system
  - Controls and coordinates use of hardware among various applications and users
- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
  - word processors, email, web browsers, database systems, video games, media player
- Users
  - People, machines, other computers

Computer System Architecture



What OS Does? : User View

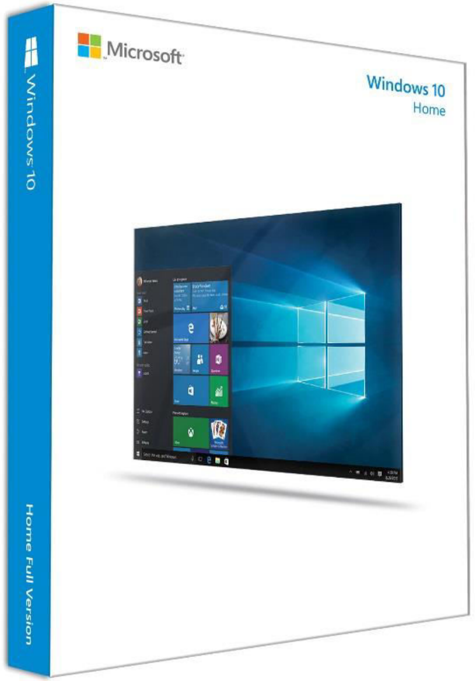
- Users want **convenience**, **ease of use** and **good performance**
  - Don’t care about **resource utilization**
- Shared computer such as **mainframe** or **minicomputer** must keep all users happy
- Users of dedicated systems such as **workstations** have dedicated resources but frequently use shared resources from **servers**
- **Handheld computers** are resource poor, optimized for individual usability and battery life
- Some computers have little or no user interface, such as **embedded computers** in devices and automobiles

What OS Does? : System View

- OS is a **resource allocator**
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
  - Controls execution of user programs to prevent errors and improper use of the computer

How do we define OS?

- ❖ Everything a vendor ships when you order an operating system is a good approximation

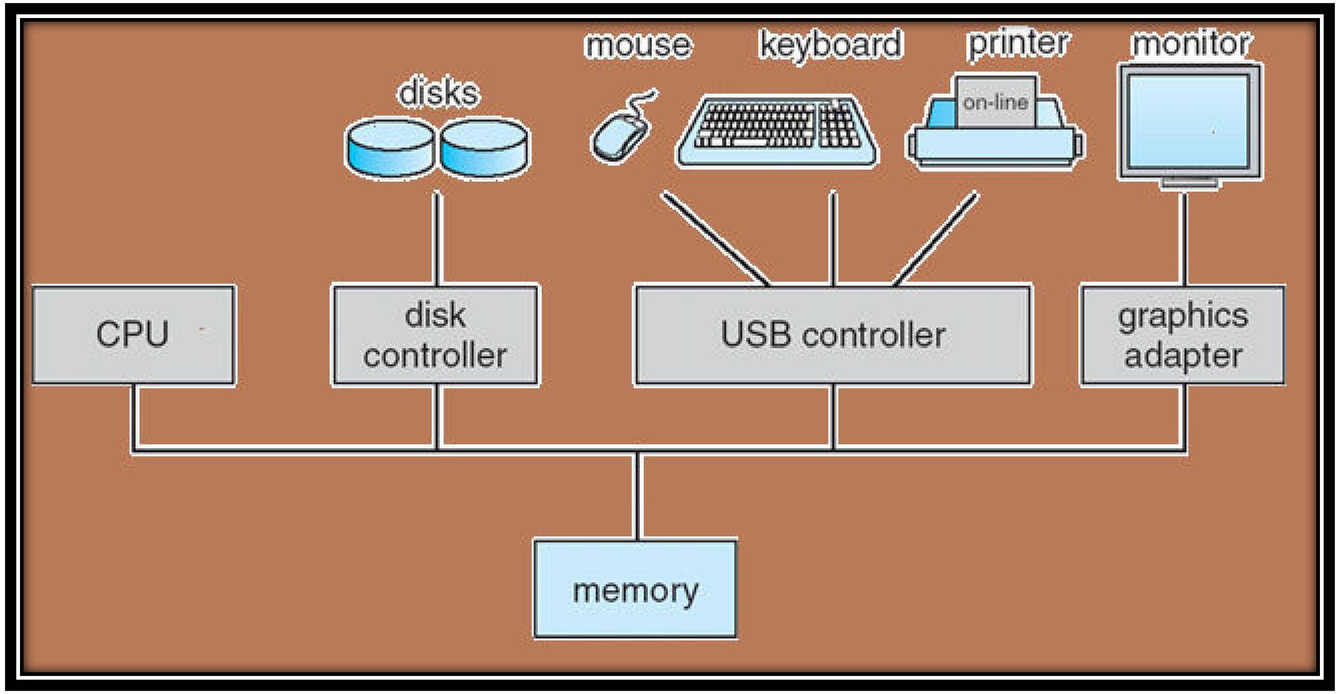


- ❖ “The one program running at all times on the computer”

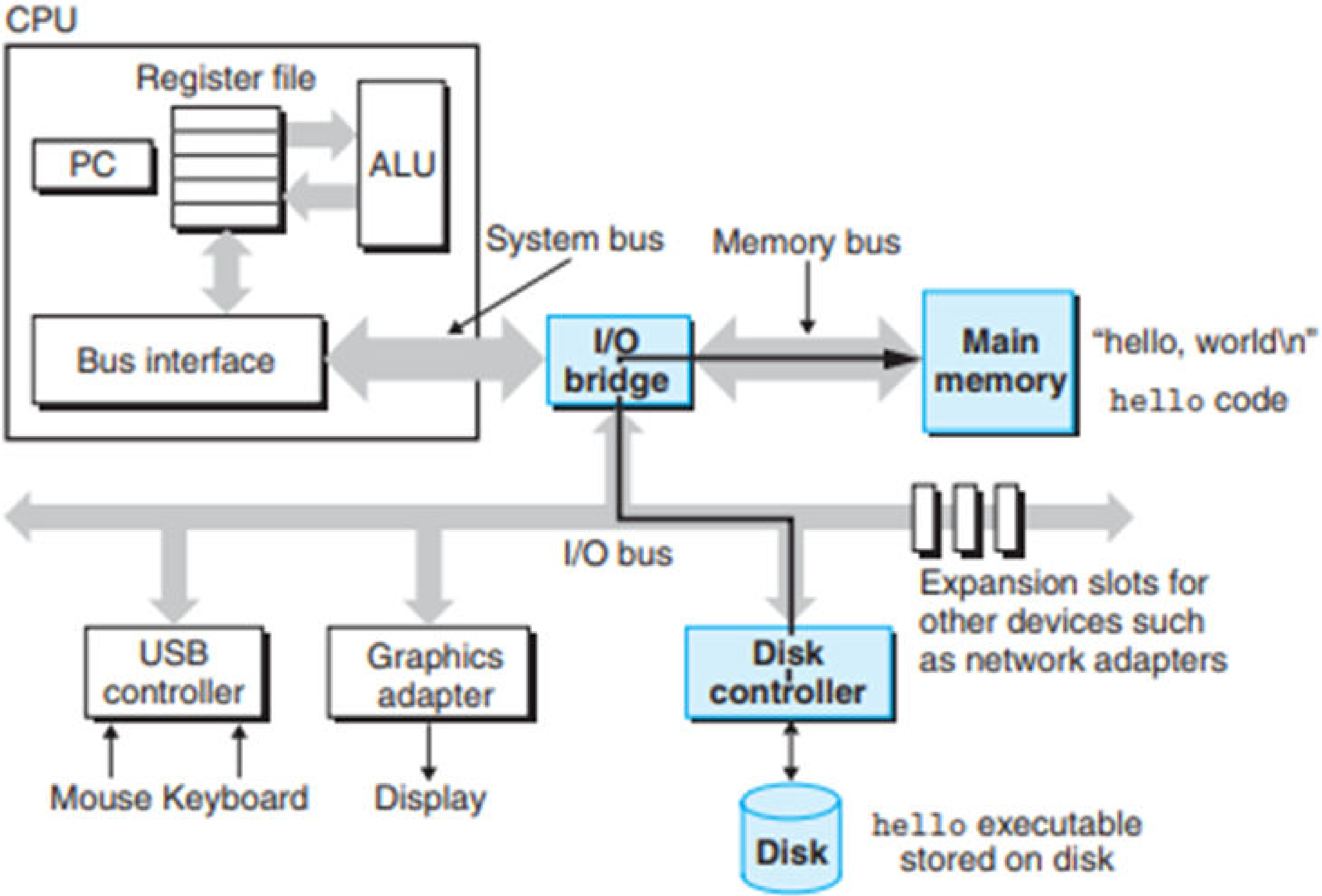
Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles

device controller is a hardware component that works as a bridge between the hardware device and the operating system or an application program



Computer-System Organization



Computer-System Operation

- **Bootstrap program** is loaded at power-up or reboot
  - initial program
  - stored in ROM or EPROM, generally known as **firmware**
  - initializes all aspects of system like CPU registers, device controllers, memory contents
  - locates and loads operating system kernel and starts execution

Computer-System Operation

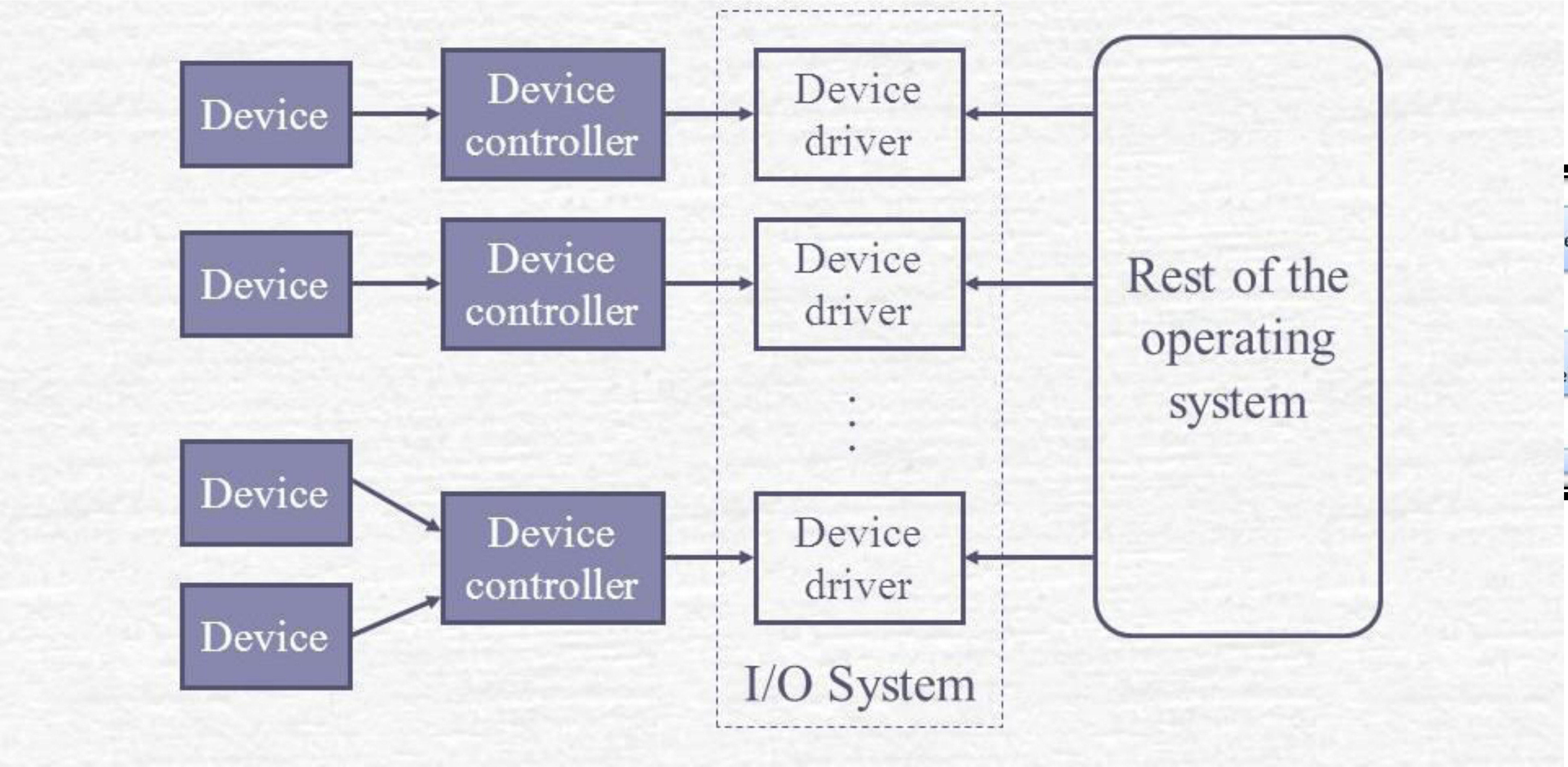
- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an interrupt

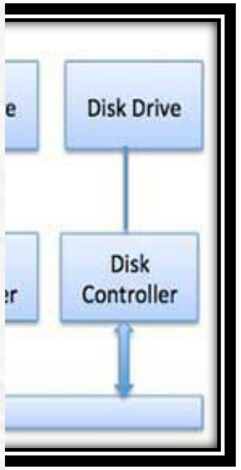
Interrupt Handling

- interrupt transfers control to the interrupt service routine (stored in a fixed location) through the interrupt vector (address for finding ISR)
  - IVT – table of pointers containing the addresses of all the service routines
- must save the address of the interrupted instruction, system stack
- trap/exception is a software-generated interrupt caused either by an error or a user request
- operating system is interrupt driven
- operating system preserves the state of the CPU by storing contents of registers and the program counter

Computer-System Operation

- ❖ co
  co
- ❖ ea
  de
- ❖ de
  - ❖
  - ❖
- ❖ op
  co
- ❖ de
  pr
  uniform interface to the device





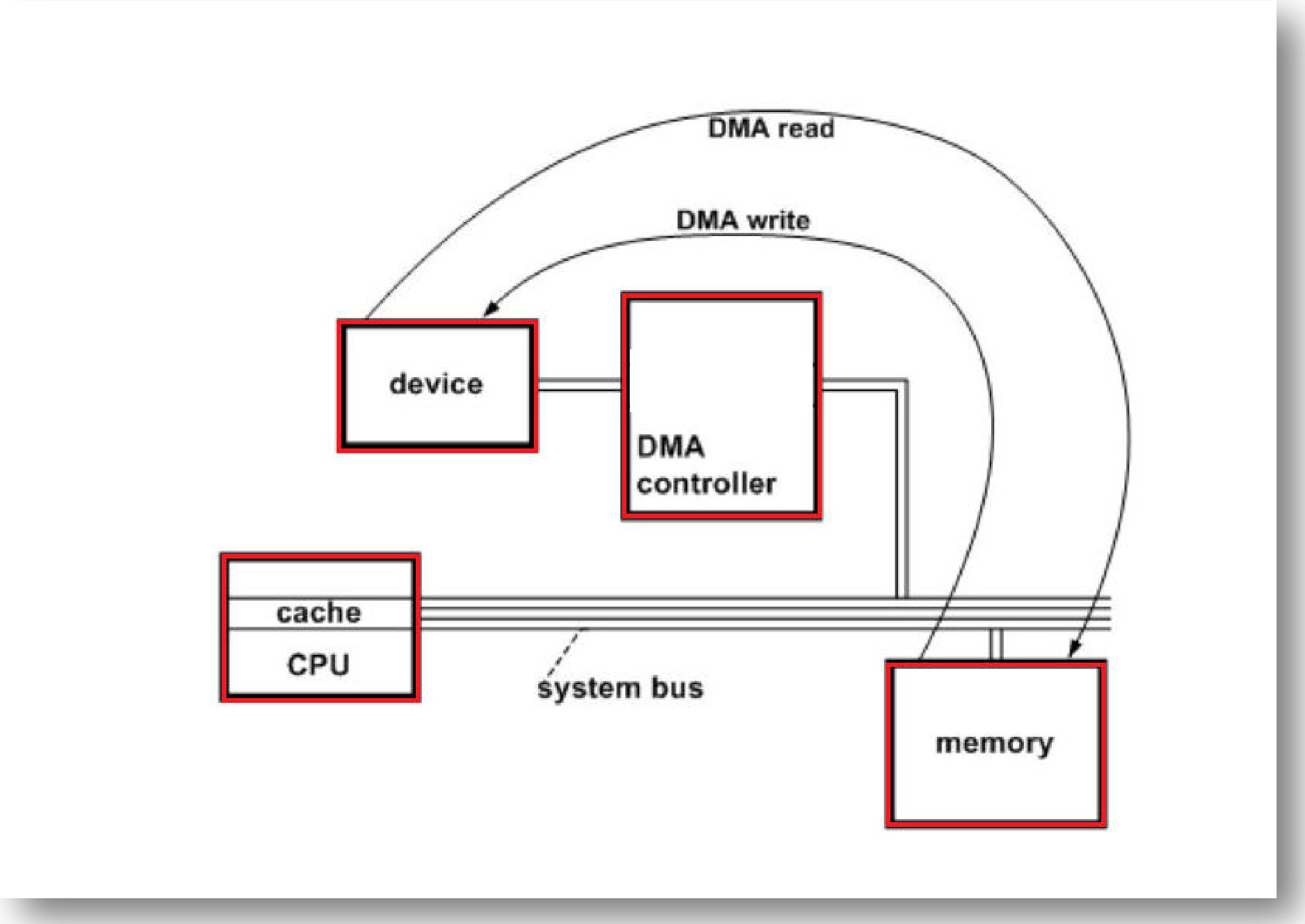
I/O Structure

- to start an I/O operation, the **device driver loads** the registers within the device controller
- **device controller** examines the contents of registers to determine what action to take
- controller starts the transfer of data from the device to its **local buffer**
- device controller informs the **device driver** via an interrupt that it has finished its operation
- device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read
- for other operations, the device driver returns status information

I/O Structure

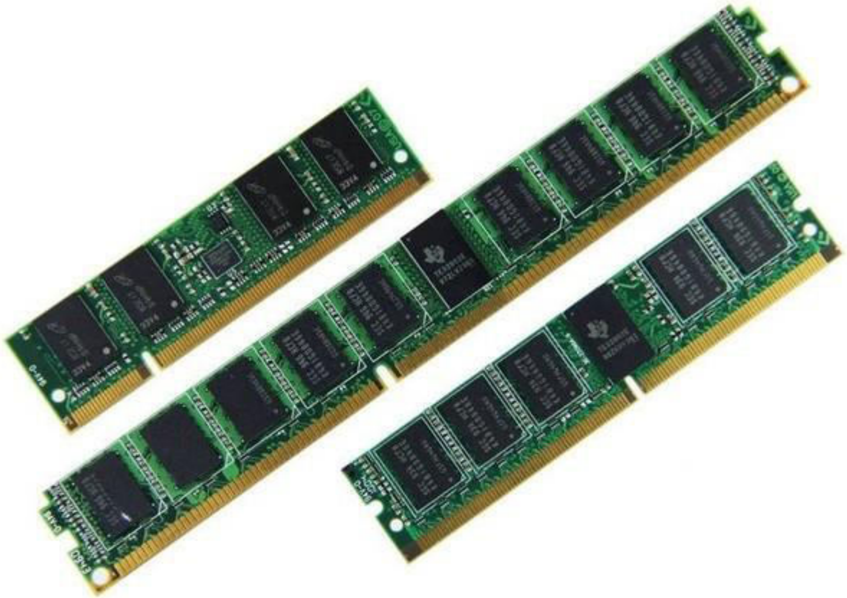
- interrupt-driven I/O is fine for moving small amounts of data
- can produce high overhead when used for bulk data movement such as disk I/O
- **direct memory access (DMA)**
- device controller sets up buffers, pointers, and counters for the I/O device
- device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU
- only one interrupt is generated per block, to tell the device driver that the operation has completed
- instead of one interrupt per byte generated for low-speed devices
- CPU is available to accomplish other work

I/O Structure



Storage Structure

- Main memory –
  - only storage media that the CPU can access directly
  - instruction execution
  - random access
  - volatile

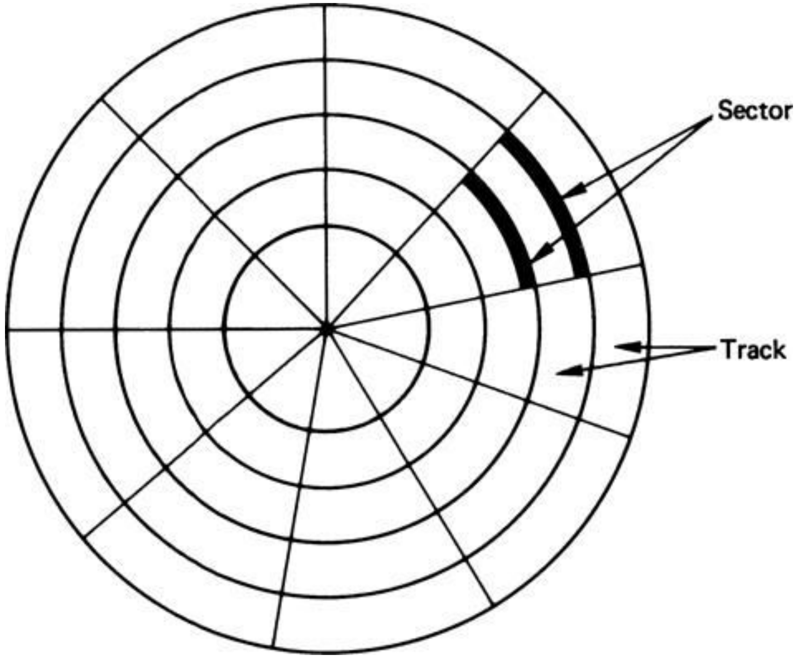


Storage Structure

- Secondary storage –
  - extension of main memory that provides large nonvolatile storage capacity
  - stores both program and data

