CSE321: Operating Systems

Introduction

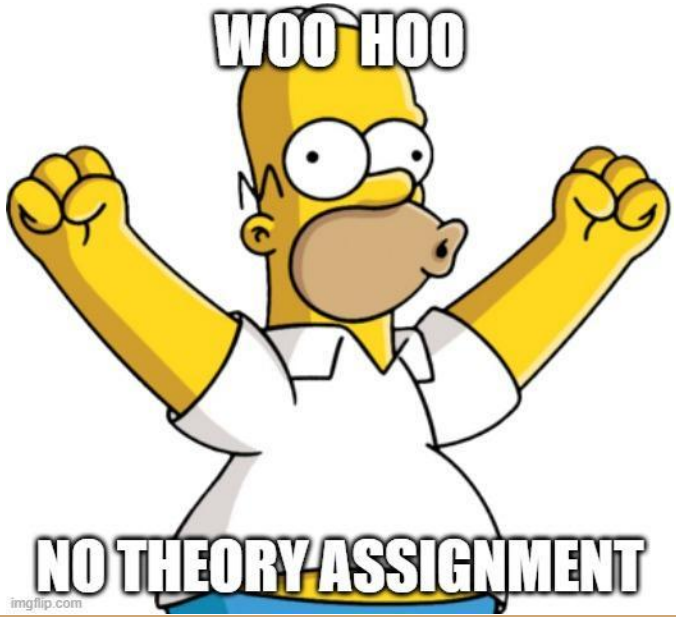
FBA

Course Outcome

- **To understand** the fundamental concepts of computer system organization and the structure of operating systems.
- **To explore** various aspects of process management in operating system
- **To know** how different CPU scheduling algorithm works and their respective importance
- **To develop practical knowledge on** the concept of threads
- **To inspect** process synchronization mechanisms and deadlocks
- **To be able to analyze** the management of main and virtual memory

Marks Distribution

- Theory – 80%
  - Class participation – 5%
  - Quiz – 15%
  - Mid – 25%
  - Final – 35%
- Lab – 20%



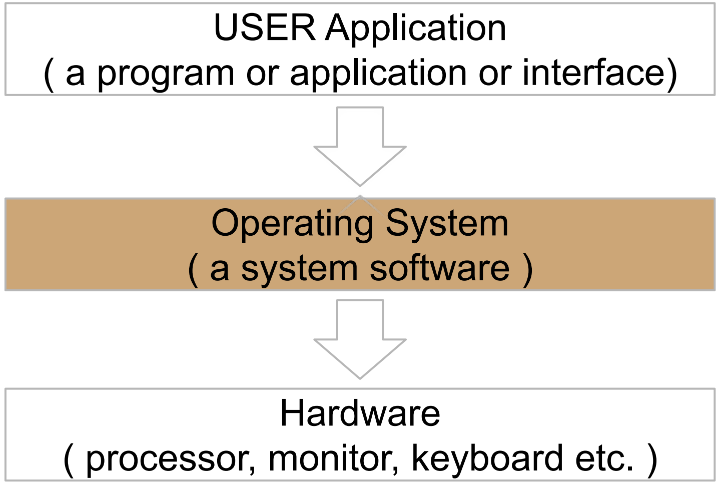
Operating Systems

"Actual" Introduction

FBA

What is an Operating System?

A program that acts as an intermediary
between a user of a computer and the computer hardware.


USER Application
(a program or application or interface)
Operating System
(a system software)
Hardware
(processor, monitor, keyboard etc.)

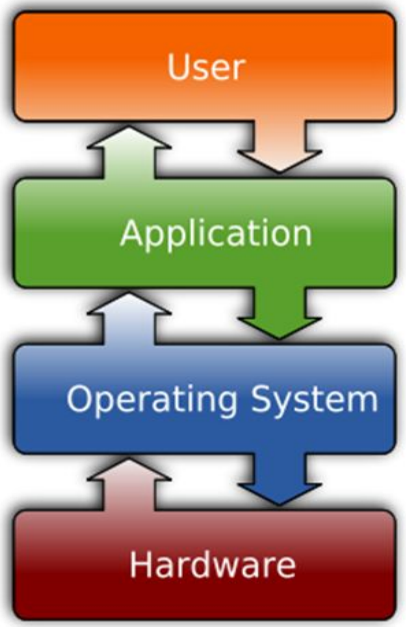
System Software Vs Application Software

System Software:

- System Software refers to the operating system and all utility programs that manage computer resources at a low level.
- Systems software includes compilers, loaders, linkers, and debuggers.

Application Software:

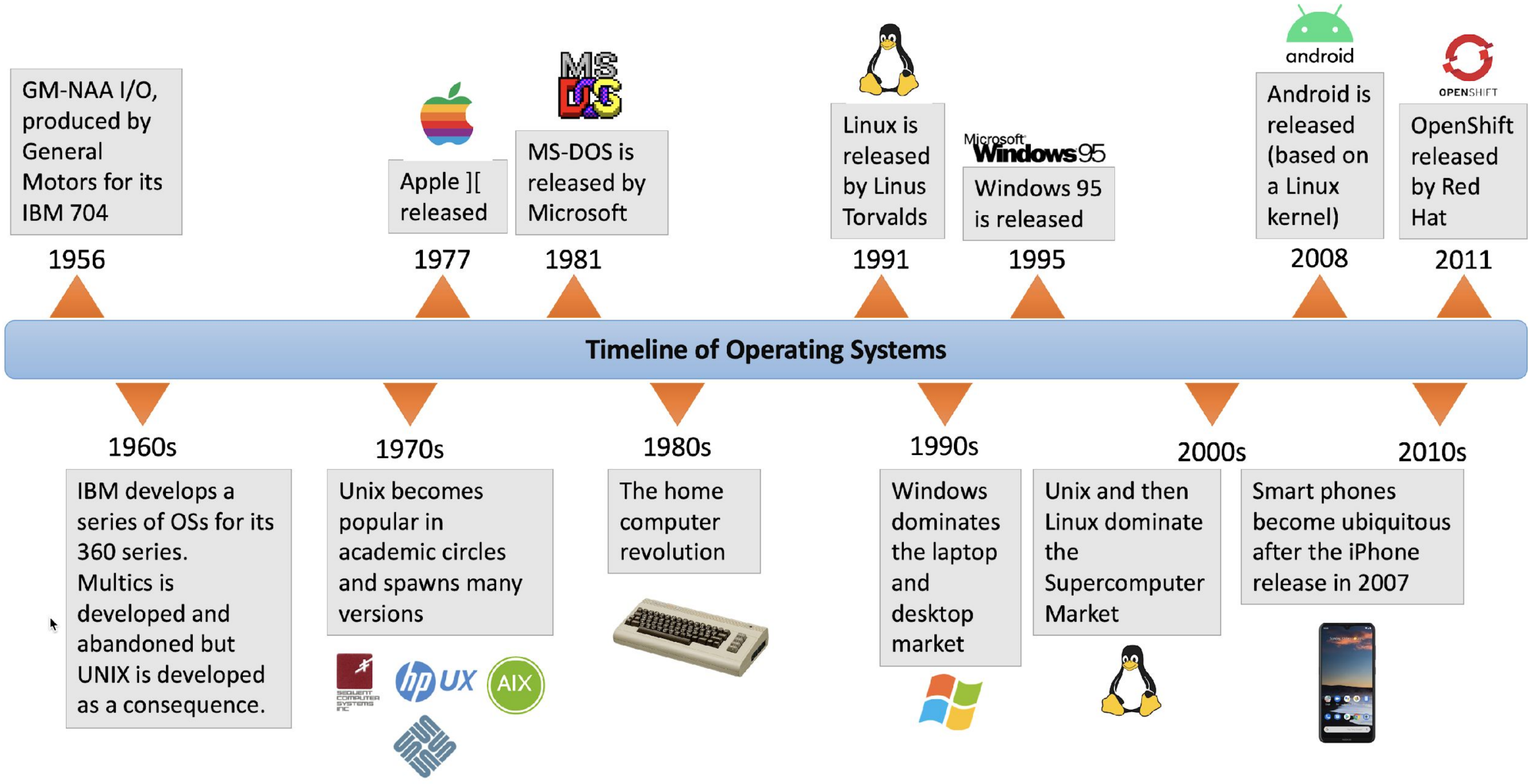
- Applications software comprises programs designed for an end user, such as word processors, database systems, and spreadsheet programs.



Major Goals of OS

- Execute user programs.
- Make the computer system convenient to use.
- Use the computer hardware in an efficient manner
- Manages and allocate all resources
- Controls the execution of user programs and operations of I/O devices

Timeline of OS



Timeline of Operating Systems

1956
GM-NAA I/O, produced by General Motors for its IBM 704

1960s
IBM develops a series of OSs for its 360 series. Multics is developed and abandoned but UNIX is developed as a consequence.

1970s
Unix becomes popular in academic circles and spawns many versions

1977
Apple][released

1981
MS-DOS is released by Microsoft

1980s
The home computer revolution

1991
Linux is released by Linus Torvalds

1990s
Windows dominates the laptop and desktop market

1995
Windows 95 is released

2000s
Unix and then Linux dominate the Supercomputer Market

2008
Android is released (based on a Linux kernel)

2011
OpenShift released by Red Hat

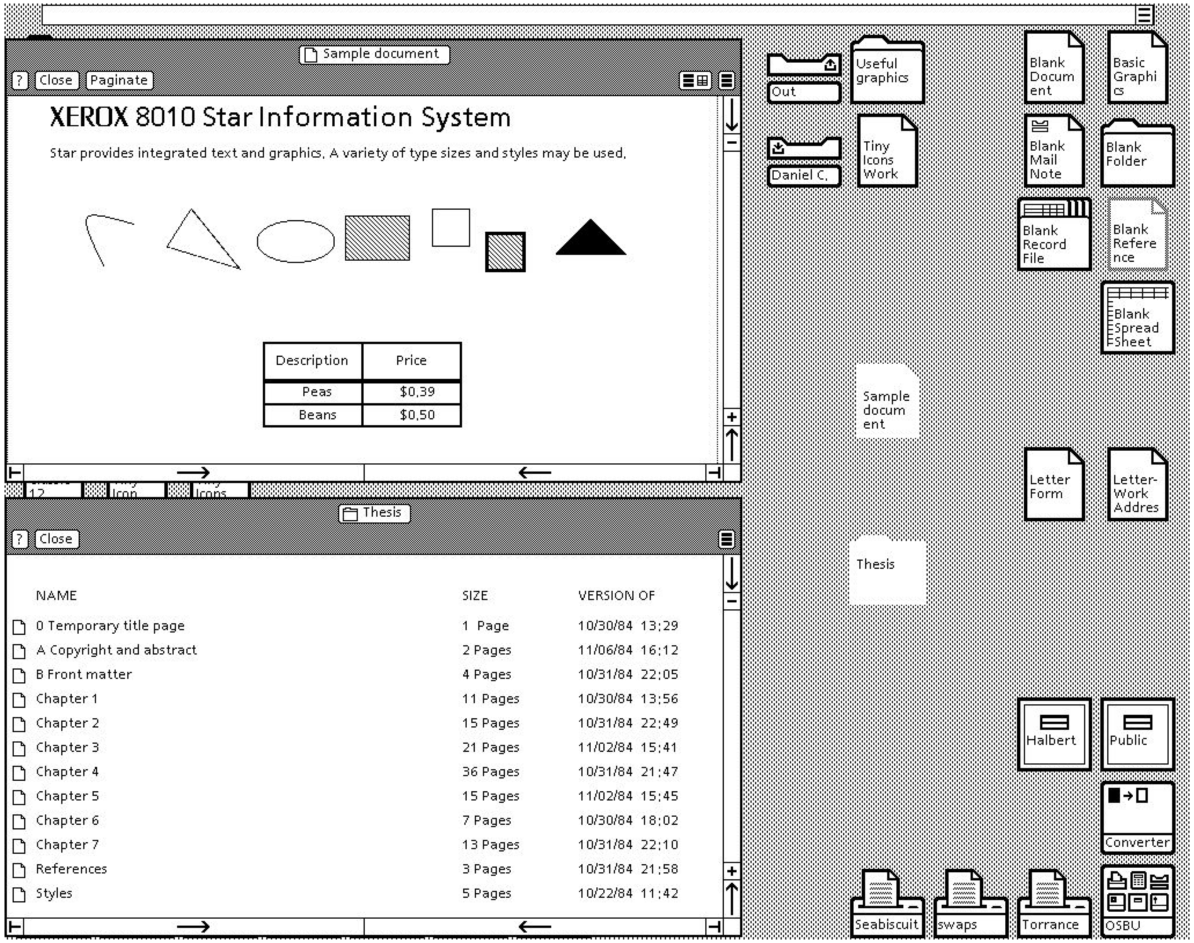
2010s
Smart phones become ubiquitous after the iPhone release in 2007

Timeline of OS

- IBM 704 was the first mass-produced computer
- GM-NAA (General Motors-North American Aviation) I/O is the OS used in the IBM 704
- There was no UI in the OS as we know now
- It was mostly based on command prompts (terminals)

Timeline of OS

- Xerox 8010 Star (released in 1981) was the first system that was referred to as a fully integrated desktop computer including applications and a GUI



XEROX 8010 Star Information System

Star provides integrated text and graphics. A variety of type sizes and styles may be used.