Storage Structure

- Hard disks/Magnetic disks –
  - rigid metal or glass platters covered with magnetic recording material
  - disk surface is logically divided into tracks, which are subdivided into sectors
  - disk controller determines the interaction between the device and the computer

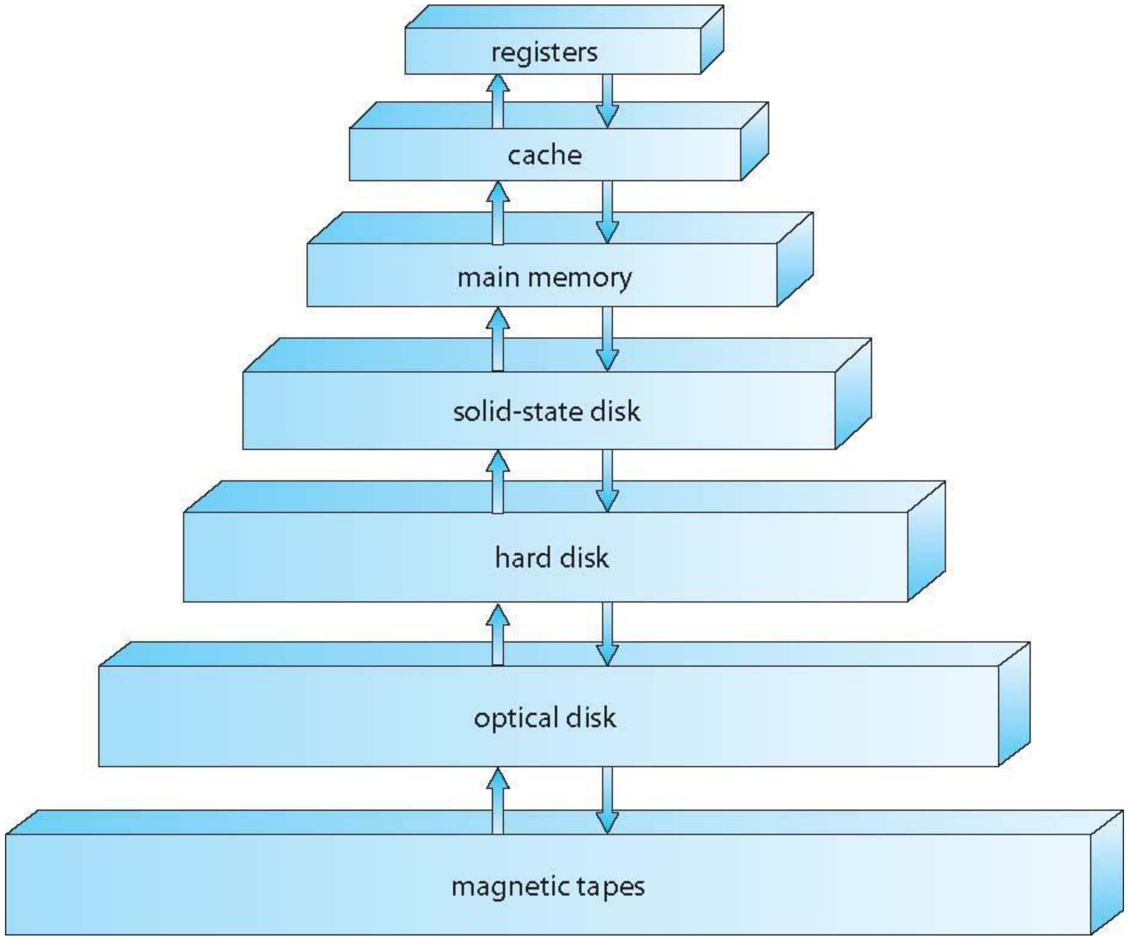


Storage Structure

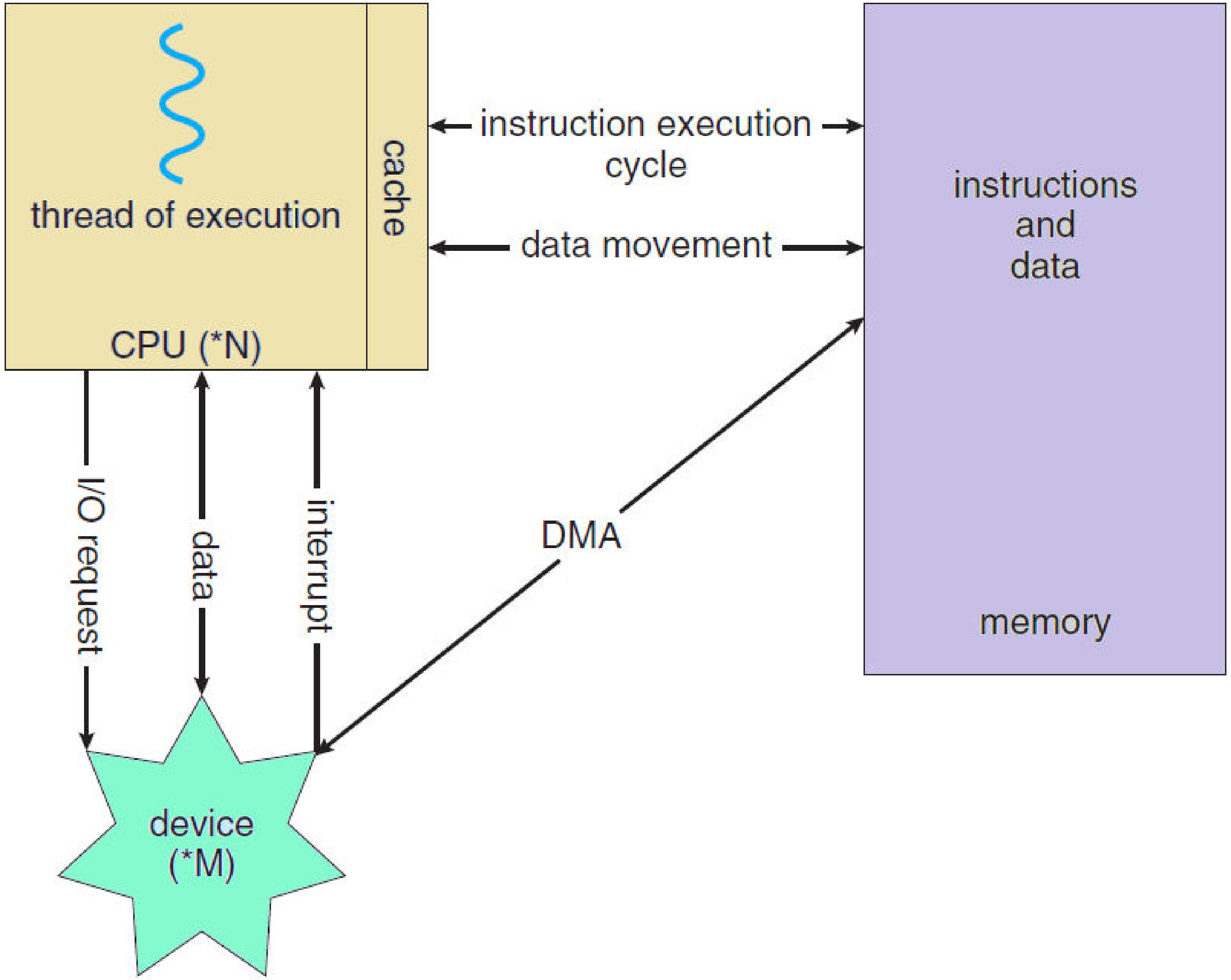
- Solid-state disks –
    - faster than magnetic disks, nonvolatile
    - becoming more popular

- stores data in DRAM during normal operation
- also contains a hidden magnetic hard disk and a battery for backup power
- if external power is interrupted, solid-state disk's controller copies the data from RAM to the magnetic disk
- when external power is restored, the controller copies the data back into RAM
- another form of solid-state disk is flash memory, which is popular in cameras and personal digital assistants (PDAs), slower than DRAM but needs no power to retain its contents

Storage Structure

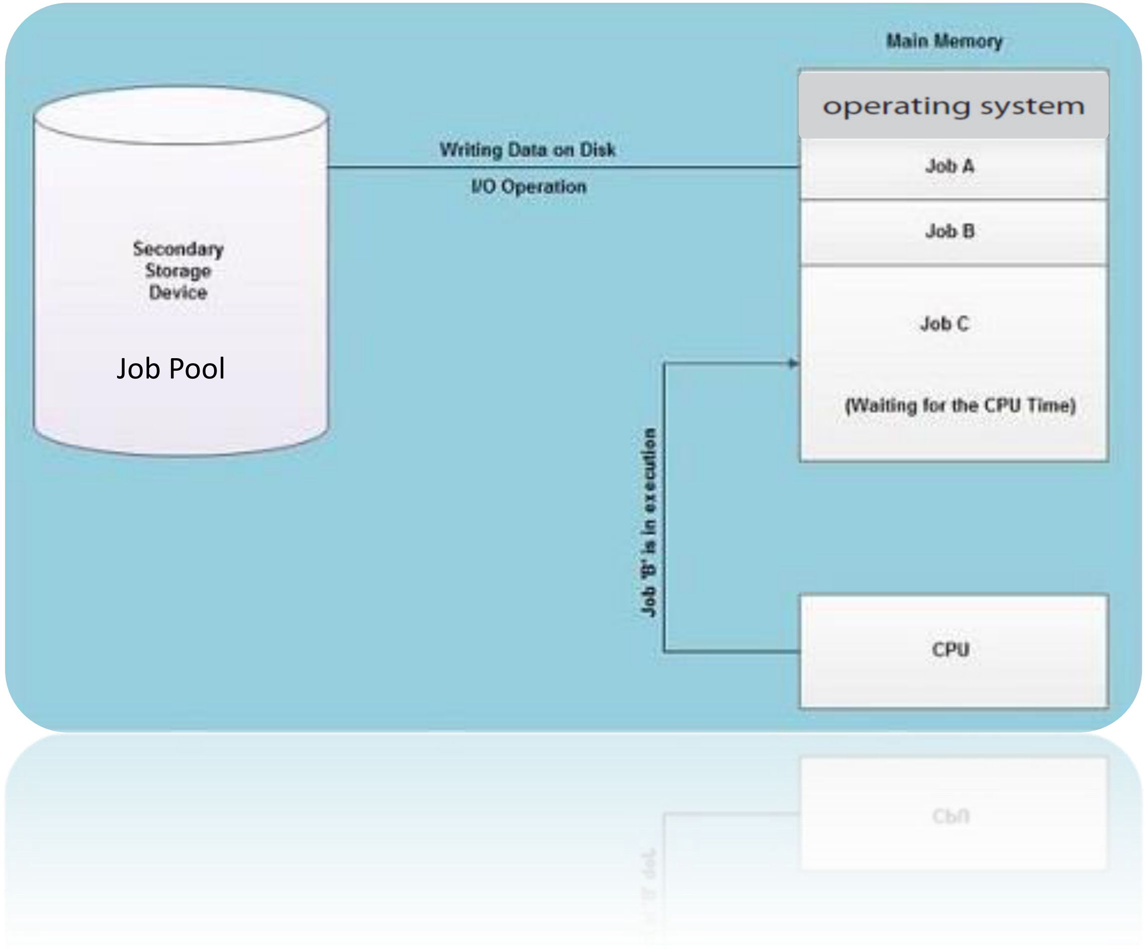


Putting it All Together



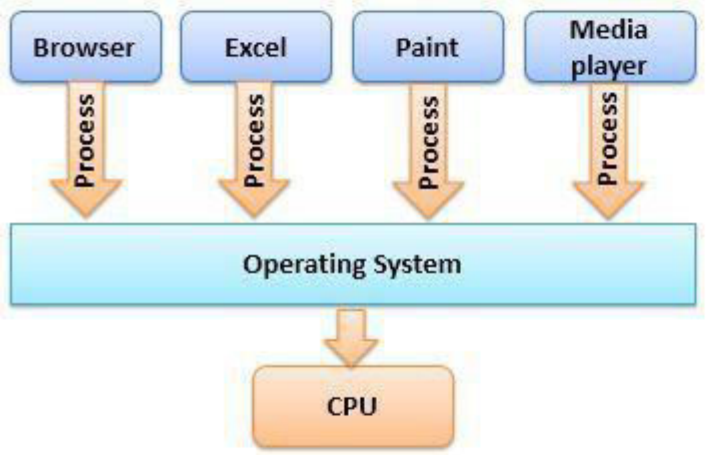
Operating System Structure

- **Multiprogramming (Batch system)**
  - Needed for efficiency
  - Single process cannot keep CPU and I/O devices busy at all times
  - Organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job is selected and run via **job scheduling**
  - When it has to wait for I/O, OS switches to another job



Operating System Structure

- **Timesharing (multitasking):** CPU switches jobs so frequently that users can interact with each job while it is running
- **interactive** computing
  - User interaction via input devices
  - **Response time** should be minimal
  - Each user has at least one program executing in memory ⇨**process**
  - If several processes ready to run at the same time ⇨ **CPU scheduling**
  - If processes don’t fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes larger than physical memory



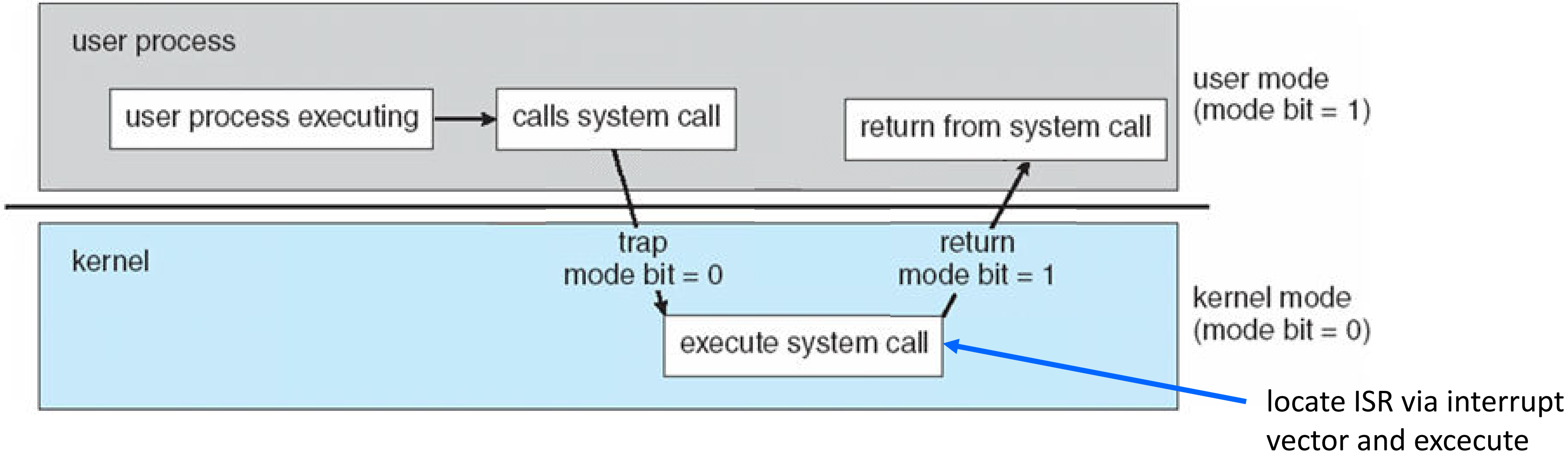
Multitasking

Operating System Operations

- **Interrupt driven** - hardware and software
- Hardware interrupt by one of the devices
- Software interrupt (**exception** or **trap):**
  - Software error (e.g., division by zero, invalid memory access)
  - Request for operating system service
  - Other process problems include infinite loop, processes modifying each other or the operating system

Operating System Operations

- **Dual-mode** operation allows OS to protect itself and protect users from one another
- **User mode** and **Kernel/Supervisor/System/Privileged mode**
- **Mode bit** provided by hardware
  - Provides ability to distinguish when system is running user code or kernel code
  - Some instructions designated as **privileged**, only executable in kernel mode
  - **System call** changes mode to kernel, return from call resets it to user



locate ISR via interrupt vector and excecute

Operating System Operations

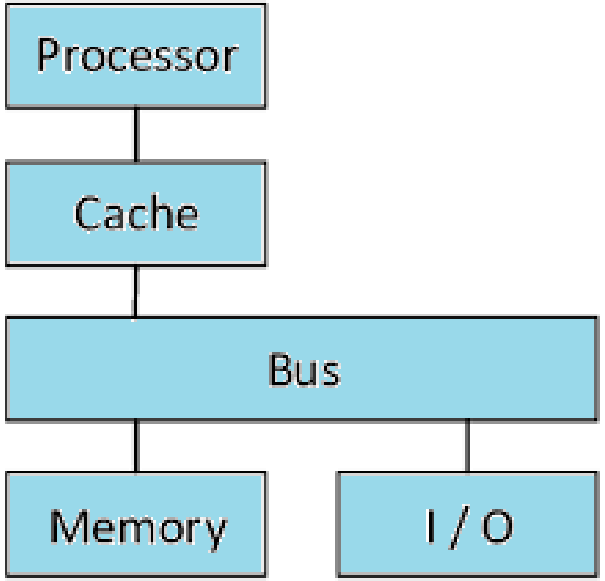
- ❖Boot time → hardware starts in kernel mode
- ❖After loading OS, user applications are started in user mode
- ❖When trap/interrupt occurs, hardware switches from user mode to kernel mode
- ❖examples of privileged instructions – switch to kernel mode, I/O control, timer management, interrupt management

Operating System Operations: Timer

- User processes must return control to OS
- Prevent infinite loop / process hogging resources
- Set to interrupt the computer after some time period
- Keep a counter that is decremented for every physical clock tick
- Operating system sets the counter (privileged instruction) before switching to user mode
- When counter reaches zero, generate an interrupt
- Set up before scheduling process to regain control or terminate program that exceeds allotted time

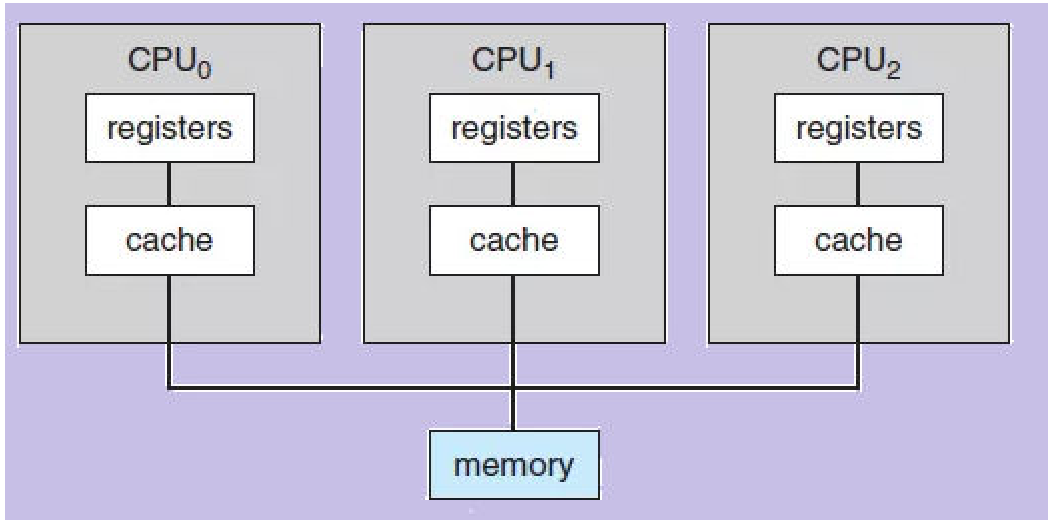
Computing Environments

Single-Processor Systems - one main CPU executing instructions, including instructions from user processes, some device-specific processors like disk, keyboard and graphics controller and I/O processor may be present



Computing Environments

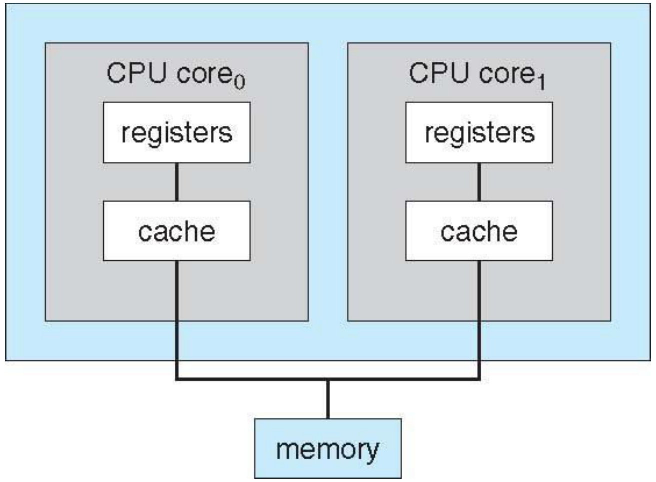
- **Multiprocessors**
  - Also known as **parallel systems**, **multi-core systems**
  - 2 or more processors in close communication, sharing the computer bus and sometimes the clock, memory and peripheral devices
  - Advantages:
    - **Increased throughput**
    - **Economy of scale**
    - **Increased reliability** – graceful degradation, fault tolerance



- Two types:
  - **Asymmetric Multiprocessing** – each processor is assigned a specific task, boss processor controls worker processors
  - **Symmetric Multiprocessing** – each processor performs all tasks, peers

Computing Environments

- **Multicore Systems**
  - include multiple computing cores on a single chip
  - more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication

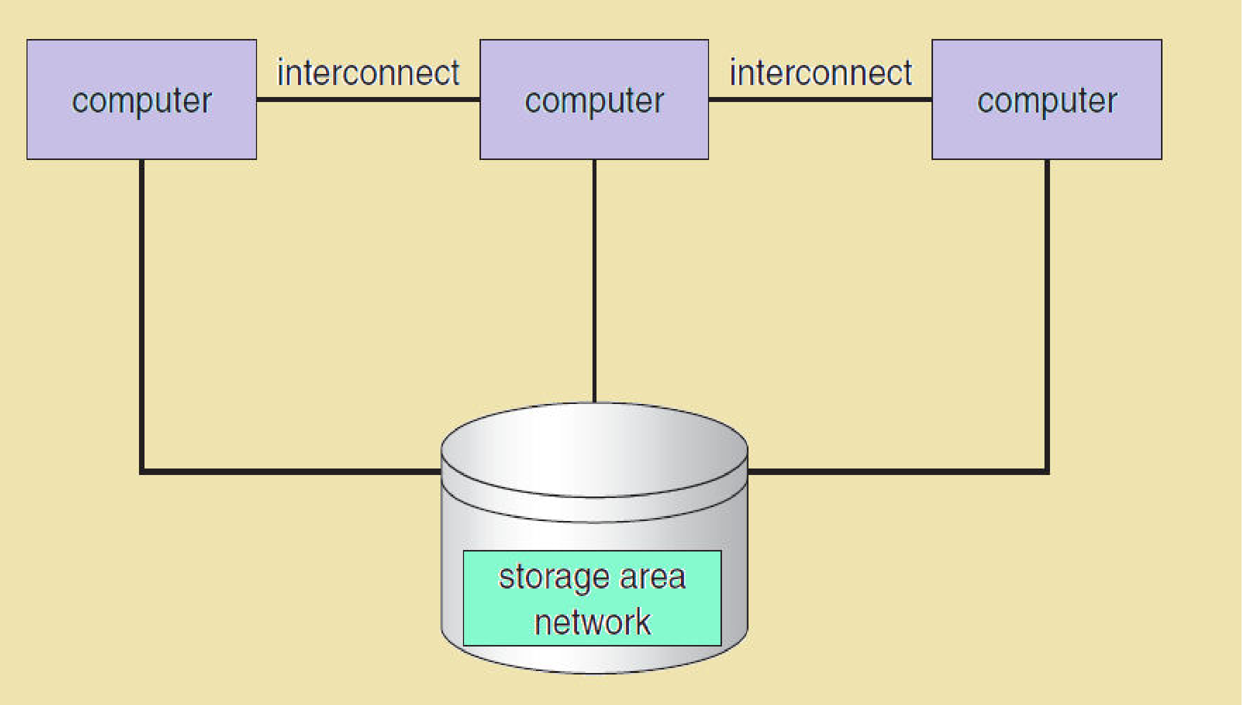


dual-core design with both cores on same chip

Computing Environments

Clustered Systems

- Like multiprocessor systems, but multiple systems working together
- Usually sharing storage via a **storage-area network (SAN)**
- Provides a **high-availability** service which survives failures, users can see only a brief interruption of service
  - **Asymmetric clustering** has one machine in hot-standby mode
  - **Symmetric clustering** has multiple nodes running applications, monitoring each other



- Some clusters are for **high-performance computing (HPC) -** Applications must be written to use **parallelization**

Computing Environments

Traditional Computing

- stand-alone general purpose machines
- most systems interconnect with others (i.e., the Internet)
- mobile computers interconnect via wireless networks
- home computers

Computing Environments

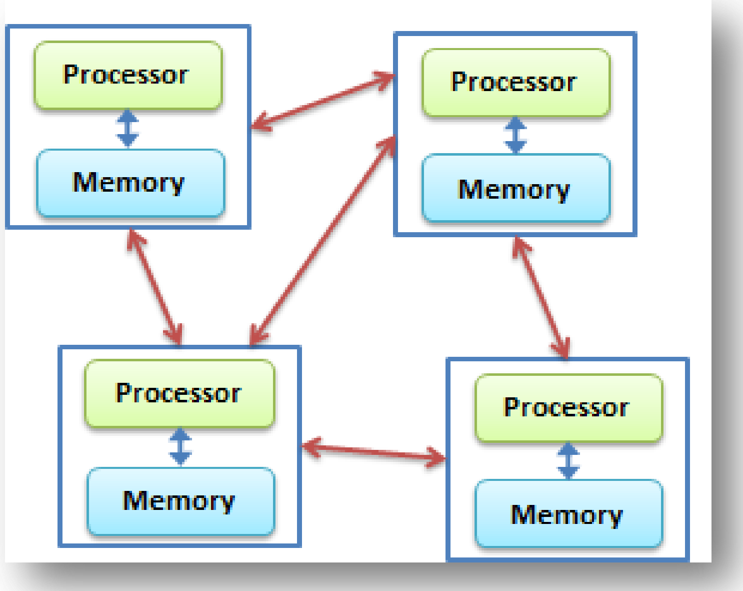
Mobile Computing

- handheld smartphones, tablets, etc.
- portable, lightweight
- allows different types of apps
- use wireless, or cellular data networks for connectivity
- Apple iOS, Google Android

Computing Environments

Distributed Computing

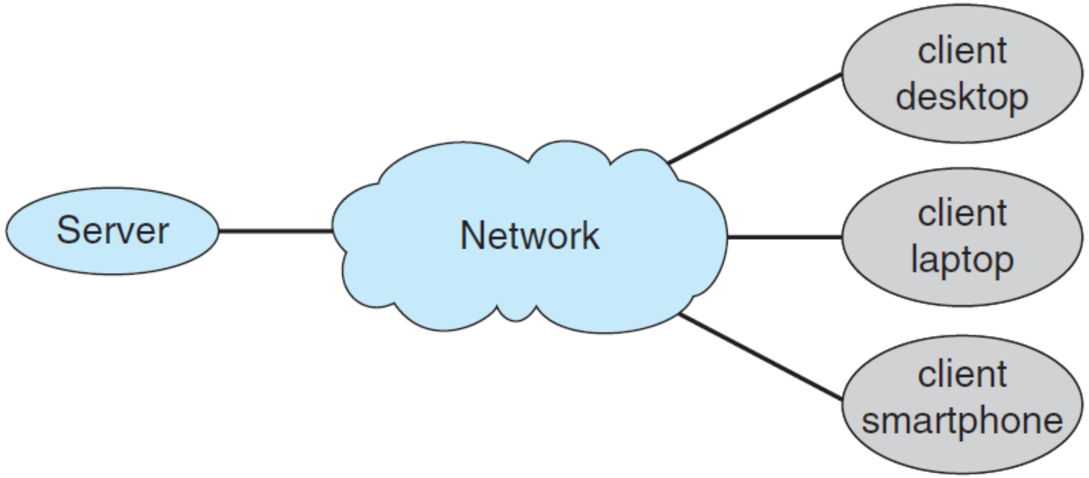
- collection of separate, possibly heterogeneous, systems networked together
- access to shared resources
- network is a communications path
  - Local Area Network (LAN)
  - Wide Area Network (WAN)
  - Metropolitan Area Network (MAN)
  - Personal Area Network (PAN)
- systems exchange messages



Computing Environments

Client Server Computing

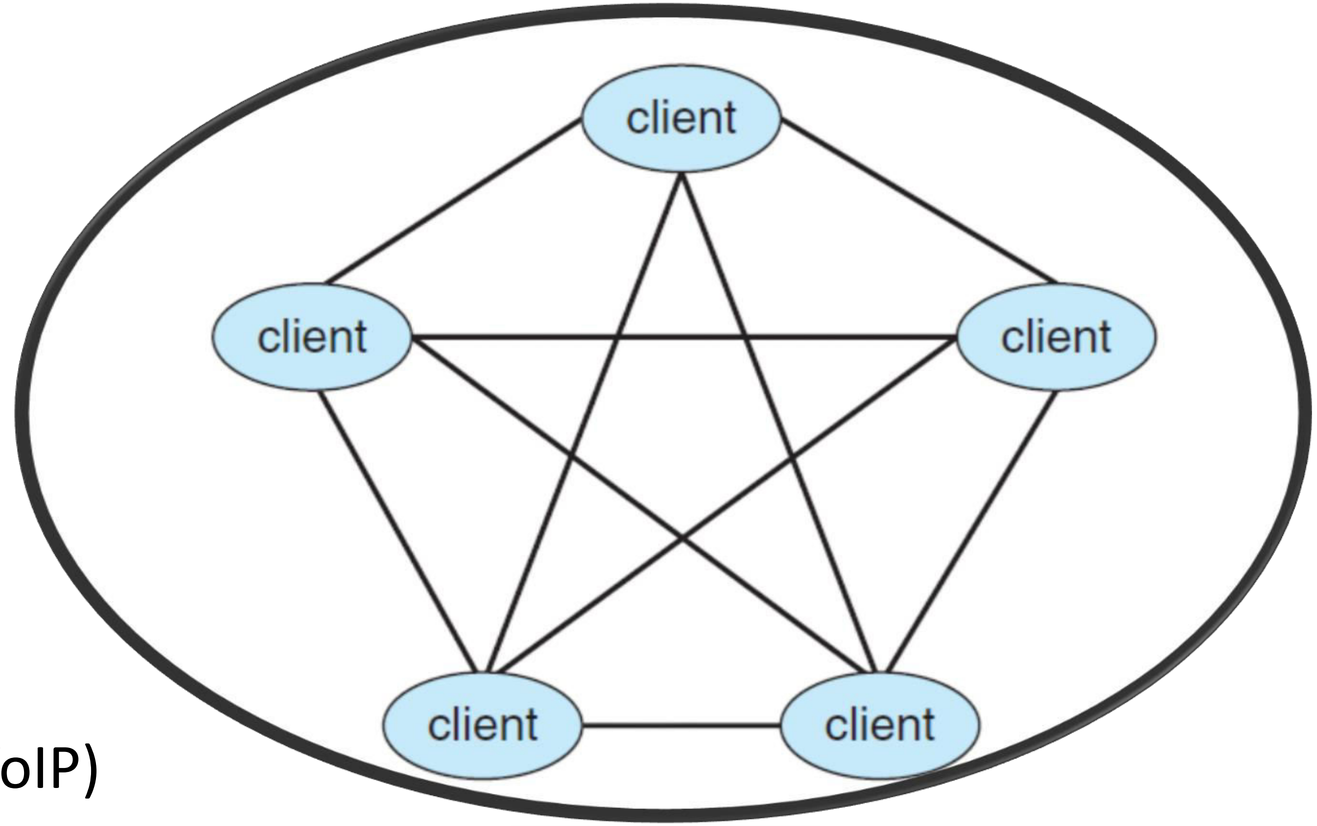
- terminals are PCs and mobile devices
- servers respond to requests generated by clients
- servers can be of 2 types
  - **compute-server** system provides an interface to client to request services, server executes the action and sends the results to clients
  - **file-server system** provides interface for clients to create, update, read and delete files



Computing Environments

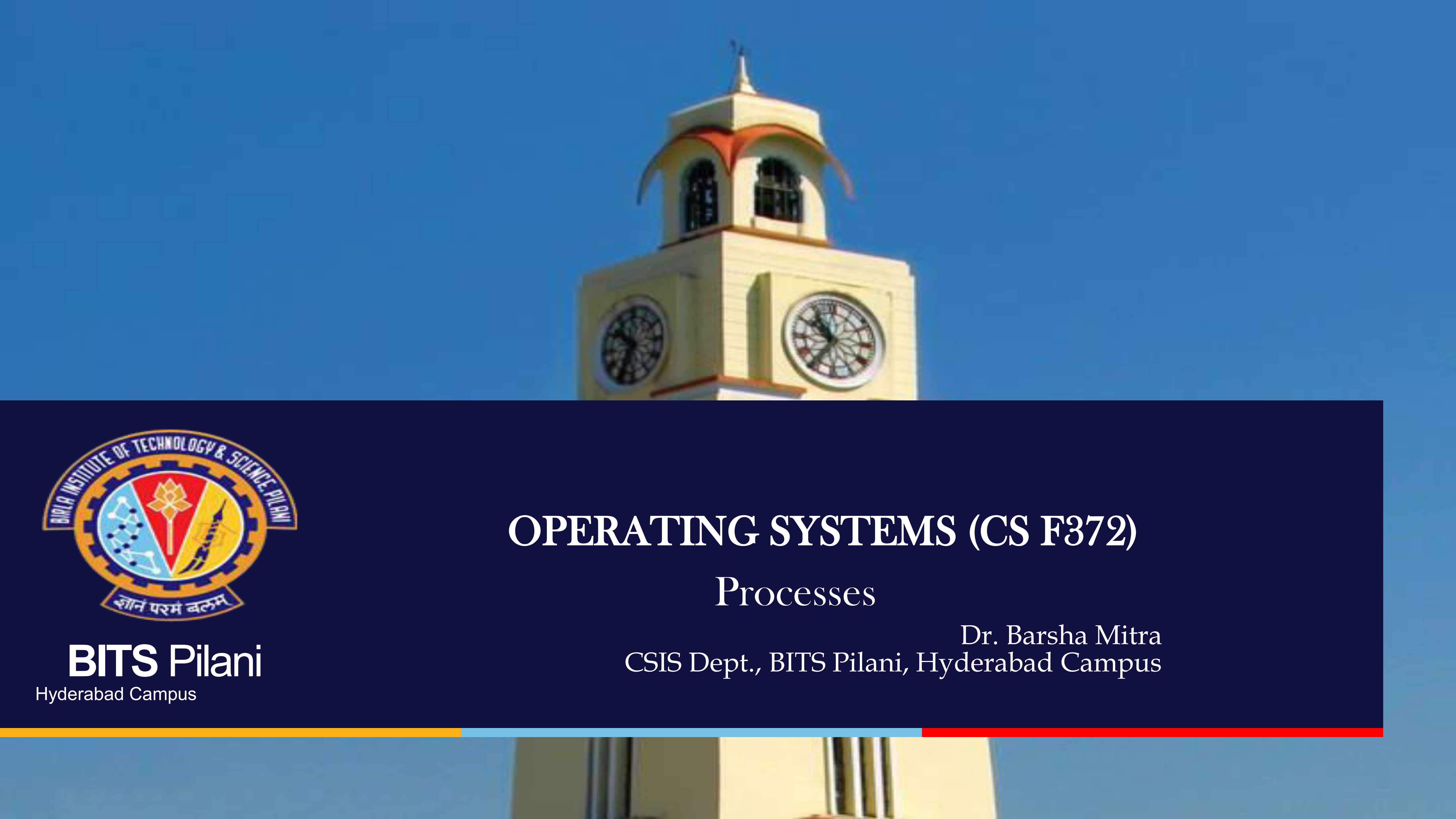
Peer-to-peer Computing

- does not distinguish clients and servers
- nodes join and may also leave P2P network
- advantage over client server system



- Napster, Gnutella, BitTorrent, Skype (VoIP)

Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

Processes

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus

Process Concept

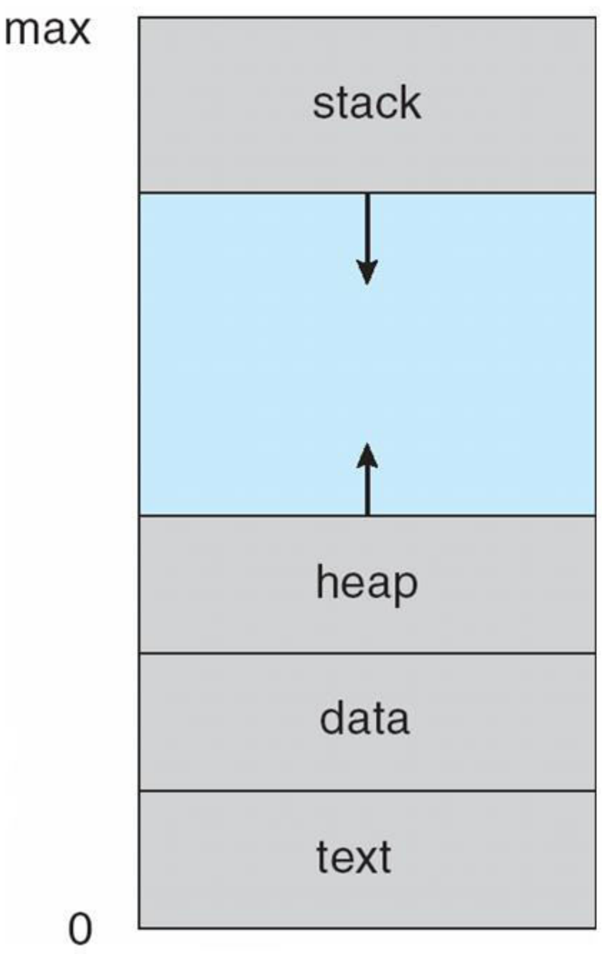
- An operating system executes a variety of programs
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Terms ***job*** and ***process*** used interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion

Process Concept

- Multiple parts
    - The program code, also called **text section**
    - Current activity including **program counter**, processor registers
    - **Stack** containing temporary data
        - Function parameters, return addresses, local variables
    - **Data section** containing global variables
    - **Heap** containing memory dynamically allocated during run time

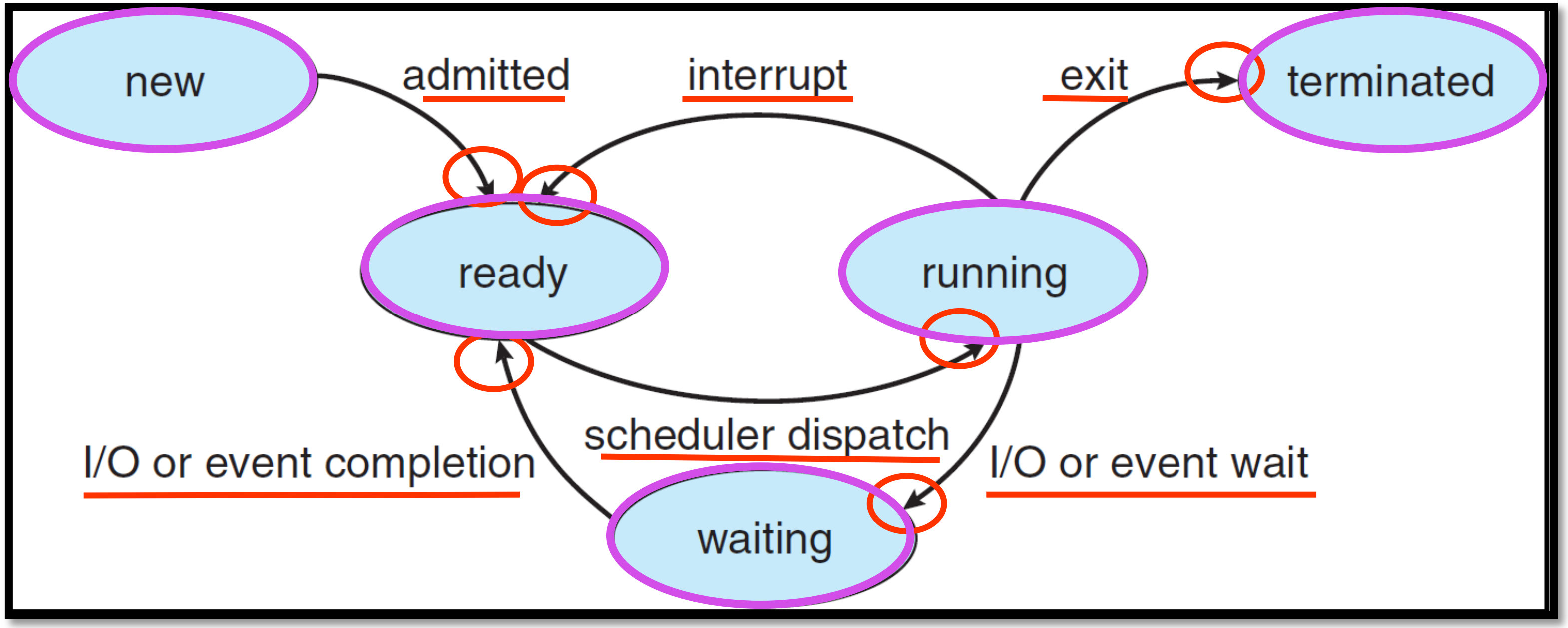
Process Concept

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program



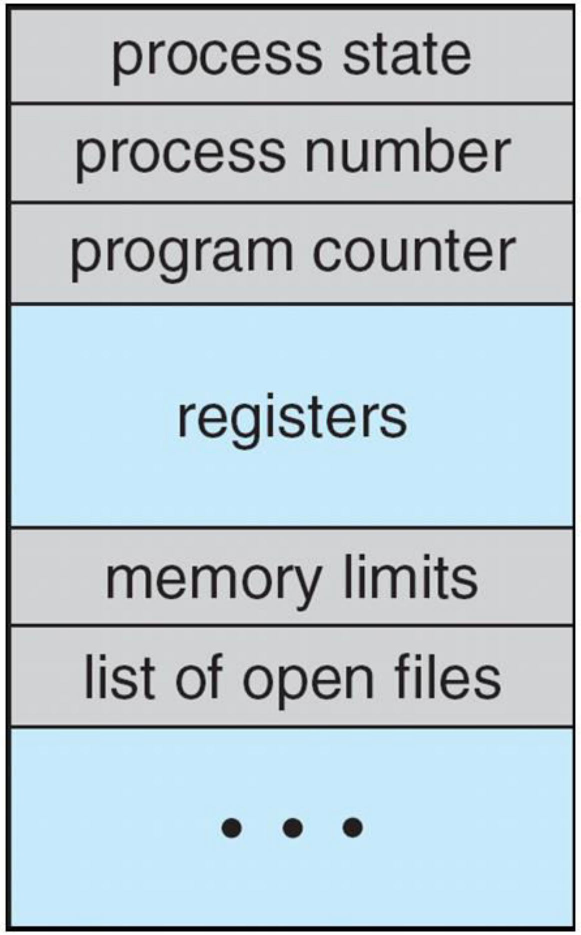
States of Process

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution



Process Control Block

- **Process state** – new, ready, running, waiting, etc.
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information**- priorities, scheduling queue pointers, scheduling parameters
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits, account nos., process nos.
- **I/O status information** – I/O devices allocated to process, list of open files