Description	Price
Peas	$0.39
Beans	$0.50

NAME	SIZE	VERSION OF
0 Temporary title page	1 Page	10/30/84 13:29
A Copyright and abstract	2 Pages	11/06/84 16:12
B Front matter	4 Pages	10/31/84 22:05
Chapter 1	11 Pages	10/30/84 13:56
Chapter 2	15 Pages	10/31/84 22:49
Chapter 3	21 Pages	11/02/84 15:41
Chapter 4	36 Pages	10/31/84 21:47
Chapter 5	15 Pages	11/02/84 15:45
Chapter 6	7 Pages	10/30/84 18:02
Chapter 7	13 Pages	10/31/84 22:10
References	3 Pages	10/31/84 21:58
Styles	5 Pages	10/22/84 11:42

Timeline of OS

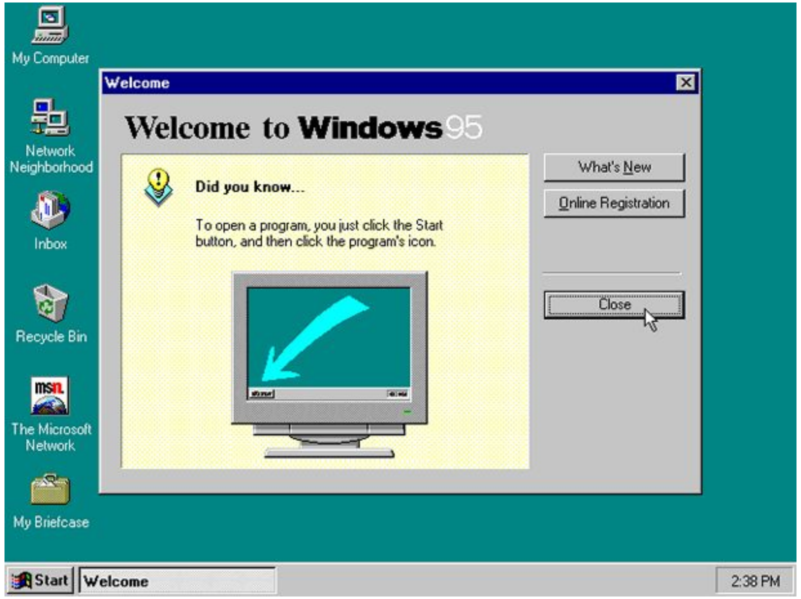
- Bill Gates got lucky to sign the so-called “Deal of the Century” with IBM to design a new OS for IBM machines
- The deal allowed Microsoft to add an OS with $50 per PC and IBM did not own copyright for the OS
- This allowed Microsoft to start the their OS dominance on the PC domain

Timeline of OS

- Developed by Apple Computer, Inc for their new product, the Macintosh home PC, The Macintosh 128K, was released in 1984
- was widely advertised (the famous 1984 commercial is available below).
- Mac OS was the first OS with a GUI built-in
- It was also one of the first consumer computers with mouse!
- Even though Microsoft introduced mouse in their PCs a bit earlier
- The use of GUI with mouse was not Steve Job’s idea, the idea was taken during his visit in Xerox PARC (Palo Aalto Research Center)
- Jobs reportedly traded US $1 million in stock options to Xerox for a detailed tour of their facilities and current projects
- One of the things Xerox showed Jobs was the Alto, which sported a GUI and a three-button mouse

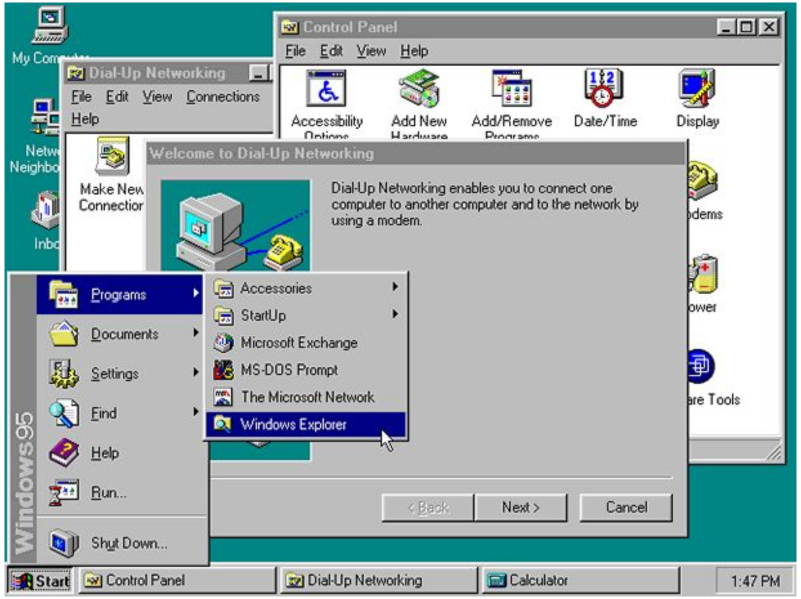
Timeline of OS

Timeline of OS


Welcome to Windows 95

Did you know...