Process Creation

- **Parent** process creates **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**), integer number
- Resource sharing options (CPU time, memory, files, I/O devices)
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation

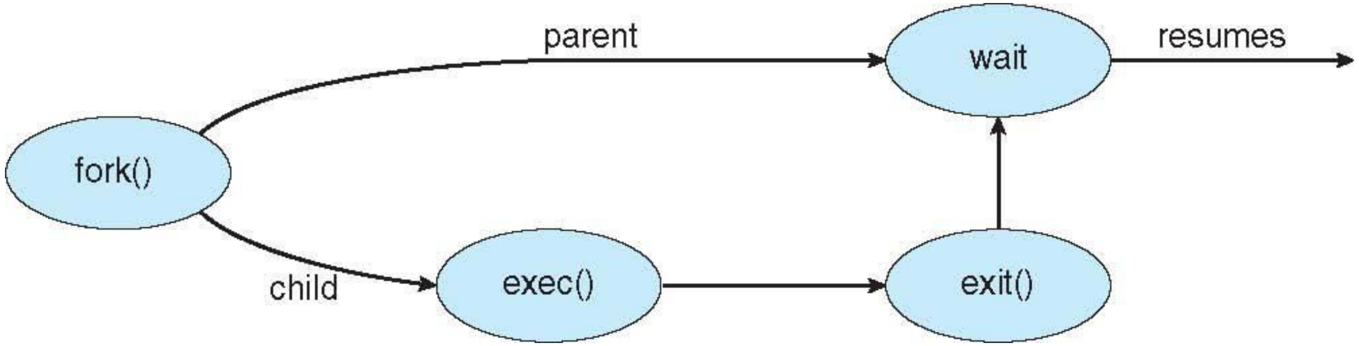
- Address space
  - Child duplicate of parent (same program and data)
  - Child has a program loaded into it
- UNIX examples
  - fork() system call creates new process
  - exec() system call used after a fork() to replace the process’s memory space with a new program

Process Creation

- fork()
  - address space of child process is a copy of parent process
  - both child and parent continue execution at the instruction after fork()
  - return code for fork() is 0 for child
  - return code for fork() is non-zero (child *pid*) for parent
- exec()
  - loads a binary file into memory and starts execution
  - destroys previous memory image
  - call to exec() does not return unless an error occurs
- wait()
  - parent can issue wait() to move out of ready queue until the child is done

Process Creation

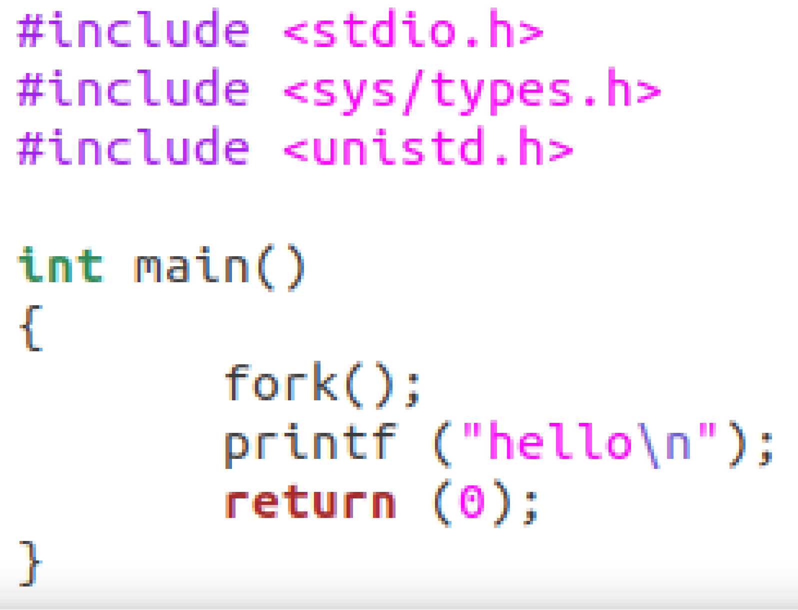
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include<sys/wait.h>
int main()
{
        pid_t pid;
        pid = fork(); /* fork a child process */
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) { /* child process */
                printf("Child Process\n");
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                wait(NULL); /* parent waits for child to complete */
                printf("Child Complete");
        }
        return 0;
}
```



OUTPUT:

```
Child Process
a.out
Documents  examples.desktop  MyPrograms  Pictures  Templates
Desktop  Downloads  Music     parent.c  Public    Videos
Child Complete
```

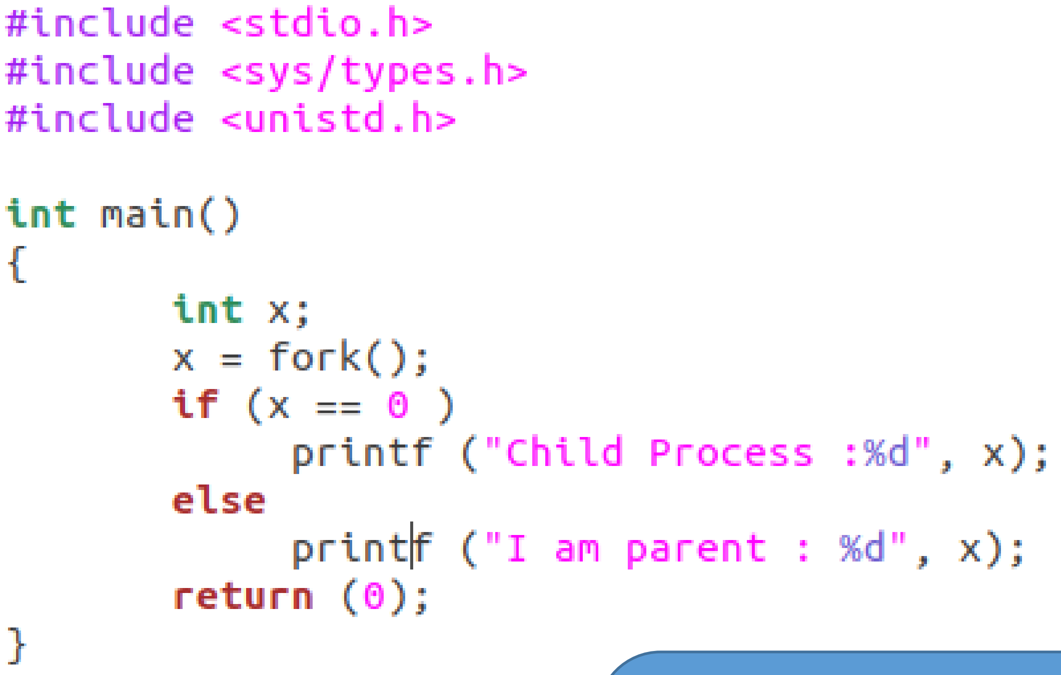
Example 1



```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        fork();
        printf ("hello\n");
        return (0);
}
```

```
hello
hello
```

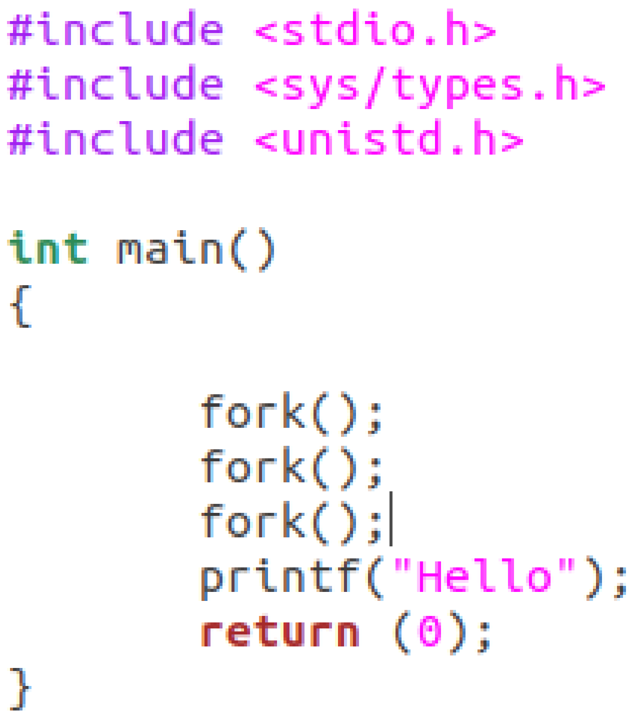
Example 2



```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        int x;
        x = fork();
        if (x == 0 )
             printf ("Child Process :%d", x);
        else
             printf ("I am parent : %d", x);
        return (0);
}
```

```
Child Process : 0
I am Parent : 1234
```

Example 3



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{

        fork();
        fork();
        fork();
        printf("Hello");
        return (0);
}
```

Process Creation

```
int main()
{
        pid_t pid;
        pid = fork();
        if (pid < 0) {
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) {
                printf("Child Process\n");
                printf("child pid = %d\n", getpid());
        }
        else {
                printf("parent pid = %d\n", getpid());
                wait(NULL);
                printf("Child Complete");
        }
        return 0;
}
```

OUTPUT:

```
parent pid = 6597
Child Process
child pid = 6598
Child Complete
```

Process Creation

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include<sys/wait.h>
int main()
{
        pid_t pid;   int x = 10;
        pid = fork();
                if (pid < 0) {
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) {
                printf("Child Process: x = %d\n", x);
                execlp("/bin/ls","ls",NULL);
        }
        else {
                printf("Parent Process: x = %d\n", x);
                wait(NULL);
                printf("Child Complete");
        }
        return 0;
}
```

OUTPUT:

```
Parent Process: x = 10
Child Process: x = 10
a.out Documents  examples.desktop  MyPrograms  Pictures  Templates
Desktop  Downloads  Music     parent.c  Public    Videos
Child Complete
```

Process Creation

```
int main()
{
        pid_t pid;
        int x = 10;
        pid = fork();
        if (pid < 0) {
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) {
                x = x + 10;
                printf("Child Process: x = %d\n", x);
        }
        else {
                wait(NULL);
                printf("Parent Process: x = %d\n", x);
                printf("Child Complete");
        }
        return 0;
}
```

OUTPUT:

```
Child Process: x = 20
Parent Process: x = 10
Child Complete
```

Process Creation

```
int main()
{
        pid_t pid;
        int x = 10;
        pid = fork();
        if (pid < 0) {
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) {
                for(long i = 0; i < 50000000000; i++);
                printf("Child Process: x = %d\n", x);
        }
        else {
                x = x + 10;
                wait(NULL);
                printf("Parent Process: x = %d\n", x);
                printf("Child Complete");
        }
        return 0;
}
```

OUTPUT:

```
Child Process: x = 10
Parent Process: x = 20
Child Complete
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the exit() system call.
- May return status data from child to parent (via wait())
- Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes because:
  - Child has exceeded allocated resources limit
  - Task assigned to child is no longer required
  - Parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exist if parent has terminated.
- If a process terminates, then all its children must also be terminated.
  - **cascading termination -** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()** system call**.** The call returns status information and the pid of the terminated process

```
pid_t pid;
int status;
pid = wait(&status); //parent can tell which child has terminated
```

- If no parent waiting (did not invoke **wait()** till then) process is a **zombie**
- If parent terminated, process is an **orphan**

Zombie Process

```
int main()
{
   pid_t child_pid = fork();
   if (child_pid > 0) {
      sleep(10);  ————→  zombie
      wait(NULL);  ————→  no longer a zombie
      sleep(200);
   }
   else{
      printf("\n%d",getpid());
      exit(0);
    }
    return 0;
}
```

```
F S   UID    PID    PPID  C  PRI   NI ADDR   SZ  WCHAN  TTY       TIME       CMD

1 Z  1001 17404 17403  0  80    0     -      0      -      pts/0   00:00:00   a.out <defunct>
```

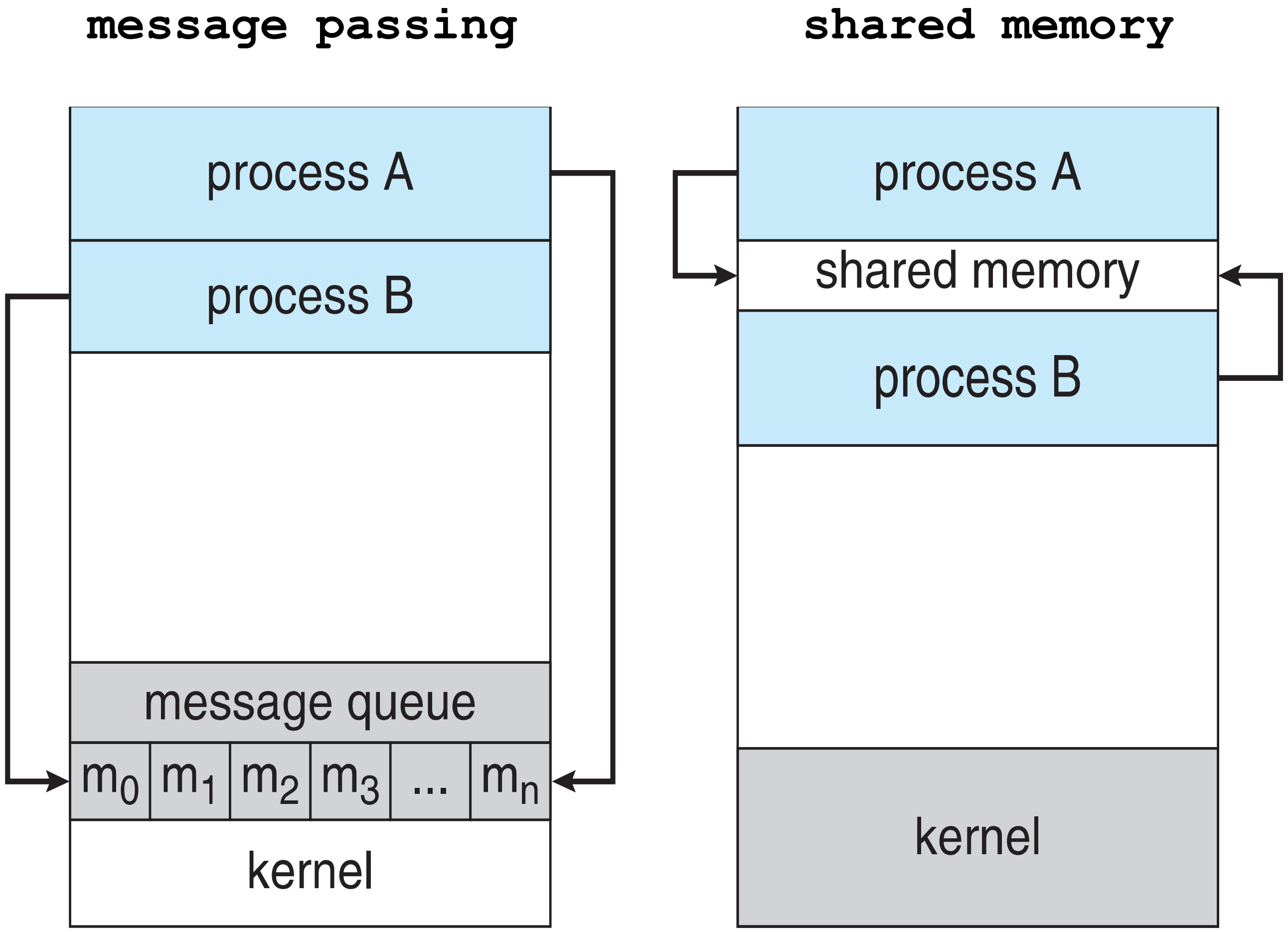
Orphan Process

```
int main()
{
    pid_t child_pid = fork();
    if (child_pid > 0){
        printf("\nParent process: %d\n",getpid());
        sleep(6);
     }
    else{
        printf("\nParent PID: %d\n",getppid());
        sleep(20);
        printf("\nChild Process: %d",getpid());
        printf("\nParent PID: %d",getppid());
        exit(0);
     }

    return 0;
}
```

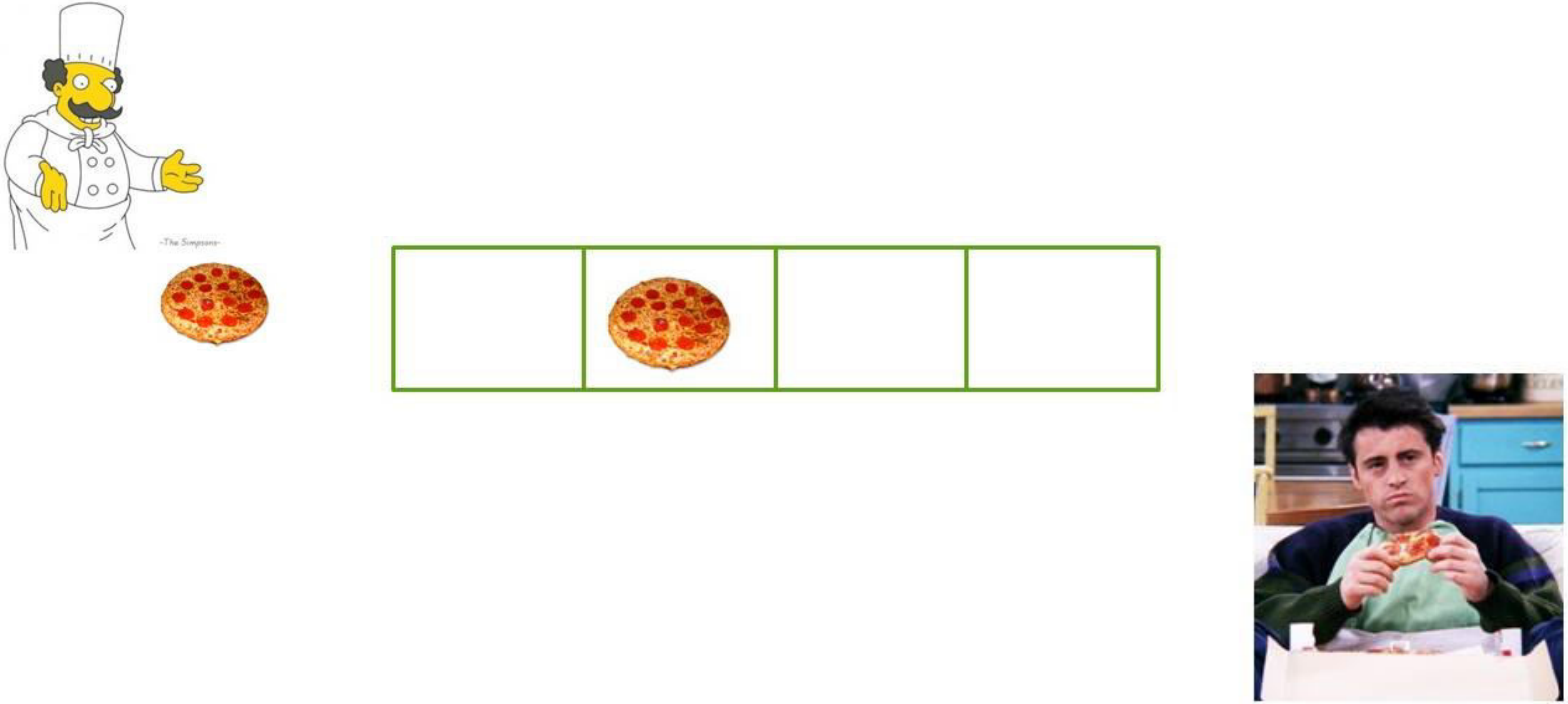
Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Interprocess Communication



Producer-Consumer Problem



IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate

- The communication is under the control of the users processes not the operating system