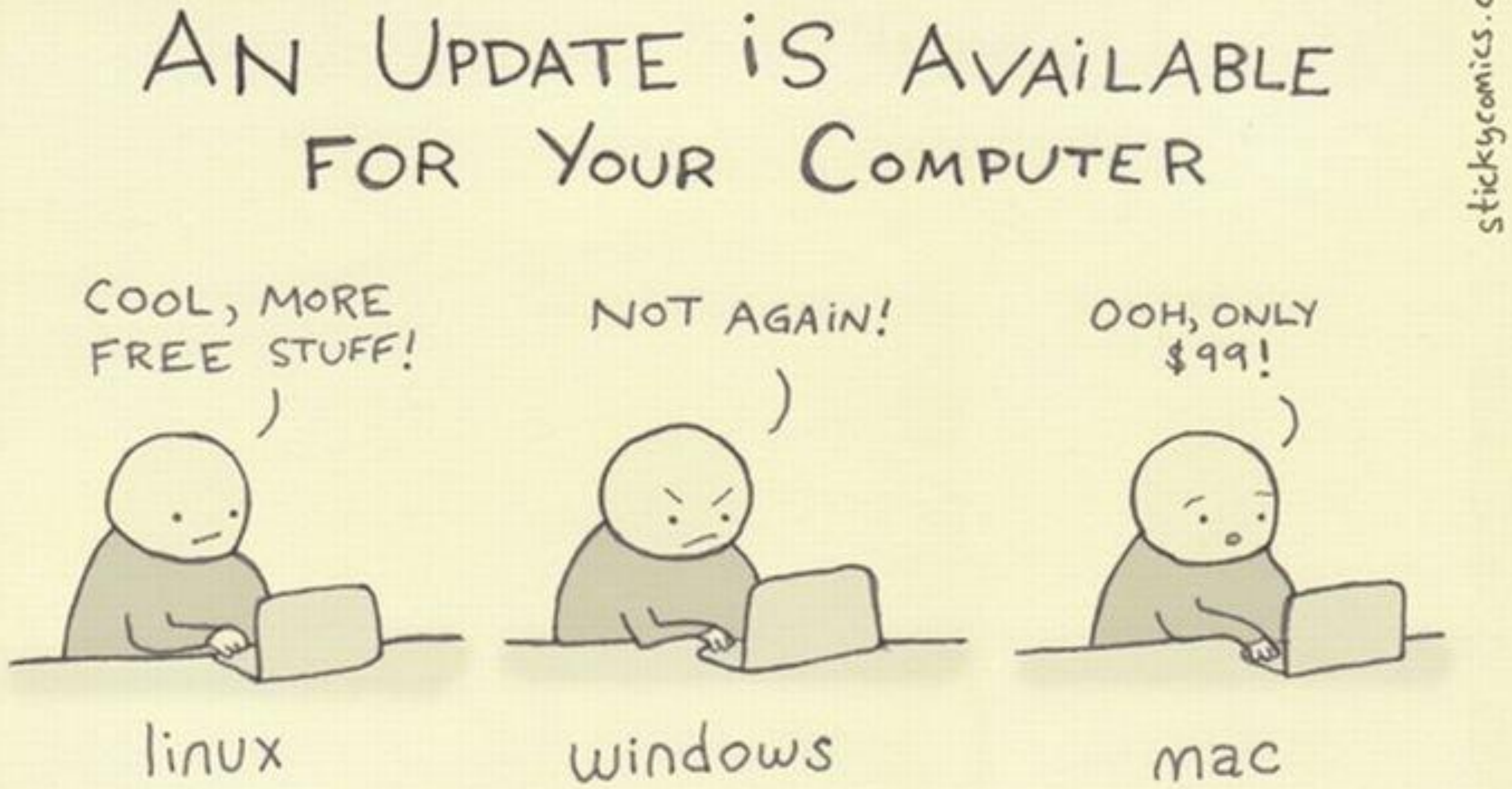
To open a program, you just click the Start button, and then click the program's icon.


Welcome to Dial-Up Networking

Dial-Up Networking enables you to connect one computer to another computer and to the network by using a modem.

Timeline of OS

- Another revolution of OS came in the mobile computing domain, when Steve Jobs introduced iPhone with iOS in 2007
- The iPhone introduction video is now regarded as a classic advertise video

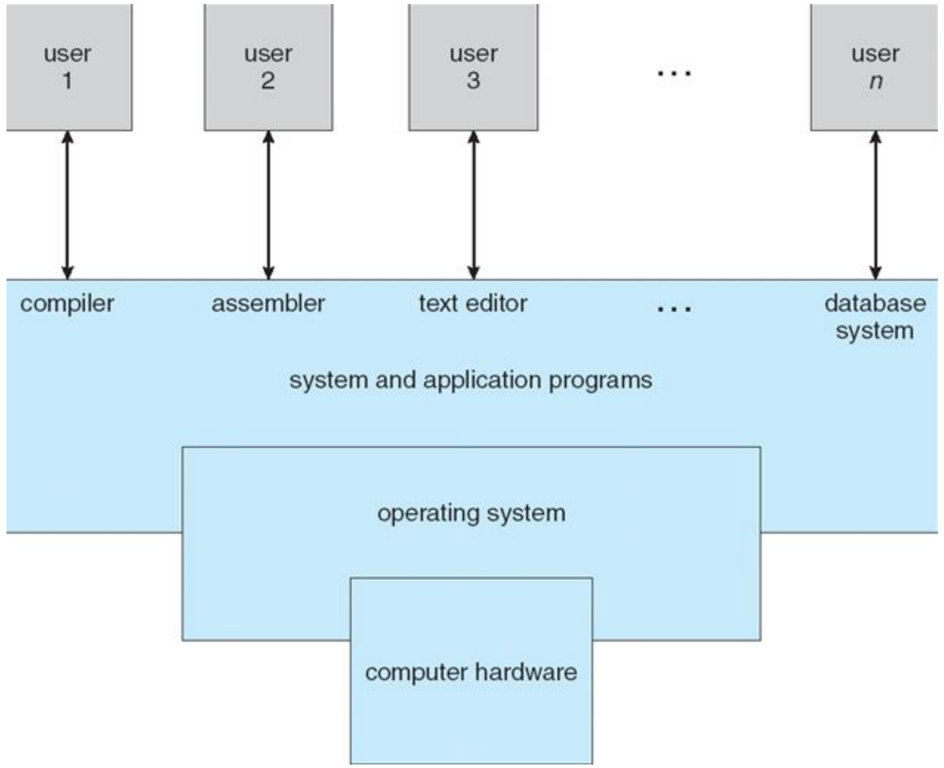

AN UPDATE IS AVAILABLE FOR YOUR COMPUTER
COOL, MORE FREE STUFF!
NOT AGAIN!
OOH, ONLY $99!
linux
windows
mac
stickycomics.com

Operating Systems

Computer System Organization

FBA

Components of a Computer System



Kernel

The one program running at all times.

- Kernel is the central module of an operating system
- Part of OS that loads first, and it remains in main memory.
- As small as possible
- Provide all the essential services required by other parts of the operating system and applications.
- Kernel code is usually loaded into a protected area of memory to prevent it from being overwritten.

Bootstrap Program

An initial program executed when a computer starts running.

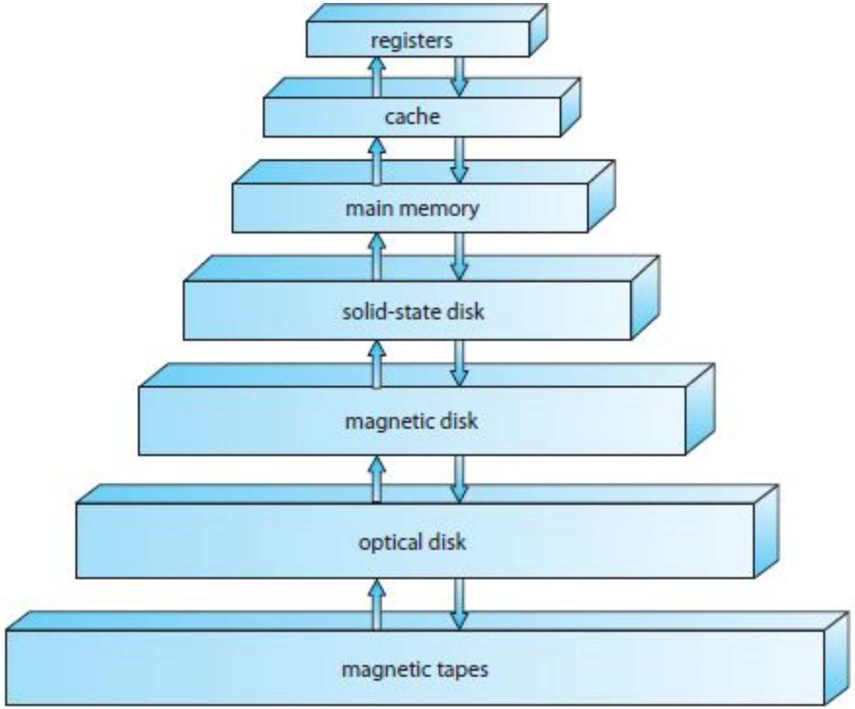
- When a computers is powered up or rebooted, it is executed first.
- Stored in the ROM or EEPROM, known as firmware
- Initializes all aspects of the system, from CPU registers to device controllers to memory contents
- bootstrap program must know how to load the operating system and how to start executing
- Once the OS kernel is loaded and executing, it can start providing services to the system and its users

Storage Structure

- Main memory – only large storage media that the CPU can access directly
  - Random access
  - Typically volatile
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity

- CPU can load instructions only from main memory.
- General-purpose computers run most of their programs from rewritable memory, called main memory (also called RAM)
- Computers use other forms of memory as well - Read Only Memory (ROM) and electrically erasable programmable read-only memory (EEPROM)
- Only static programs, such as the bootstrap program described earlier, are stored here.

Storage Device Hierarchy



Operating Systems

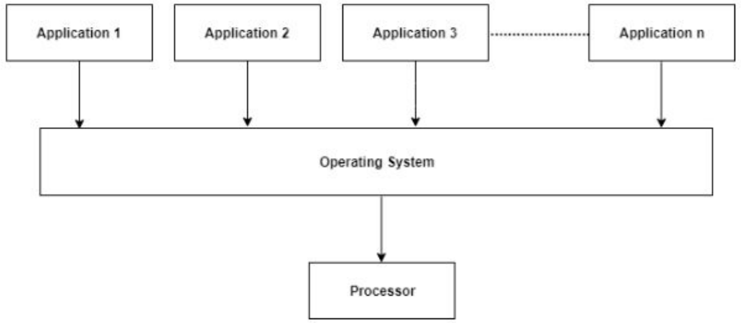
OS Architecture

FBA

Operating System Architecture

Single-Processor Systems:

- One main CPU capable of executing a general-purpose instruction set.
- Almost all single processor systems have other special-purpose processors (device-specific processors), which run a limiter instruction set.

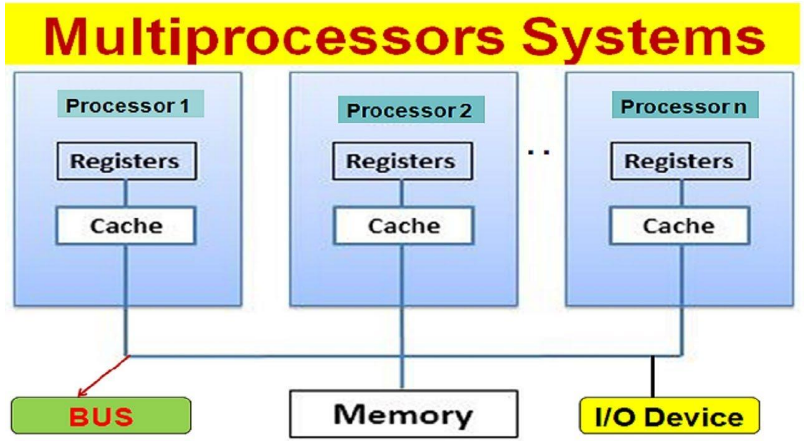


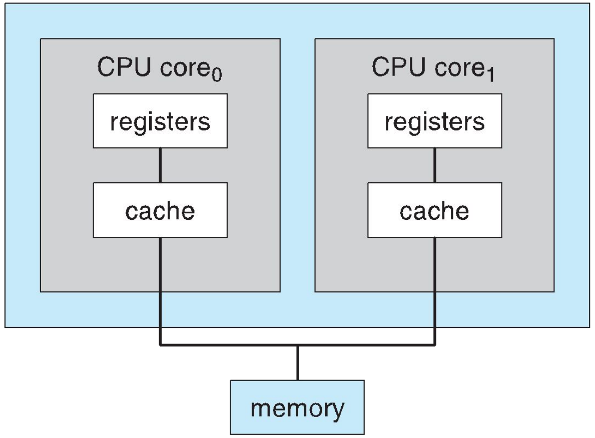
Single Processor System

Operating System Architecture

Multiprocessor Systems (parallel systems or multicore systems):

- Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.