- Provide mechanism that will allow the user processes to synchronize their actions when they access shared memory

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

- Shared data, reside in a region of memory shared by producer & consumer

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded Buffer - Producer

```
item next_produced;
while (true) {
       /* produce an item in next produced */
       while (((in + 1) % BUFFER_SIZE) == out) //buffer is full
              ; /* do nothing */
       buffer[in] = next_produced;
       in = (in + 1) % BUFFER_SIZE;
}
```

Producer

Consumer

```
item next_consumed;
while(true){
       while (in == out) //buffer is empty
              ; /*do nothing*/
       next_consumed = buffer[out];
       out = (out + 1) % BUFFER_SIZE;
       /* consume the item in next consumed */
}
```

IPC – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- processes communicate with each other without resorting to shared variables, no sharing of address space
- IPC facility provides two operations:
  - **`send(`*`message`*`)`**
  - **`receive(`*`message`*`)`**
- The *message* size is either fixed or variable
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via `send()` / `receive()`

Message Passing – Direct Communication

- Processes must name each other explicitly:
  - **`send (P, message)`** – send a message to process *P*
  - **`receive(Q, message)`** – receive a message from process *Q*

- Properties of communication link
  - Links are established automatically
  - Processes only need to know each other's identity
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link

Message Passing – Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique ID
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link is established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links, each link corresponds to one mailbox

Message Passing – Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
- **send(*A, message*)** – send a message to mailbox A
- **receive(*A, message*)** – receive a message from mailbox A

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received by the receiving process or mailbox
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continues
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message

Buffering

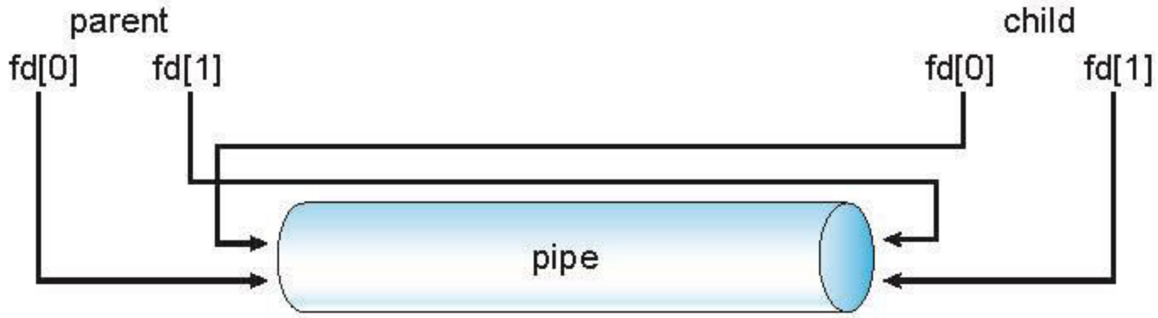
- messages exchanged by communicating processes reside in a temporary queue
- implemented in one of three ways
  - **Zero capacity** – no messages are queued, link can't have any waiting messages, sender must block until receiver receives message
  - **Bounded capacity** – queue is of finite length of *n* messages, sender need not block if queue is not full, sender must wait if queue full
  - **Unbounded capacity** – infinite length queue, sender never blocks

Pipe

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** –
  - cannot be accessed from outside the process that created it
  - parent process creates a pipe and uses it to communicate with a child process that it created
- **Named pipes** – can be accessed without a parent-child relationship

Ordinary Pipe

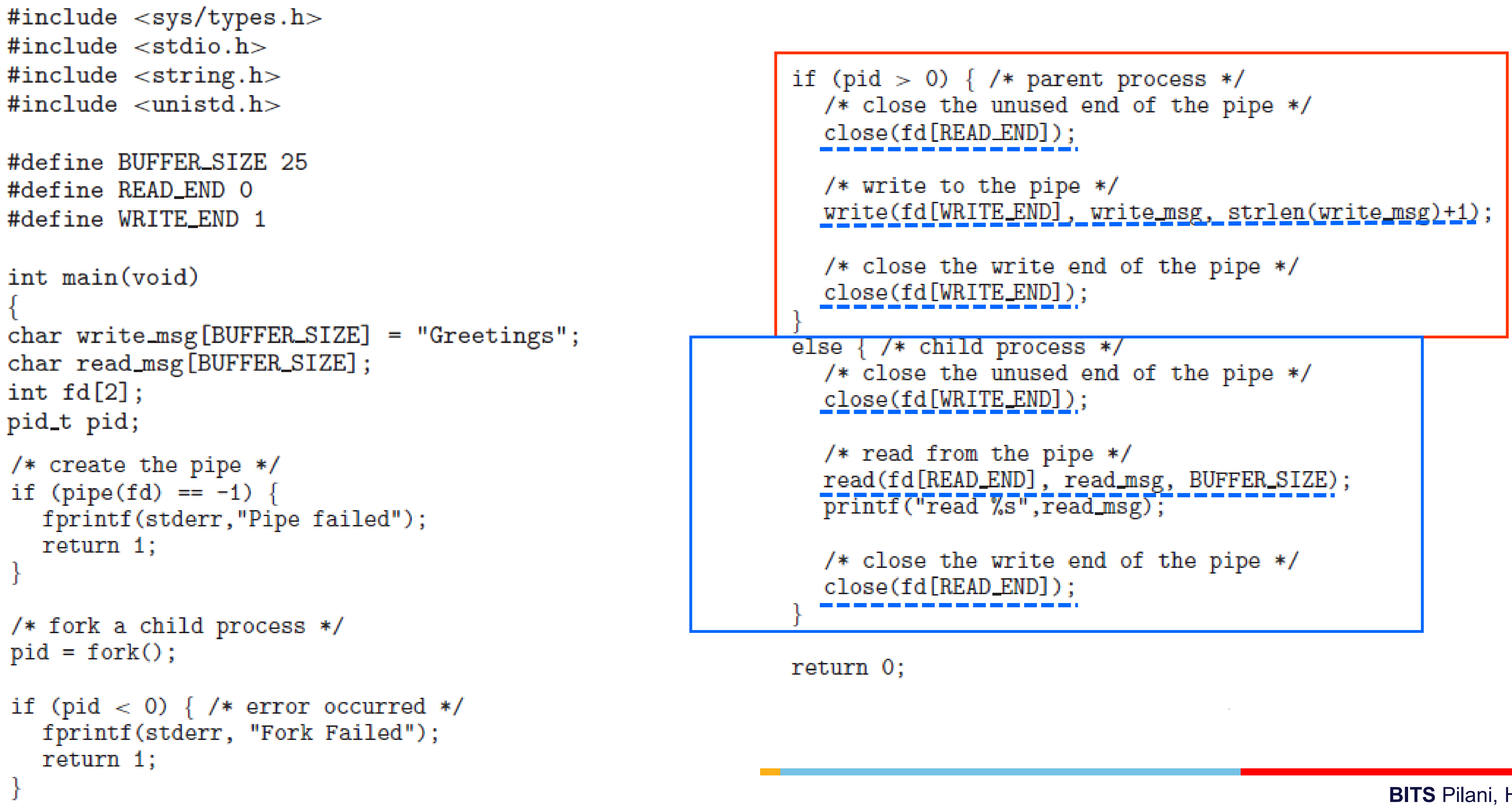
- ❖Ordinary Pipes allow communication in standard producer-consumer style
- ❖Producer writes to one end (the **write-end** of the pipe)
- ❖Consumer reads from the other end (the **read-end** of the pipe)
- ❖Ordinary pipes are unidirectional
- ❖Require parent-child relationship between communicating processes
- ❖Windows calls these **anonymous pipes**



Ordinary Pipe

- ❖ordinary pipe can’t be accessed from outside the process that created it
- ❖parent process creates a pipe and uses it to communicate with a child process that it creates via fork()
- ❖child inherits the pipe from its parent process like any other file

Ordinary Pipe



```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

 /* create the pipe */
 if (pipe(fd) == -1) {
   fprintf(stderr,"Pipe failed");
   return 1;
 }

 /* fork a child process */
 pid = fork();

 if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
 }
```

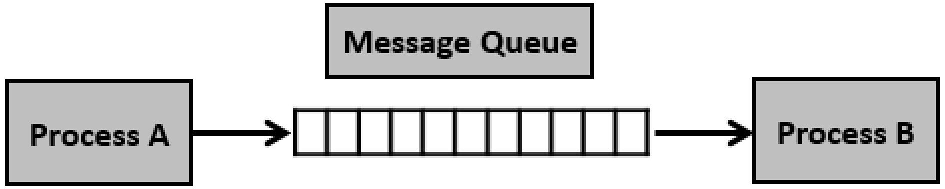
```
if (pid > 0) { /* parent process */
   /* close the unused end of the pipe */
   close(fd[READ_END]);

   /* write to the pipe */
   write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

   /* close the write end of the pipe */
   close(fd[WRITE_END]);
}
else { /* child process */
   /* close the unused end of the pipe */
   close(fd[WRITE_END]);

   /* read from the pipe */
   read(fd[READ_END], read_msg, BUFFER_SIZE);
   printf("read %s",read_msg);

   /* close the write end of the pipe */
   close(fd[READ_END]);
}

return 0;
```

Named Pipe

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Do not cease to exist if the communicating processes have terminated
- Provided on both UNIX and Windows systems
- Referred to as FIFOs in UNIX systems

Message Queue

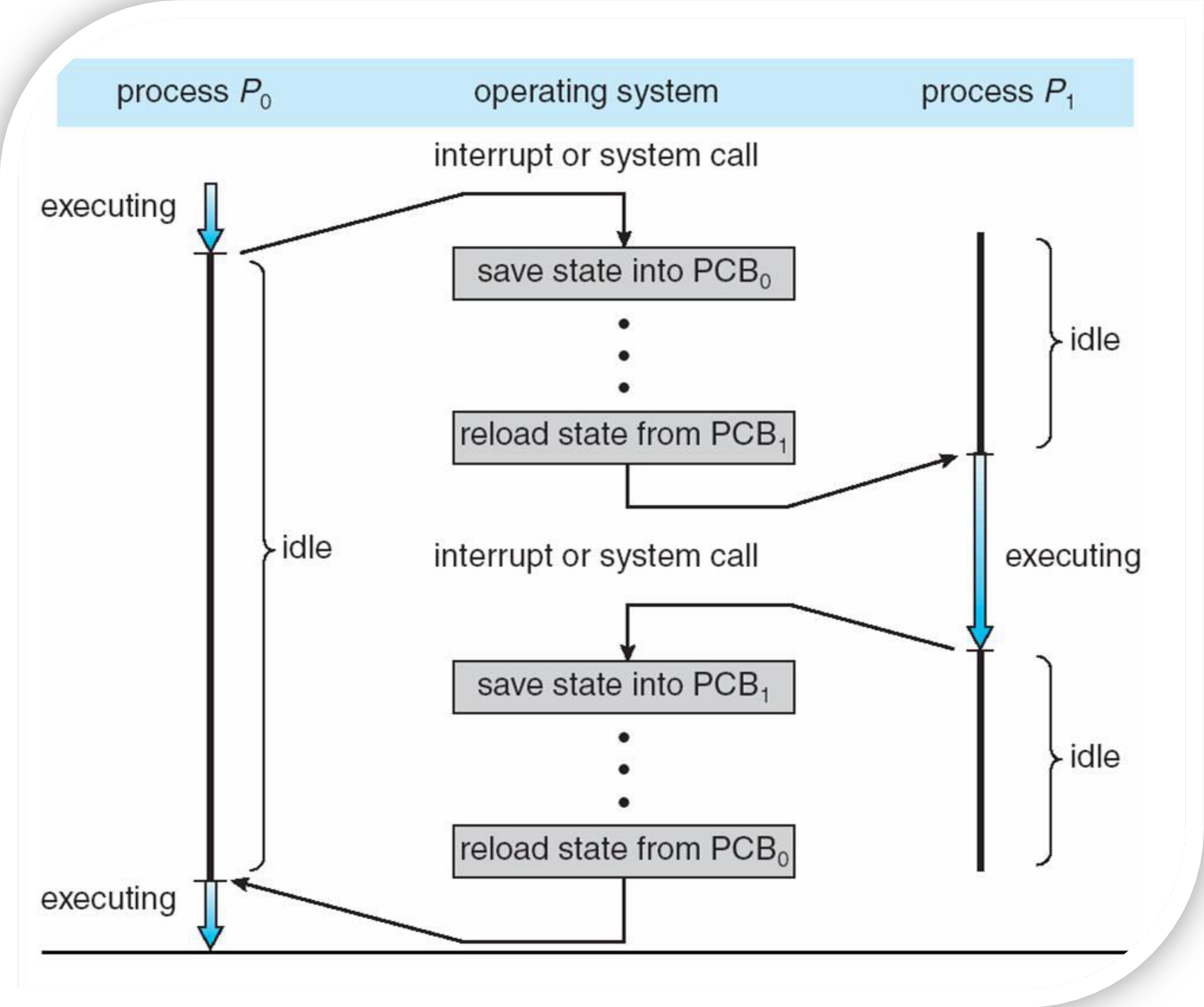
- asynchronous communication
- messages placed onto the queue are stored until the recipient retrieves them



- **Step 1** – Create a message queue or connect to an already existing message queue (msgget())
- **Step 2** – Write into message queue (msgsnd())
- **Step 3** – Read from the message queue (msgrcv())
- **Step 4** – Perform control operations on the message queue (msgctl())

Context Switch

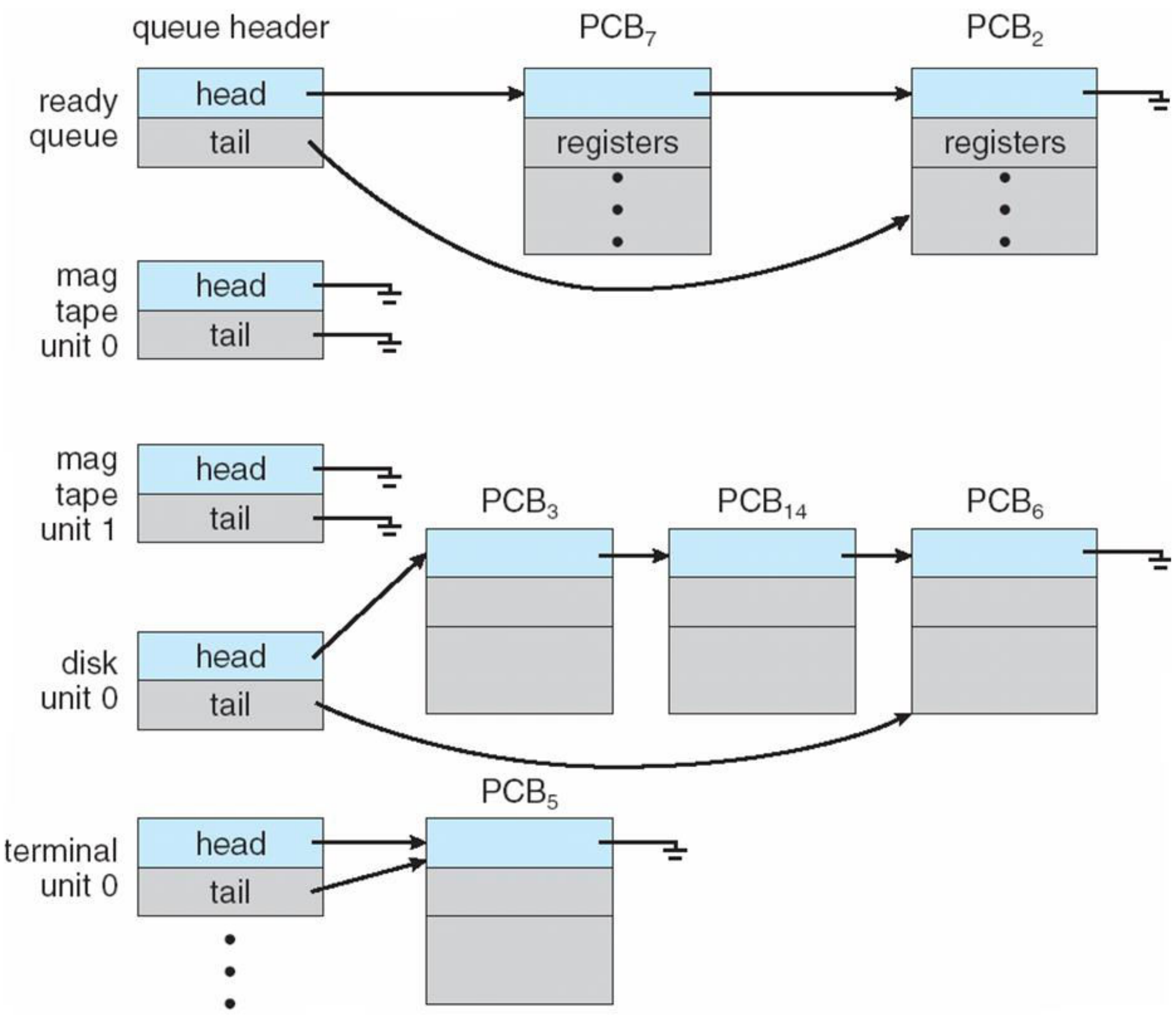
- When CPU switches to another process, system must **save state** of the old process and load the **state** for the new process via a **context switch**
- **Context** of a process represented in the PCB (CPU registers contents, process state, memory management info.)
- Context-switch time is overhead; system does no useful work while switching
- Time is dependent on hardware support



Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute, generally stored as a linked list
  - **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Various Queues

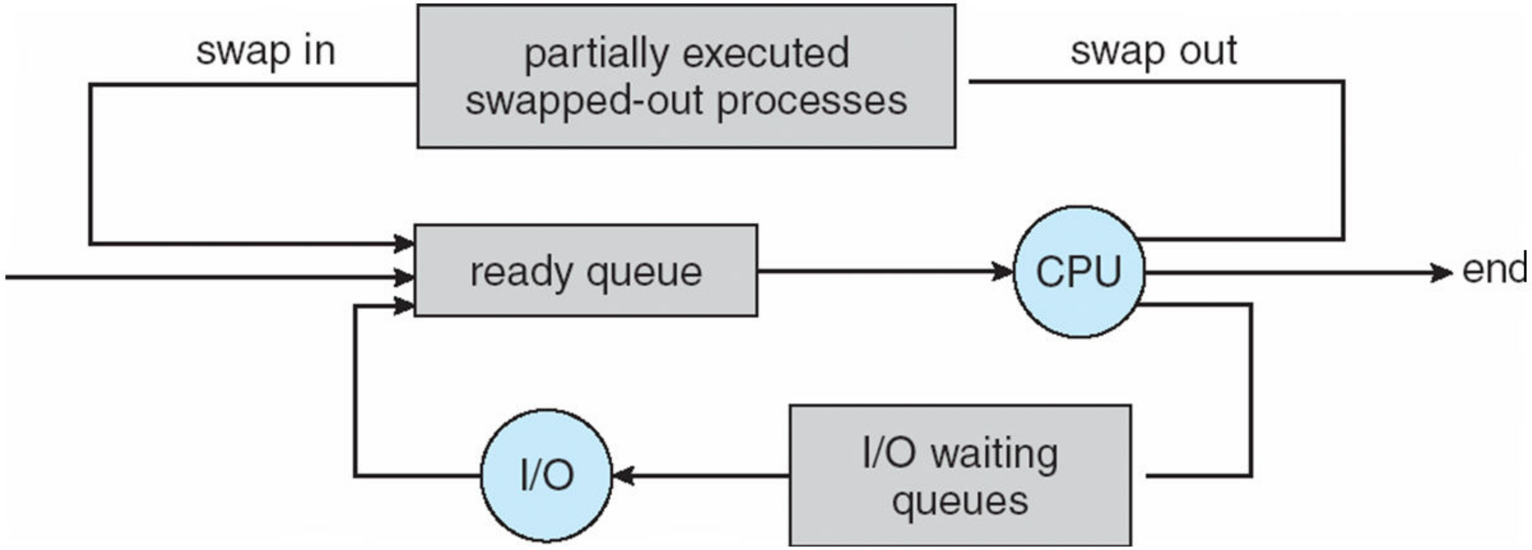


Schedulers

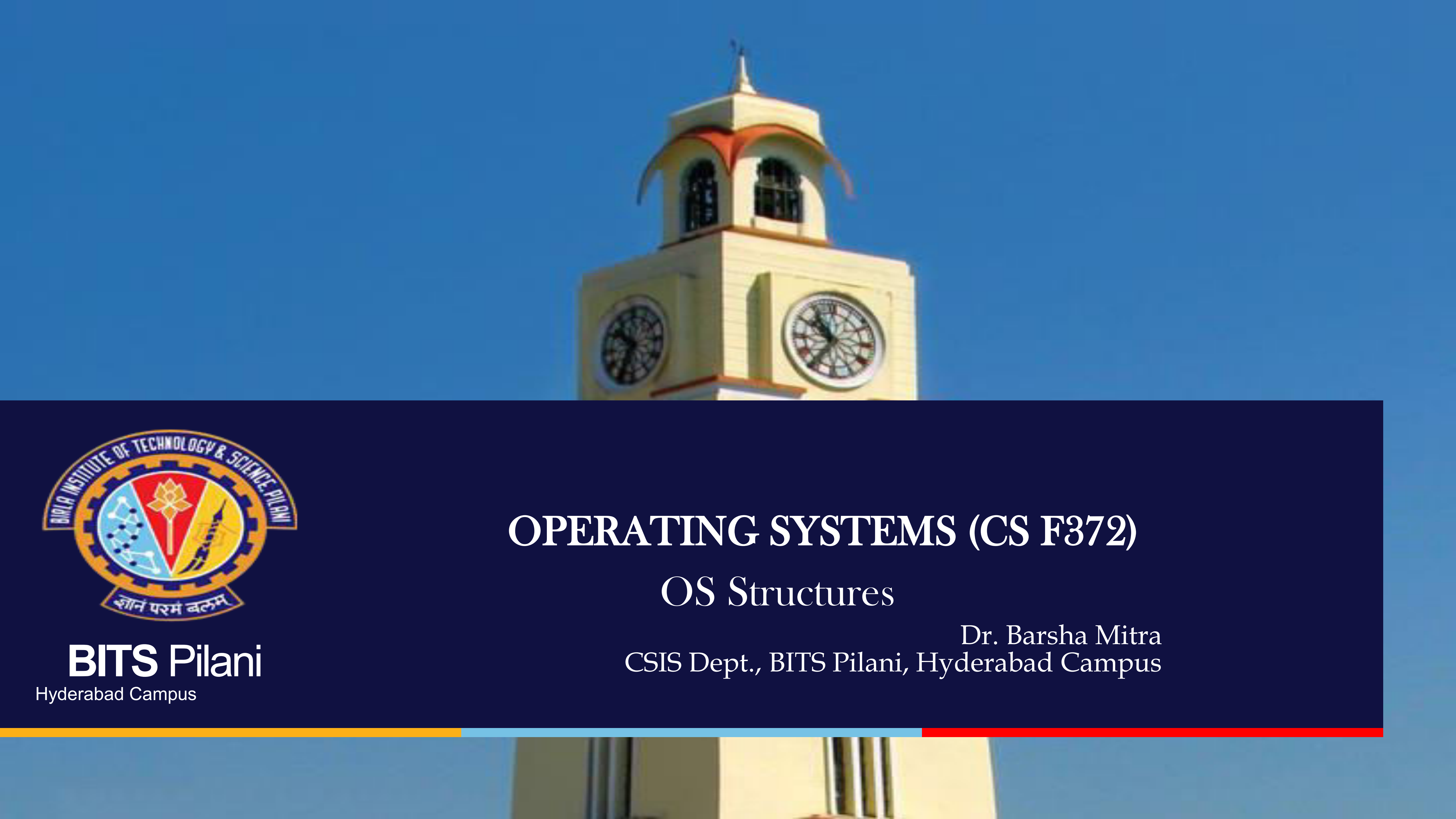
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes from job queue should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming (number of processes in main memory)**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Schedulers