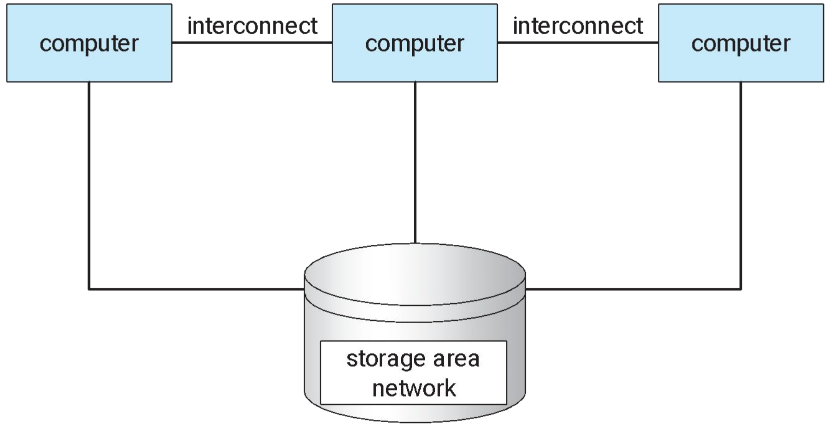


A dual-core system

Operating System Architecture

Clustered Systems:

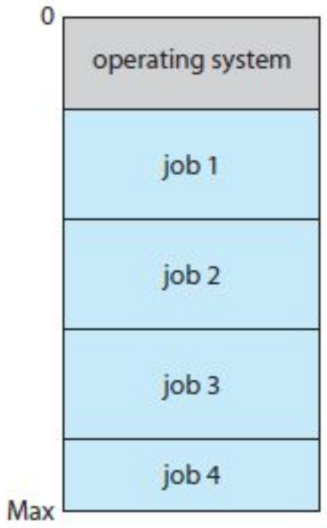
- Special kind of multiprocessor system which gathers together multiple CPUs. They are composed of two or more individual systems- or nodes - joined together.
- Clustered computers share storage and are closely linked via a local-area network (LAN) or a faster interconnect, such as InfiniBand



Operating System Structure

Multiprogramming:

- Increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.
- OS keeps several jobs in memory simultaneously. As main memory is small, it keeps the jobs on the disk (job pool) which waits there to be allocated in the main memory.
- OS picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task. OS picks another job from the pool to execute while the previous one waits.



Memory layout for a multiprogramming system.

Requirements of Multiprogramming

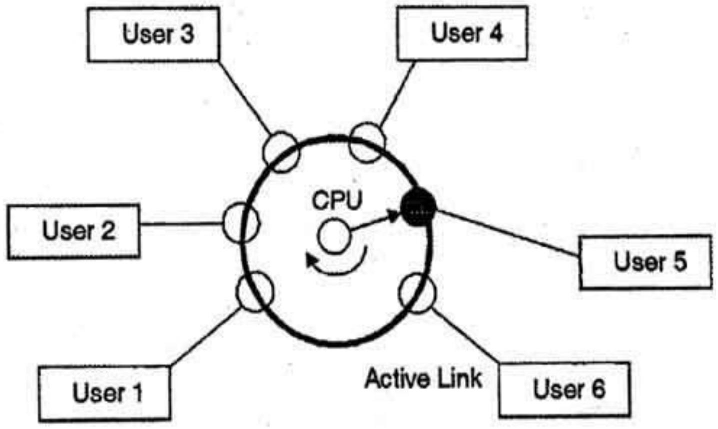
- **Job Scheduling:** If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them.
- When OS selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management.
- **CPU Scheduling:** if several jobs are ready to run at the same time, the system must choose which job will run first.
- Running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system.

★ If processes don't fit in memory, swapping moves them in and out to run from main memory achieving this goal is virtual memory.

Operating System Structure

Time Sharing:

- CPU executes multiple jobs by switching among them.
- Switches occur so frequently that the users can interact with each program while it is running.
- requires an interactive computer system, which provides direct communication between the user and the system.
- Response time should be short.

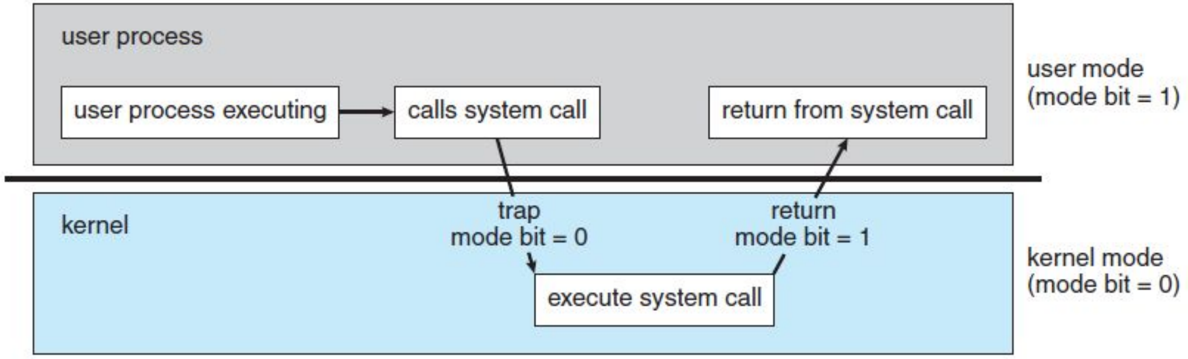


Operating System Operations

- Modern operating systems are interrupt driven
- Events are almost always signaled by the occurrence of an interrupt or a trap
- A trap (or an exception) is a software-generated interrupt
- For each type of interrupt, separate segments of code determine what action should be taken
- Errors can occur when one erroneous program modify another program, the data of another program, or even the operating system itself.
- A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

Dual Mode Operation

- Need to distinguish between the execution of operating-system code and user defined code.
- A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).
- dual mode of operation provides protection of the operating system from errant users
- this protection is provided by designating some of the machine instructions that may cause harm as privileged instructions that are executed only in kernel mode.



Transition from user to kernel mode.

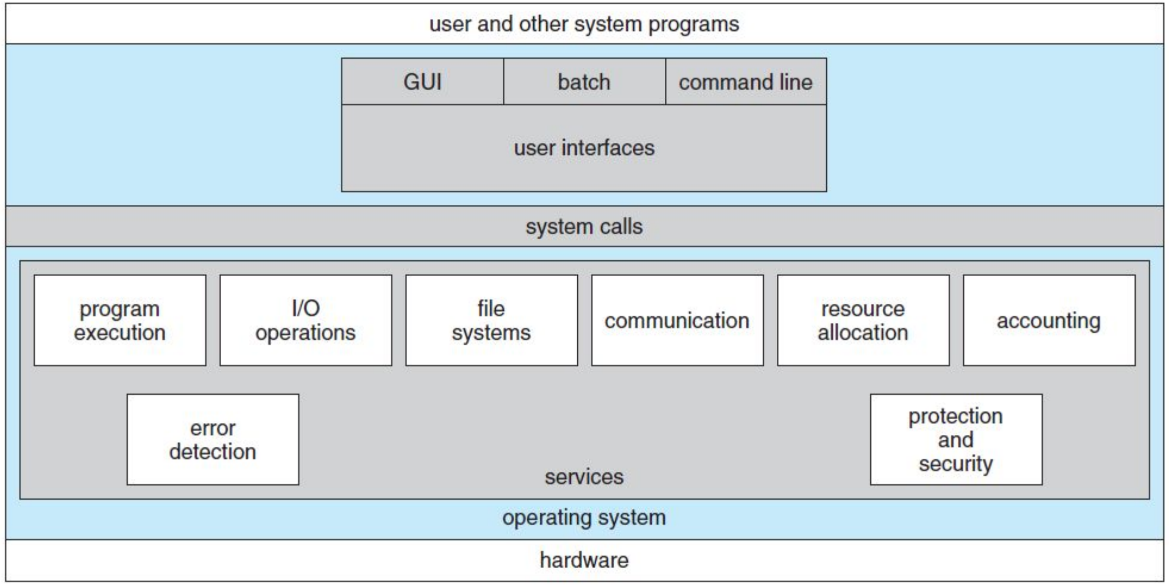
Operating Systems

OS Services

FBA

Operating System Services

- OS provides an environment for the execution of programs.
- Specific services provided, differ from one operating system to another, but there are some common classes
- Services are provided for the convenience of the programmer



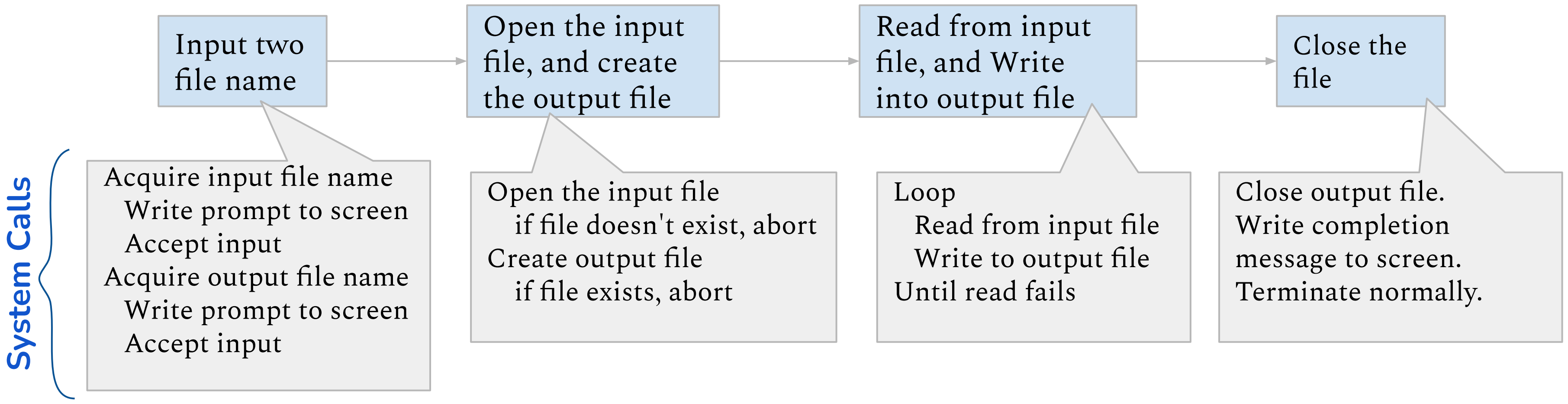
Operating Systems

System Call, System Program, System Boot

System Call

- System calls provide an interface to the services made available by an operating system.
- These calls are generally available as routines.
- Routines are written in C or C++. Some low level tasks are written in assembly language.

A program to copy the contents of a file to another file !



Input two file name
Open the input file, and create the output file
Read from input file, and Write into output file
Close the file
System Calls

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
```

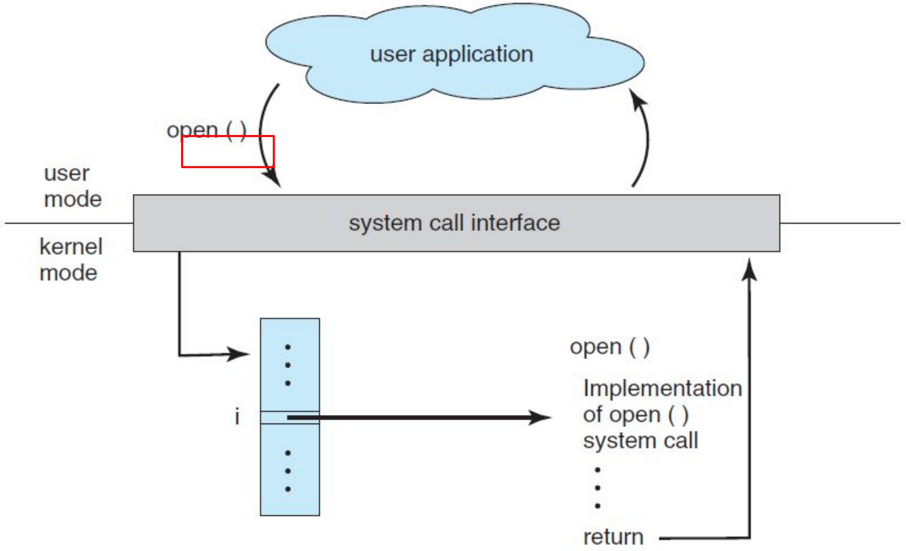
```
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
```