- **Medium-term scheduler** can be added in time sharing systems if degree of multiprogramming needs to decrease
  - Intermediate level of scheduling
  - Remove process from memory, store on disk, bring back in from disk to continue execution from where it left off: **swapping**
  - **Required for improving process mix or for freeing of memory**



Thank You



BITS Pilani
Hyderabad Campus

OPERATING SYSTEMS (CS F372)

OS Structures

Dr. Barsha Mitra
CSIS Dept., BITS Pilani, Hyderabad Campus

Operating System Services

- **User interface** - almost all operating systems have a user interface (**UI**)
  - **Command-Line (CLI), Graphics User Interface (GUI)**
- **Program execution** - system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- **I/O operations** - running program may require I/O, which may involve a file or an I/O device
- **File-system manipulation** - read and write files and directories, create and delete them, search them, list file Information, permission management

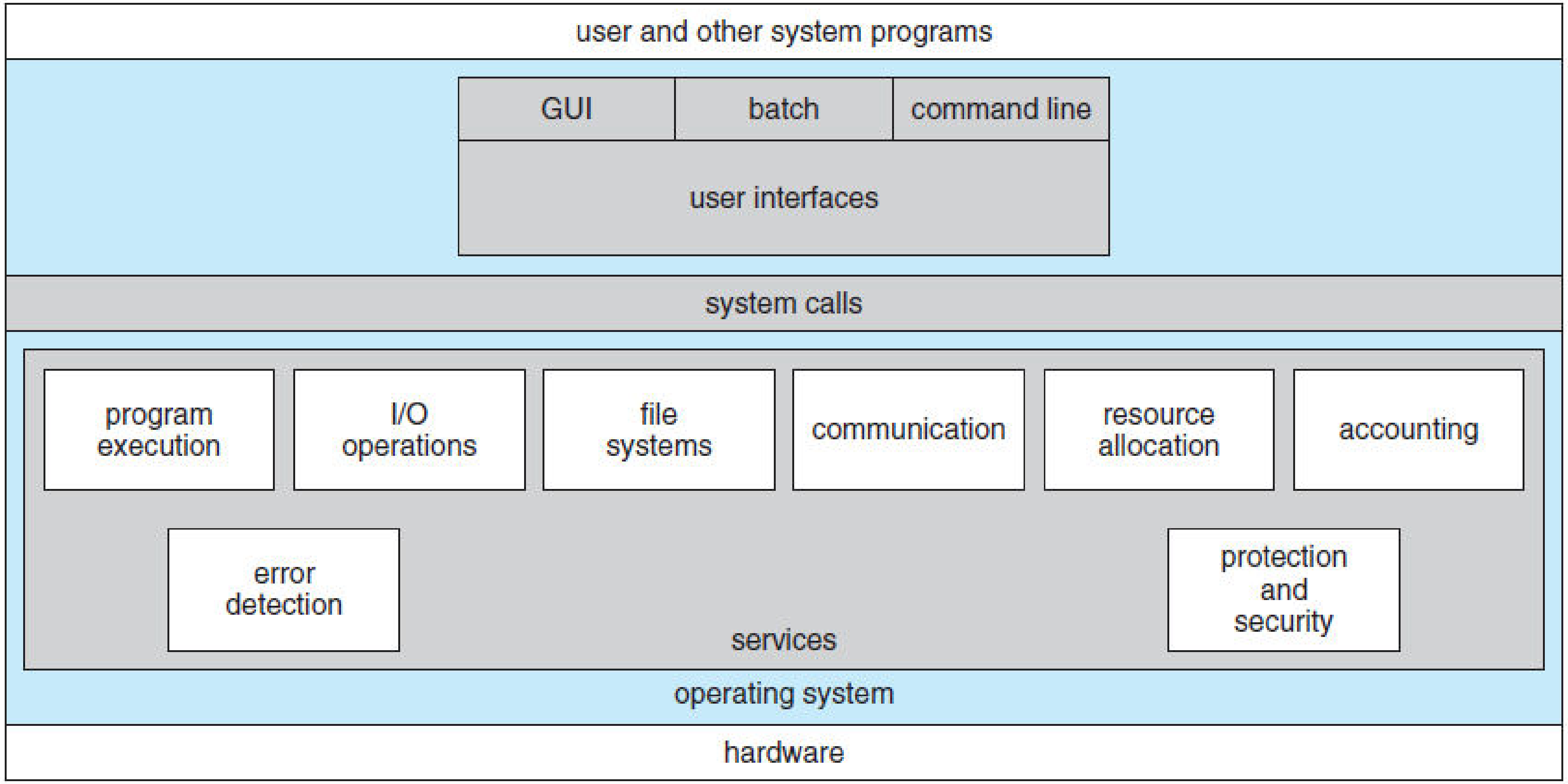
Operating System Services

- **Communications** – processes may exchange information, on the same computer or between computers over a network, **shared memory or message passing**
- **Error detection** –
  - OS needs to be constantly aware of possible errors
  - May occur in CPU and memory h/w, in I/O devices, in user program
  - OS should take the appropriate action to ensure correct and consistent computing
  - Take corrective actions

Operating System Services

- **Resource allocation** – allocating resources like CPU cycles, main memory, file storage, I/O devices for multiple concurrently executing processes
- **Accounting** – keep track of which users use how much and what kinds of computer resources
- **Protection and security** –
  - owners of information stored in a multiuser or networked computer system want to control use of that information
  - concurrent processes should not interfere with each other or with OS
  - ensuring that all accesses to system resources is controlled
  - security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

Operating System Services



User and Operating-System Interface: CLI

- **CLI** or command interpreter
- Sometimes implemented in kernel, sometimes by separate program (Unix, Windows)
- Sometimes multiple flavors implemented – shells
- Primarily fetches a command from user and executes it

User and Operating-System Interface: GUI

- User-friendly interface
- Usually mouse, keyboard, and monitor
- Icons represent files, programs, actions, etc.
- Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder))
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

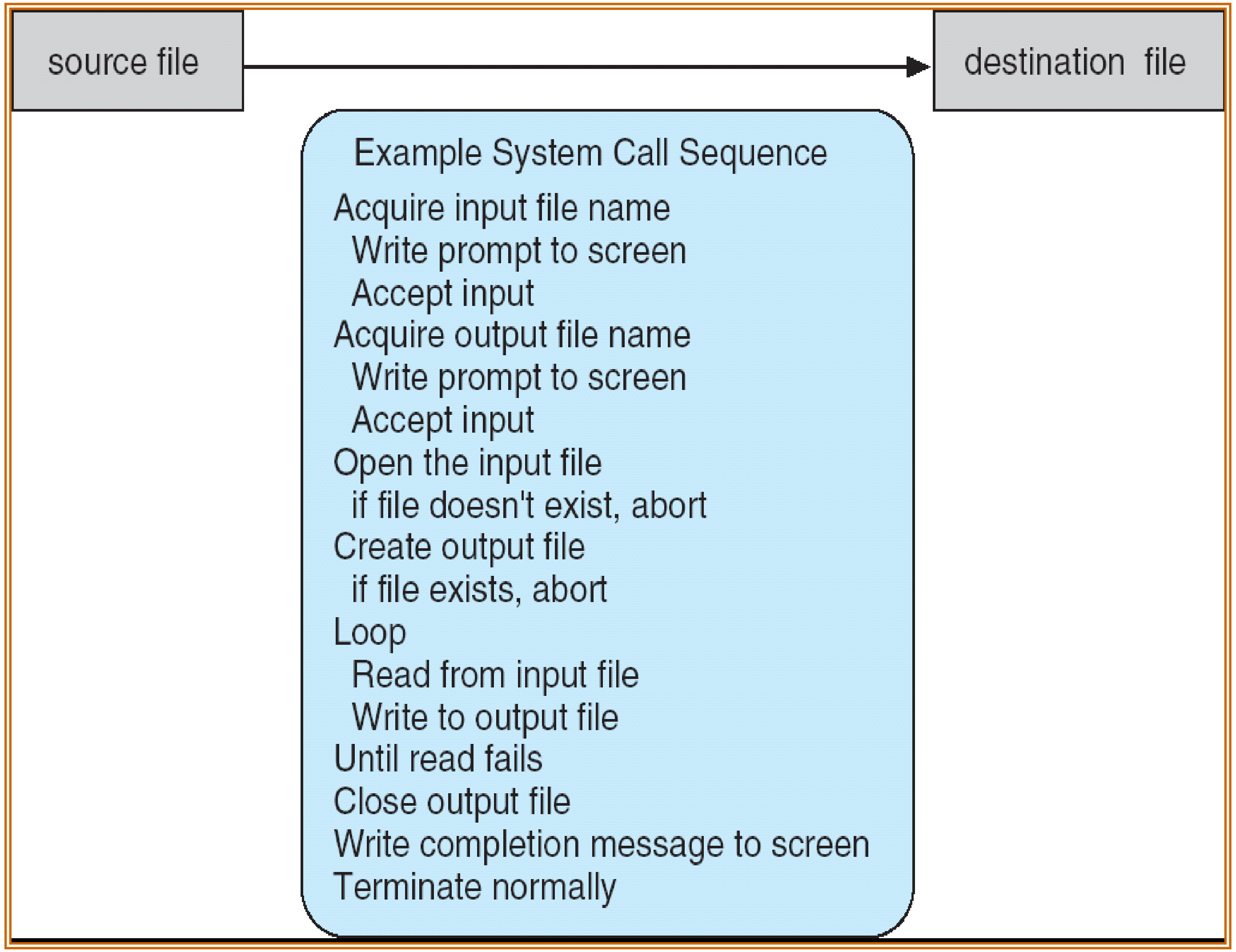
User and Operating-System Interface: Touchscreen Interface

- ❖ Touchscreen devices require new interfaces
- ❖ Mouse not possible or not desired
- ❖ Actions and selection based on gestures
- ❖ Virtual keyboard for text entry
- ❖ Voice commands

User and Operating-System Interface:

Choice of Interface

System Calls



source file

destination file

Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

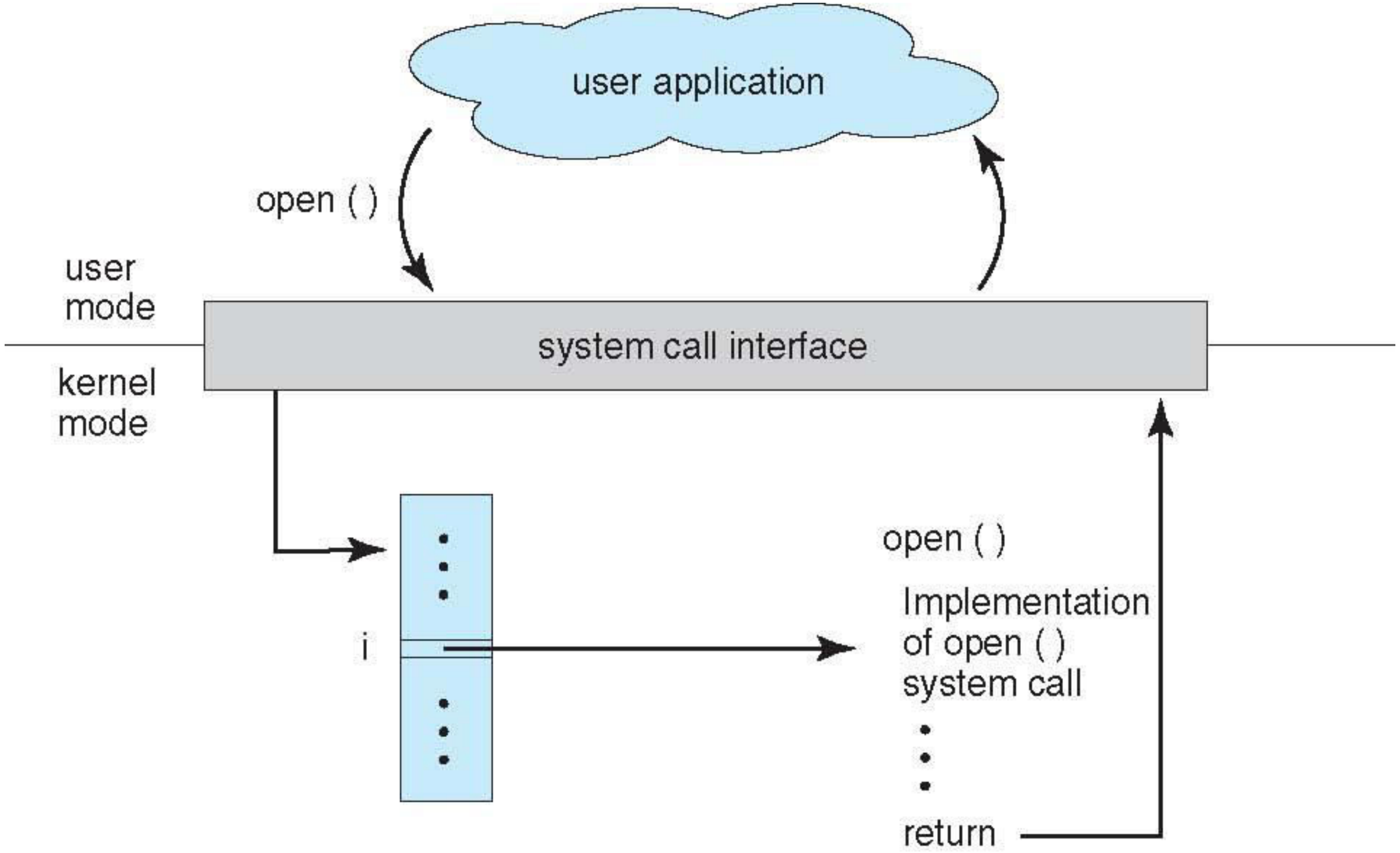
System Calls

- Interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
- API specifies a set of functions available to application programmers
- Programmers access API via code library provided by the OS
- Three most common APIs are
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

System Calls

- A number is associated with each system call
- System-call interface maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
- Just needs to obey API and understand what OS will do as a result of call execution
- Most details of OS interface hidden from programmer by API
- Managed by run-time support library (set of functions built into libraries included with compiler)

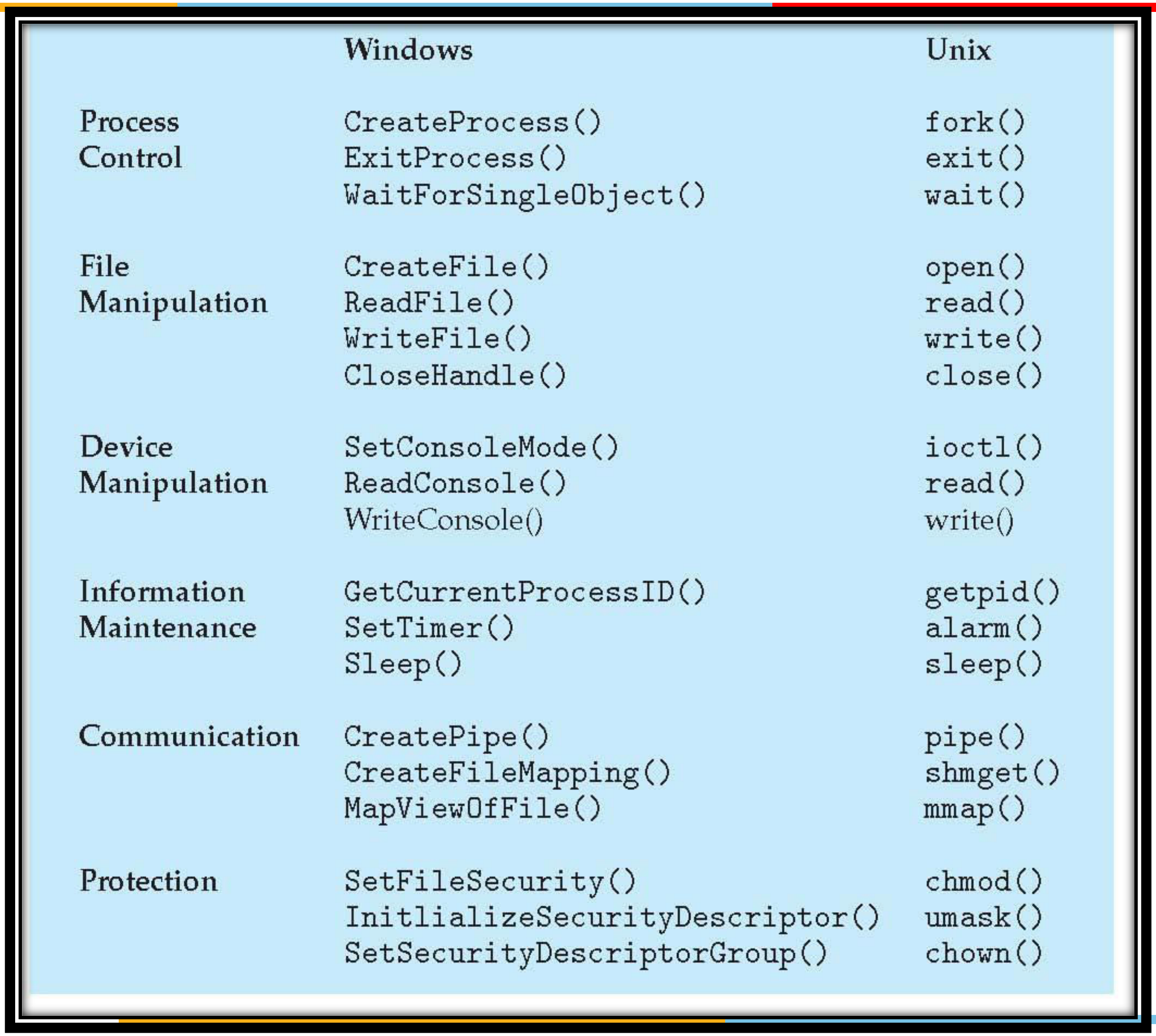
System Calls



Types of System Calls

- **Process control**
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
- **File management**
  - create file, delete file
  - open, close file
  - read, write file
  - get and set file attributes
- **Device management**
  - request device, release device
  - read, write
  - get device attributes, set device attributes
  - logically attach or detach devices
- **Information Maintenance**
- **Communication**
- **Protection**

Examples of System Calls



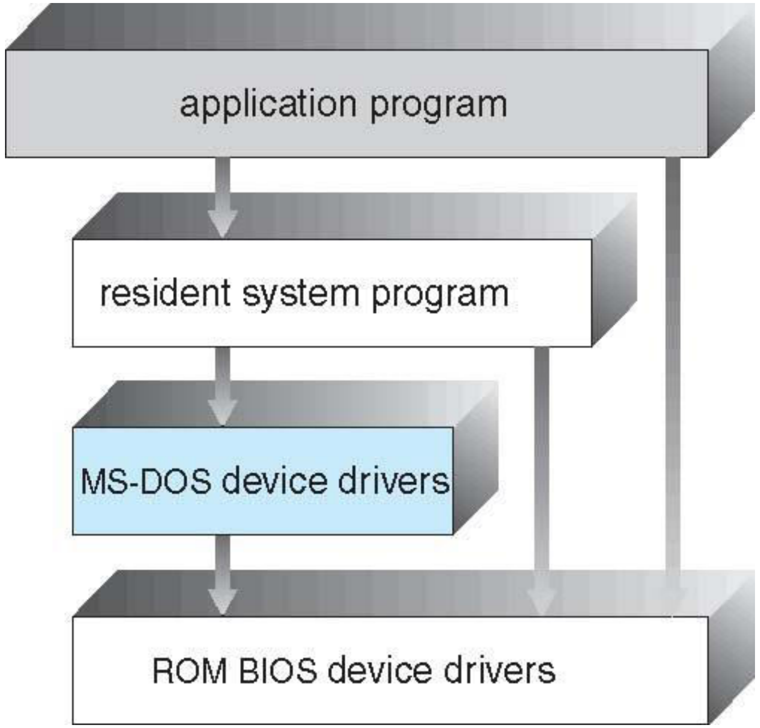
	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information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

OS Structure

- Simple Structure/ Monolithic Kernel
- Layered Approach
- Microkernels
- Modules
- Hybrid System

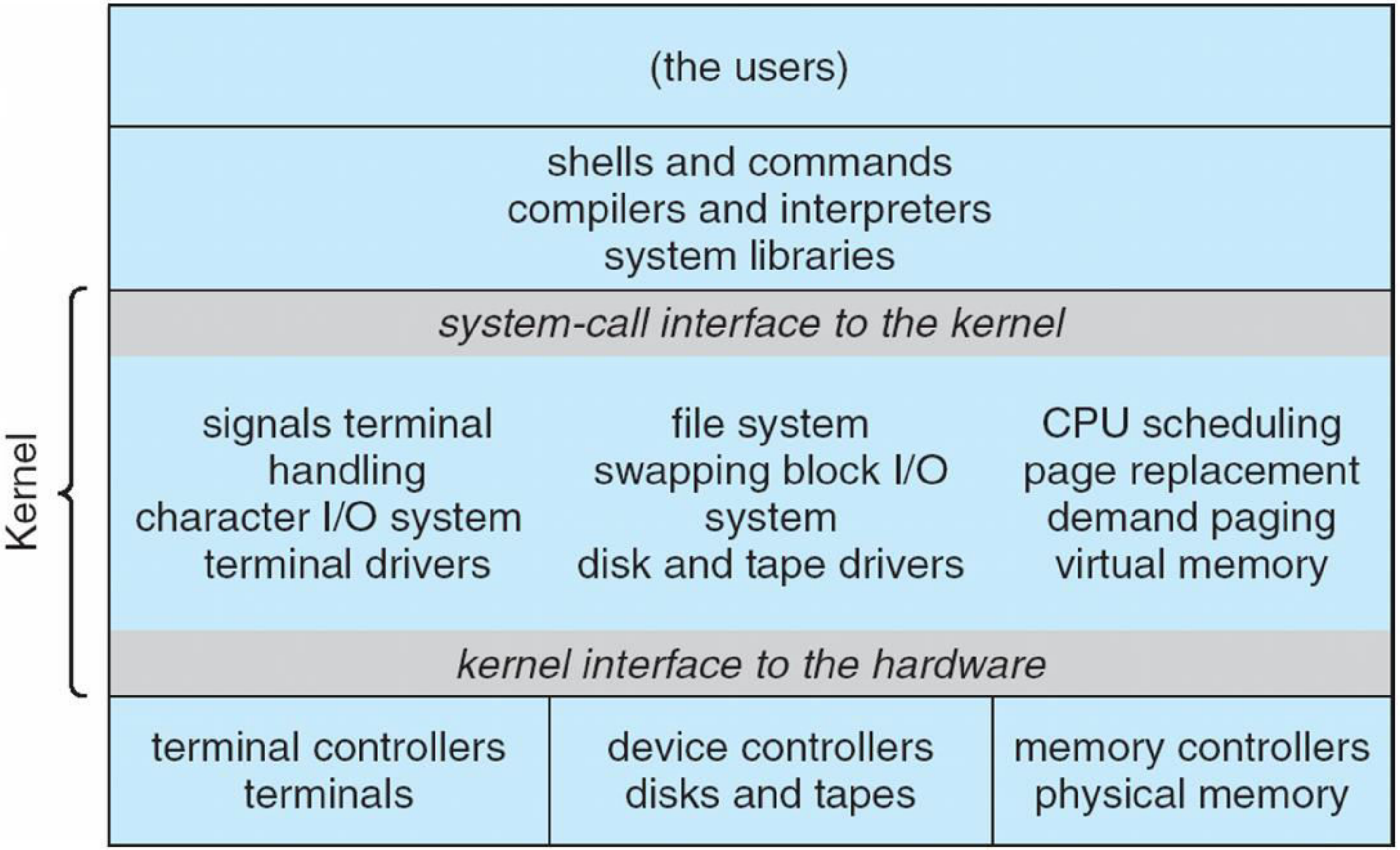
Simple Structure

- not divided into modules
- interfaces and levels of functionality are not well separated
- application programs are able to access the basic I/O routines to write directly to the display and disk drives
- vulnerable to malicious programs, causing entire system crashes when user programs fail



UNIX Architecture

- the original UNIX operating system had limited structuring
- consists of two separable parts
  - Systems programs
  - kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level



(the users)

shells and commands
compilers and interpreters
system libraries

system-call interface to the kernel

Kernel

signals terminal handling
character I/O system
terminal drivers

file system
swapping block I/O system
disk and tape drivers

CPU scheduling
page replacement
demand paging
virtual memory

kernel interface to the hardware

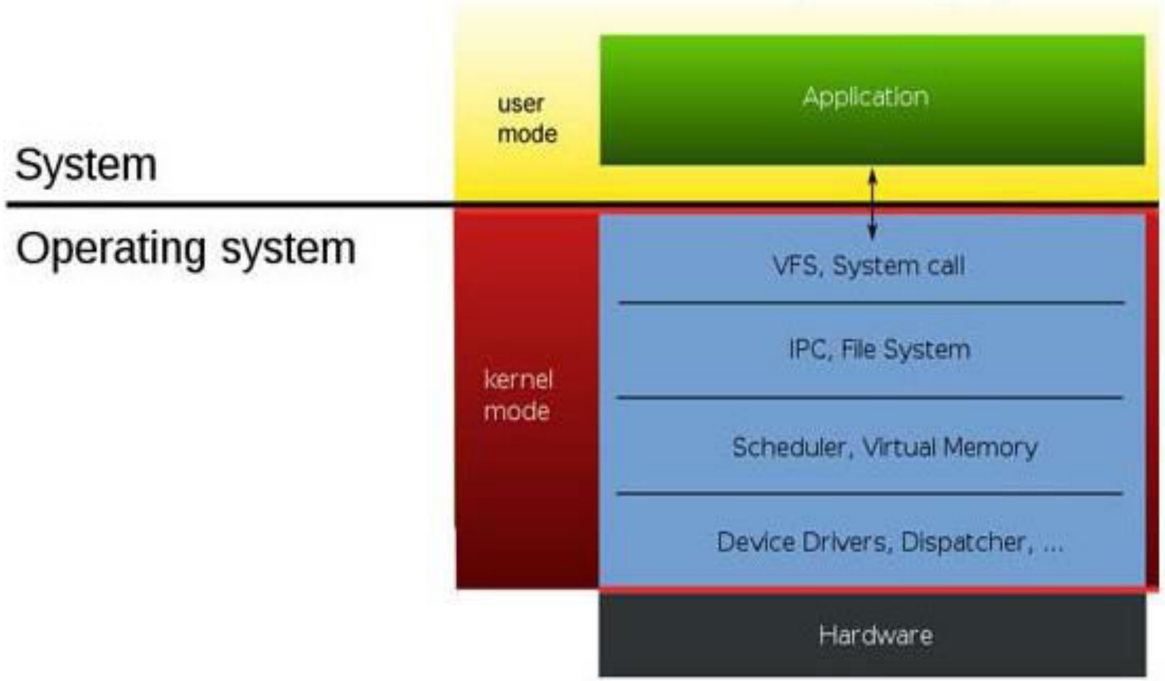
terminal controllers
terminals

device controllers
disks and tapes

memory controllers
physical memory

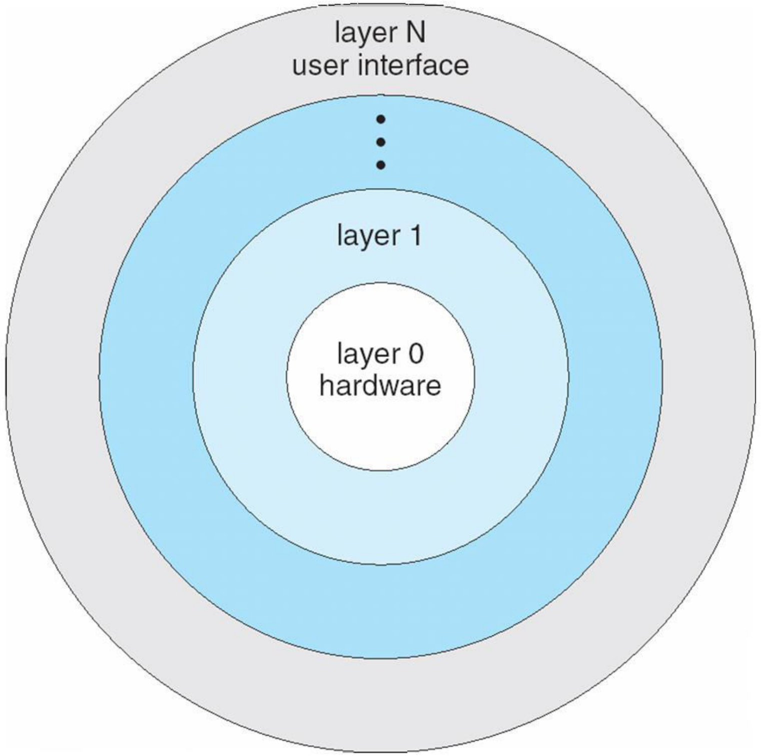
Monolithic Kernel

- entire operating system is working in kernel space
- larger in size
- little overhead in system call interface or in communication within kernel
- faster
- hard to extend
- if a service crashes, whole system is affected
- Eg., - Linux, Solaris, MS-DOS



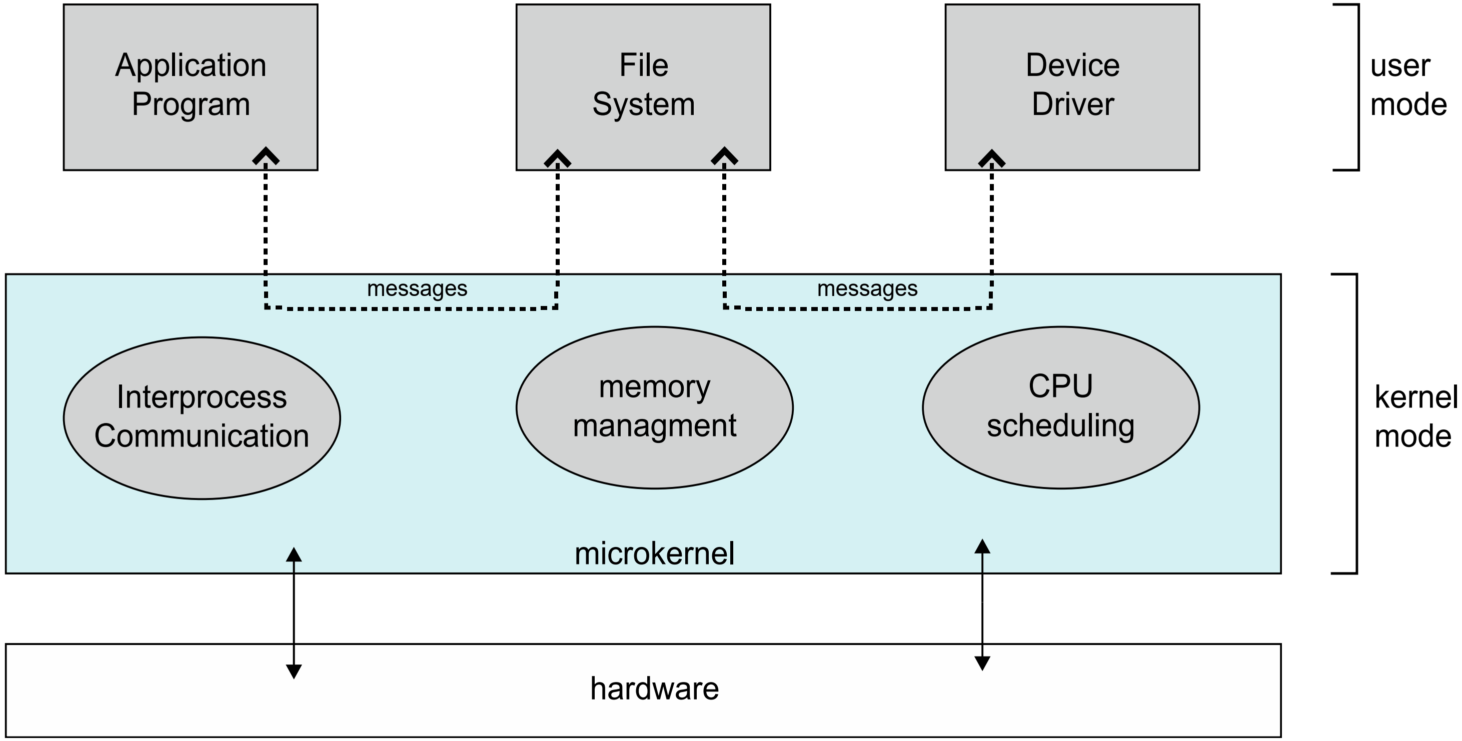
Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers
- The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



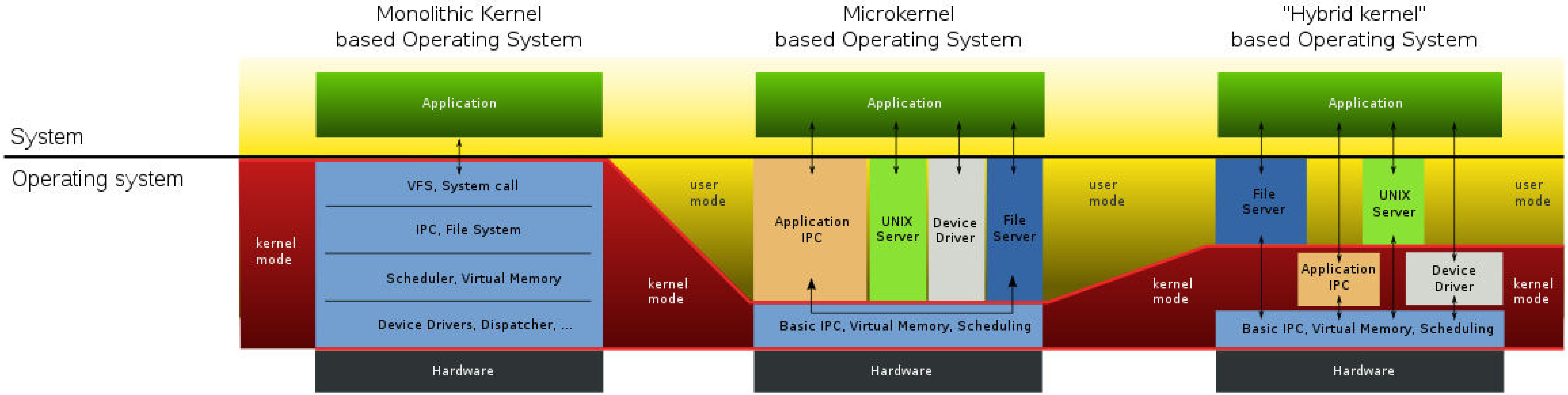
Microkernel

- user services and kernel services are in separate address spaces
- smaller in size
- slower
- extendible, all new services are added to user space
- if a service crashes, working of microkernel is not affected
- more secure and reliable
- eg., Mach, QNX, Windows NT (initial release)



- Drawback ??? *Performance overhead of user space to kernel space communication*

A Comparison

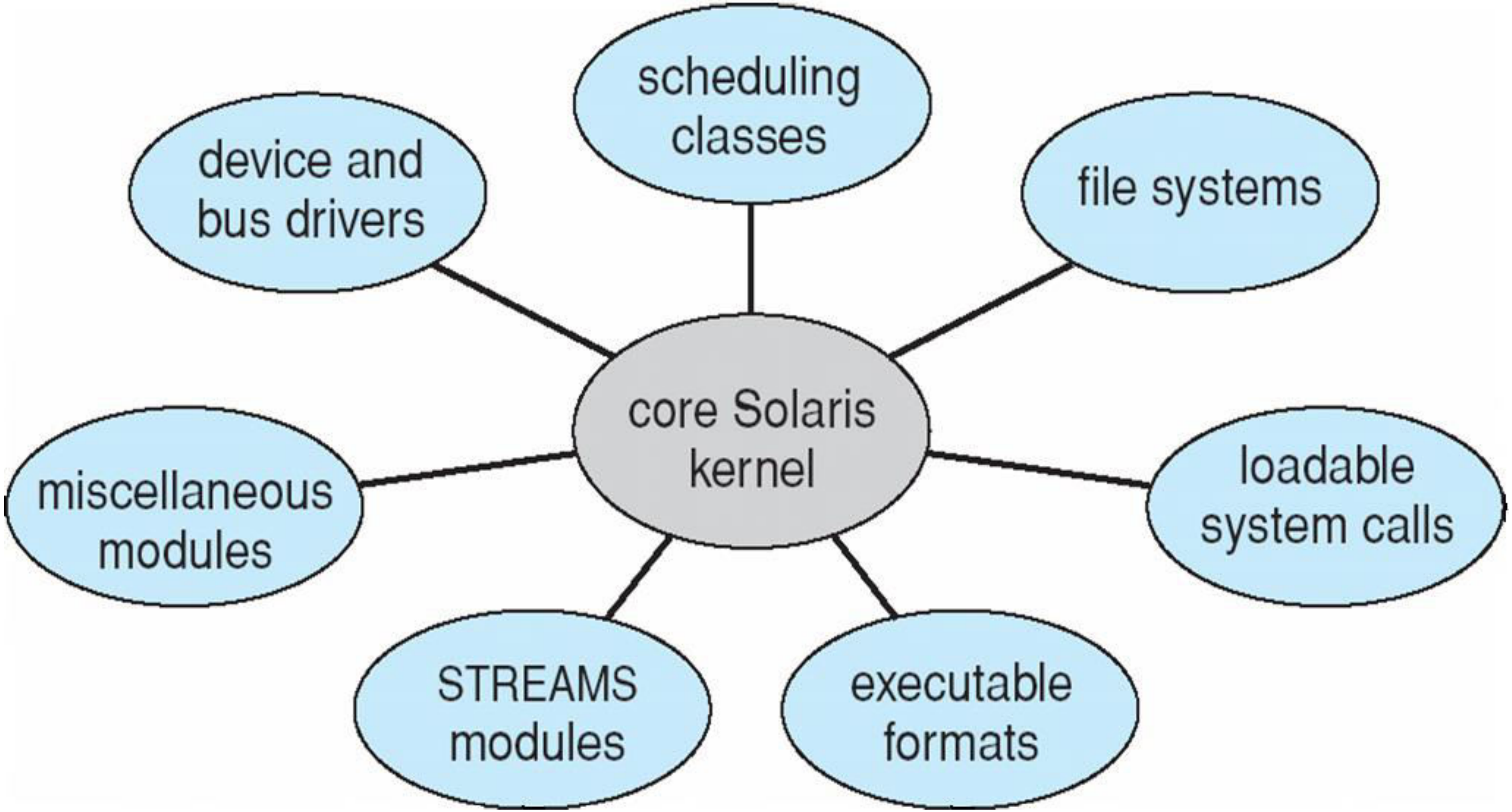


macOS X, Windows, iOS

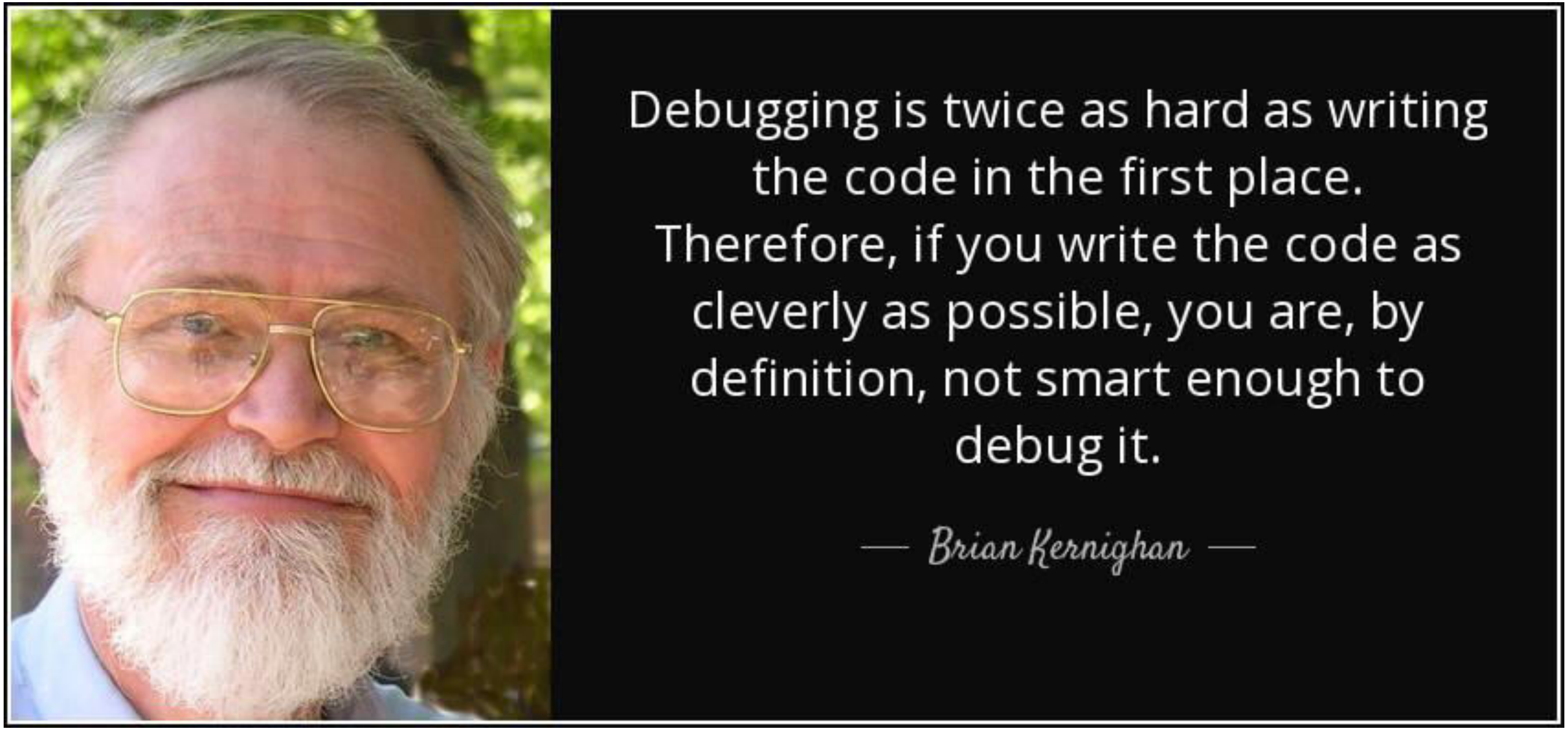
Modules

- loadable kernel modules
- kernel has a core set of components
- links in additional services via modules, either at boot time or during run time
- each module has a well defined interface
- dynamically linking services is preferable to adding new features directly to the kernel → does not require recompiling the kernel for every change
- better than a layered approach → any module can call any module
- better than microkernel → no message passing required to invoke modules

Modules


scheduling classes
device and bus drivers
file systems
core Solaris kernel
miscellaneous modules
loadable system calls
STREAMS modules
executable formats

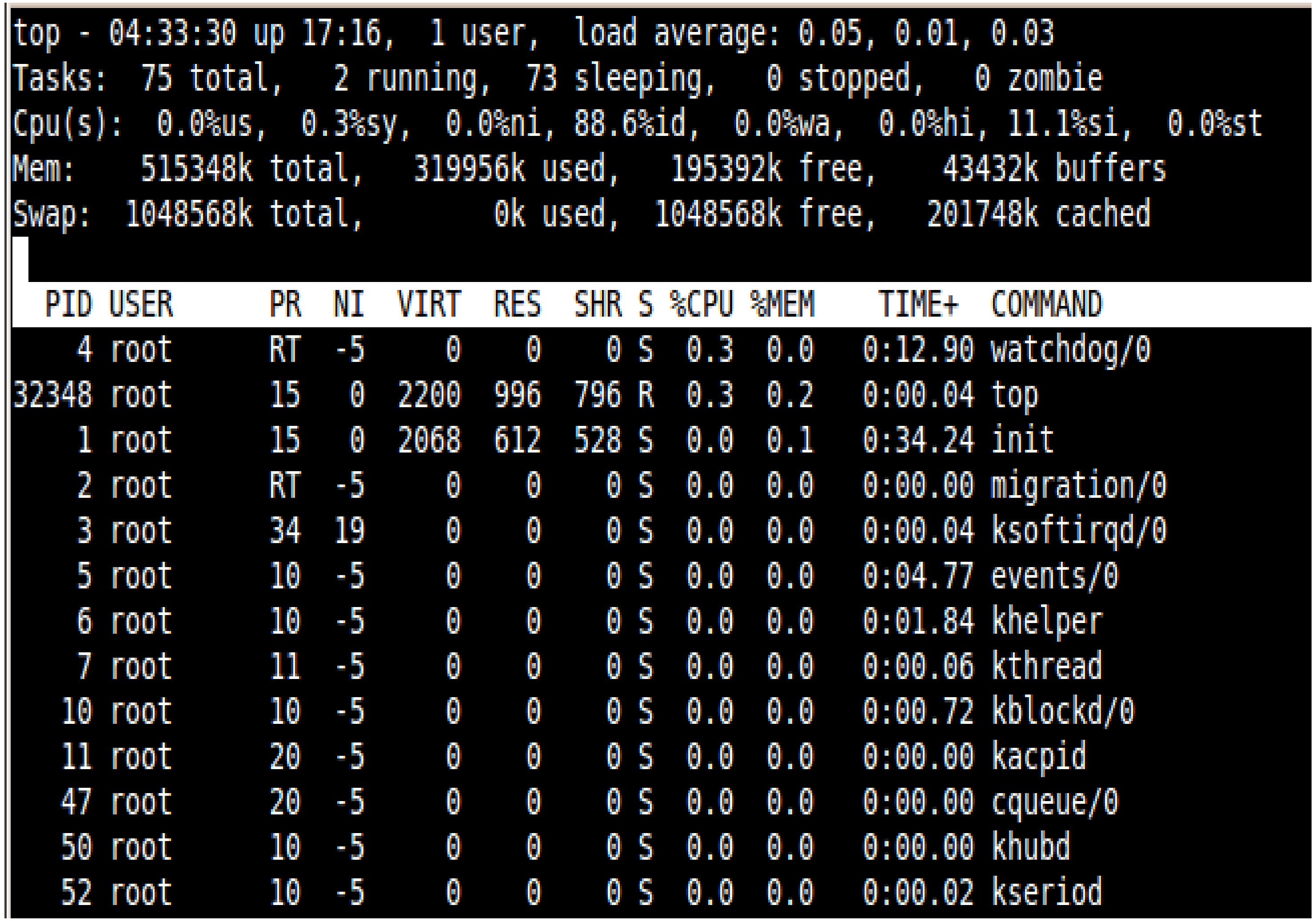
Operating-System Debugging



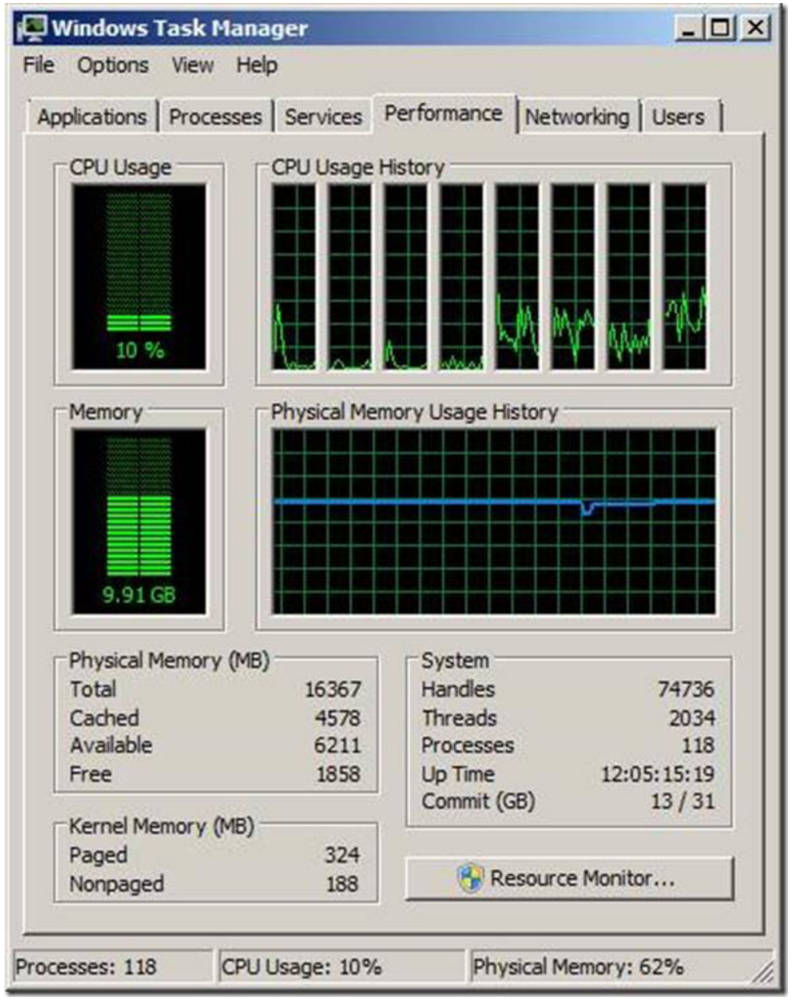
Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

— Brian Kernighan —

Performance Tuning



```
top - 04:33:30 up 17:16,  1 user,  load average: 0.05, 0.01, 0.03
Tasks:  75 total,   2 running,  73 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.3%sy,  0.0%ni, 88.6%id,  0.0%wa,  0.0%hi, 11.1%si,  0.0%st
Mem:    515348k total,   319956k used,   195392k free,    43432k buffers
Swap:  1048568k total,        0k used,  1048568k free,   201748k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    4 root      RT  -5     0    0    0 S  0.3  0.0   0:12.90 watchdog/0
32348 root      15   0  2200  996  796 R  0.3  0.2   0:00.04 top
    1 root      15   0  2068  612  528 S  0.0  0.1   0:34.24 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.04 ksoftirqd/0
    5 root      10  -5     0    0    0 S  0.0  0.0   0:04.77 events/0
    6 root      10  -5     0    0    0 S  0.0  0.0   0:01.84 khelper
    7 root      11  -5     0    0    0 S  0.0  0.0   0:00.06 kthread
   10 root      10  -5     0    0    0 S  0.0  0.0   0:00.72 kblockd/0
   11 root      20  -5     0    0    0 S  0.0  0.0   0:00.00 kacpid
   47 root      20  -5     0    0    0 S  0.0  0.0   0:00.00 cqueue/0
   50 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 khubd
   52 root      10  -5     0    0    0 S  0.0  0.0   0:00.02 kseriod
```



CPU Usage: 10 %

Memory: 9.91 GB

Physical Memory (MB)
Total 16367
Cached 4578
Available 6211
Free 1858

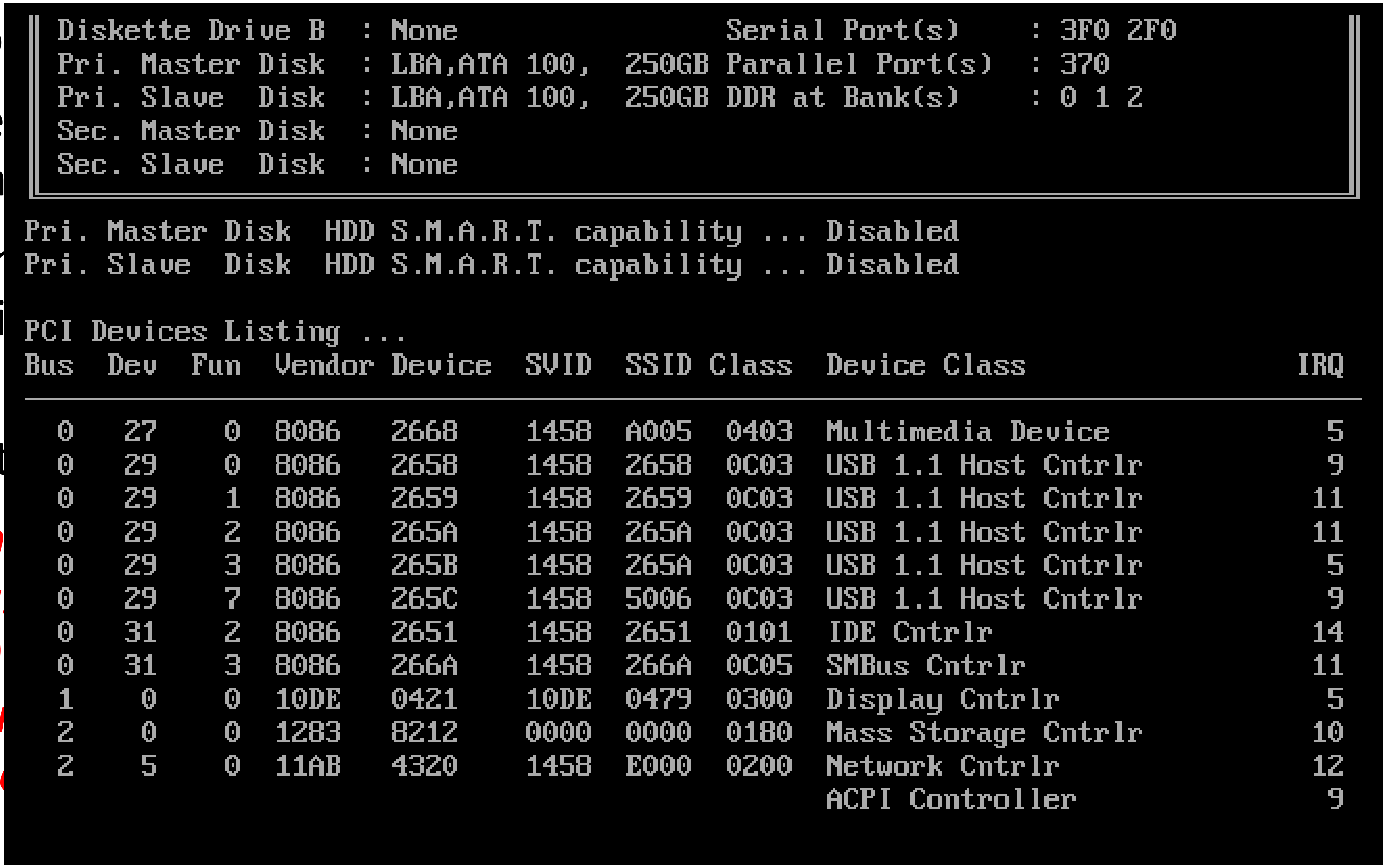
Kernel Memory (MB)
Paged 324
Nonpaged 188

System
Handles 74736
Threads 2034
Processes 118
Up Time 12:05:15:19
Commit (GB) 13 / 31

Processes: 118 | CPU Usage: 10% | Physical Memory: 62%

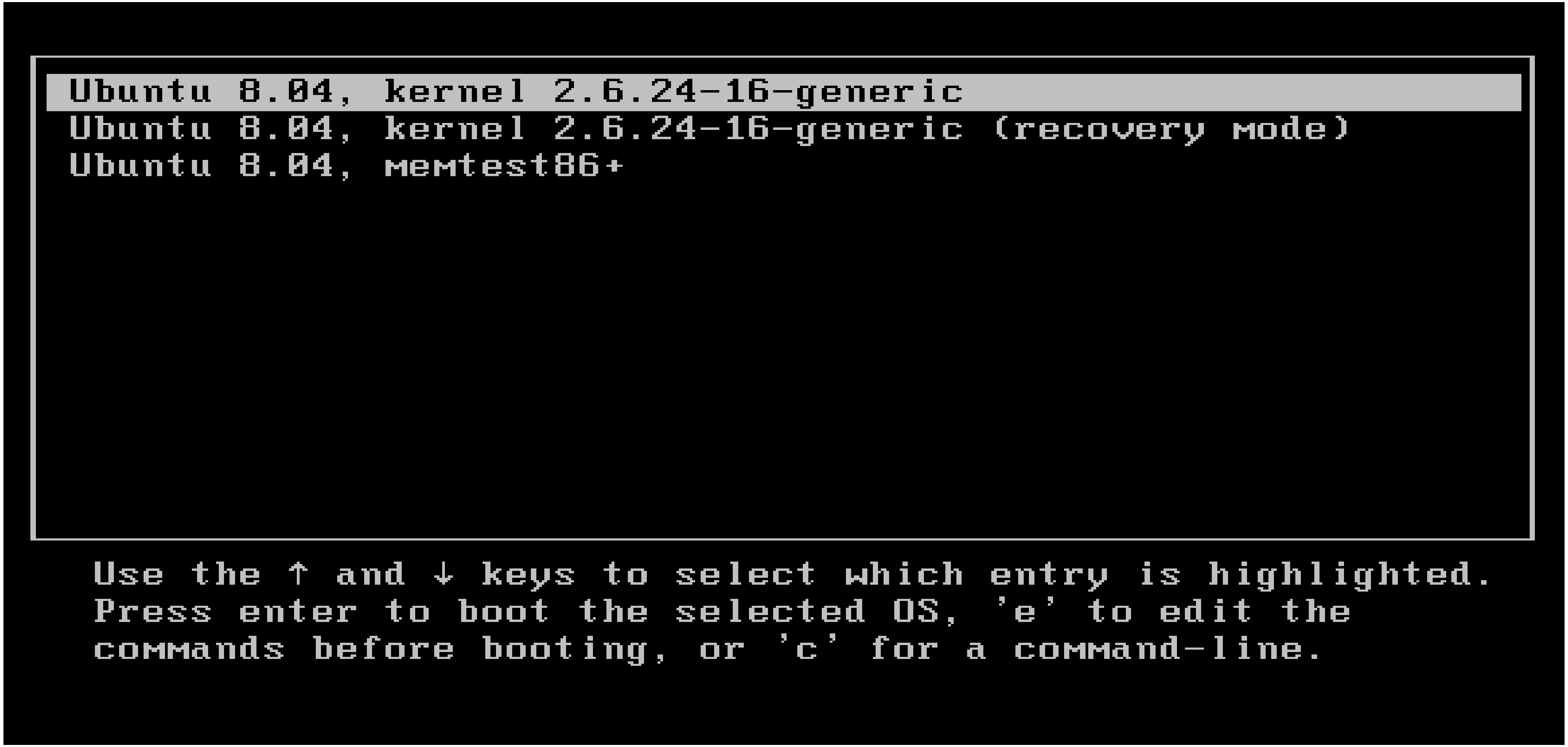
System Boot

- bootstrap program / bootstrap
- simple bootstrap loader fetche boot program from disk, which
- program counter is loaded with memory location where the ini program is located
- diagnostics to determine the st
- *POST (Power-On Self-Test) is th sequence that a computer's ba system (or "starting program") the computer keyboard, rando disk drives, and other hardwar correctly*



```
 Diskette Drive B  : None             Serial Port(s)    : 3F0 2F0
 Pri. Master Disk  : LBA,ATA 100,  250GB Parallel Port(s)  : 370
 Pri. Slave  Disk  : LBA,ATA 100,  250GB DDR at Bank(s)    : 0 1 2
 Sec. Master Disk  : None
 Sec. Slave  Disk  : None

Pri. Master Disk  HDD S.M.A.R.T. capability ... Disabled
Pri. Slave  Disk  HDD S.M.A.R.T. capability ... Disabled

PCI Devices Listing ...
Bus  Dev  Fun  Vendor Device  SVID  SSID Class  Device Class            IRQ

 0   27    0  8086   2668    1458  A005  0403  Multimedia Device         5
 0   29    0  8086   2658    1458  2658  0C03  USB 1.1 Host Cntrlr       9
 0   29    1  8086   2659    1458  2659  0C03  USB 1.1 Host Cntrlr      11
 0   29    2  8086   265A    1458  265A  0C03  USB 1.1 Host Cntrlr      11
 0   29    3  8086   265B    1458  265A  0C03  USB 1.1 Host Cntrlr       5
 0   29    7  8086   265C    1458  5006  0C03  USB 1.1 Host Cntrlr       9
 0   31    2  8086   2651    1458  2651  0101  IDE Cntrlr               14
 0   31    3  8086   266A    1458  266A  0C05  SMBus Cntrlr             11
 1    0    0  10DE   0421    10DE  0479  0300  Display Cntrlr            5
 2    0    0  1283   8212    0000  0000  0180  Mass Storage Cntrlr      10
 2    5    0  11AB   4320    1458  E000  0200  Network Cntrlr           12
                                               ACPI Controller           9
```

System Boot



```
 Ubuntu 8.04, kernel 2.6.24-16-generic
 Ubuntu 8.04, kernel 2.6.24-16-generic (recovery mode)
 Ubuntu 8.04, memtest86+

    Use the ↑ and ↓ keys to select which entry is highlighted.
    Press enter to boot the selected OS, 'e' to edit the
    commands before booting, or 'c' for a command-line.
```

Thank You