```
Loop
  Read from input file
  Write to output file
Until read fails
```

```
Close output file.
Write completion
message to screen.
Terminate normally.
```

System Call Interface

- Serves as the link to system calls made available by the operating system.
- A number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.
- Invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values.



Types of System Call

Type	Windows OS	Linux OS

Process Control	**CreateProcess()**
	ExitProcess()
	WaitForSingleObject()
		fork()
		exit()
		wait()

File Manipulation	**CreateFile()**
	ReadFile()
	WriteFile()
	CloseHandle()
		open()
		read()
		write()
		close()

Device Manipulation	**SetConsoleMode()**
	ReadConsole()
	WriteConsole()
		ioctl()
		read()
		write()

Types of System Call

Type	Windows OS	Linux OS

Information Maintenance	**GetCurrentProcessID()**
	SetTimer()
	Sleep()
		getpid()
		alarm()
		sleep()

Communication	**CreatePipe()**
	CreateFileMapping()
	MapViewOfFile()
		pipe()
		shm_open()
		mmap()

Protection	**SetFileSecurity()**
	InitlializeSecurityDescriptor()
	SetSecurityDescriptorGroup()
		chmod()
		umask()
		chown()

System Programs

System programs, also known as system utilities, provide a convenient environment for program development and execution.

These system programs provide -

- File management
- Status information
- File modification
- Programming-language support
- Program loading and execution
- Communications
- Background services

System Boot

- When power initialized on system, execution starts at a fixed memory location.
  - ➔ Firmware ROM used to hold initial boot code

- Operating system must be made available to hardware so hardware can start it.
  - ➔ Small piece of code – bootstrap loader, stored in ROM locates the kernel, loads it into memory, and starts it
  - ➔ Sometimes two-step process where boot block at fixed location loaded by ROM code, which loads bootstrap loader from disk

- Common bootstrap loader, GRUB, allows selection of kernel from multiple disks, versions, kernel options

- Kernel loads and system is then running.

Operating Systems

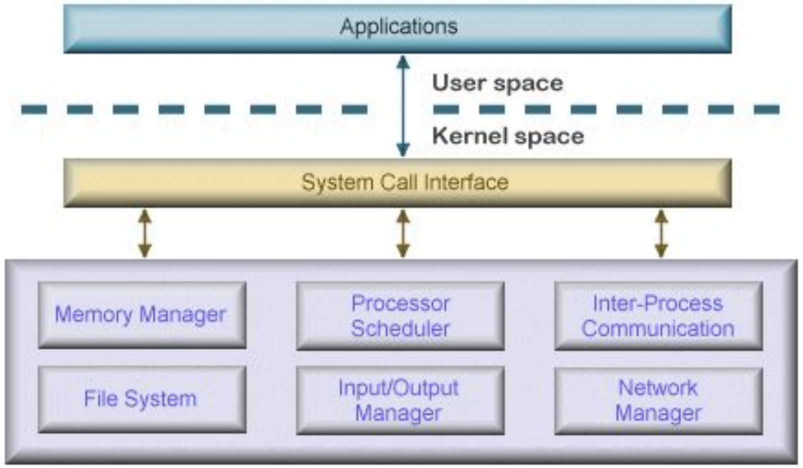
OS Structures

FBA

OS Structure

Simple/Monolithic structure:

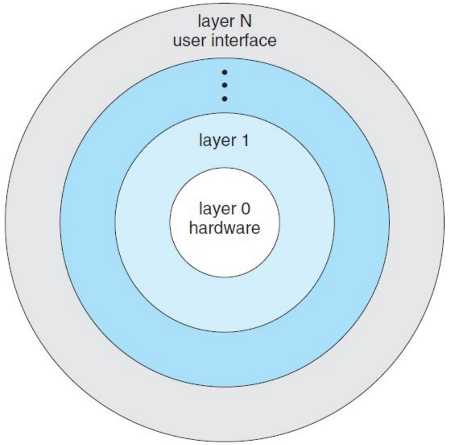
- Earliest and most common architecture
- Every component of OS is in the kernel and can communicate with each other directly
- Complex and Large (millions line of code ,) hard to maintain



OS Structure

Layered structure:

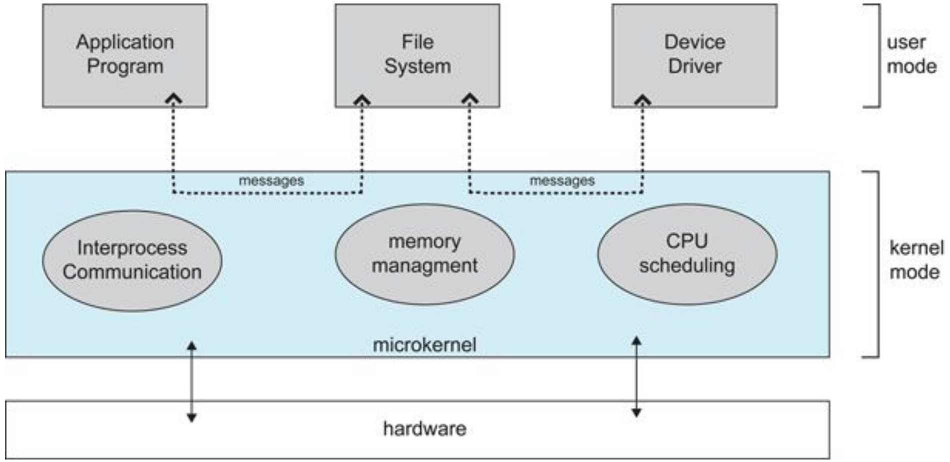
- OS is divided into layers.
- Each layer can use services of its lower layers.
- Easy to debug and develop.
- Less efficient as each layer adds some overhead



OS Structure

Microkernel structure:

- Moves as much from kernel into user space
- Communication takes place between user modules using message passing
- Easier to extend a microkernel
- More reliable(less code running in kernel mode) and secure
- Performance overhead



BRACE YOURSELF
THERE'S MORE TO COME
makeameme.org

Upcoming Episodes

Process:

- **Process States:** A condition of the process at a specific instant of time.
- **Process Architecture:** The structural design of general process systems.
- **Operations on Process:** process creation, preemption, blocking, and termination etc
- **Process Scheduling:** A task that schedules processes of different states like ready, waiting, and running
- **Process Synchronization:** The task of coordinating the execution of processes

Upcoming Episodes

Thread:

- **Definition & Purpose:** a sequential flow of tasks within a process.
- **Multithreading Models:** Ways of achieving multithreading
- **Thread Library:** provides the programmer with an Application program interface for creating and managing thread
- **Threading Issues:** Common problems and pitfalls with multi-thread programming

Upcoming Episodes

Deadlock:

- **Reason**
- **Necessary Conditions for Deadlock**
- **Methods of Handling Deadlocks**

Upcoming Episodes

Memory Management:

- **Background and importance of memory management**
- **Memory Management Techniques**
- **Virtual Memory**

THANK YOU
END OF LECTURE

Operating Systems

Process

FBA

Process Concept

What to call the activities of CPU ?

Jobs

Batch System

User Programs
or
Tasks

Time Sharing
System

These activities are called "**Processes**"

★ The terms "***job***" and "***process***" are used almost interchangeably.

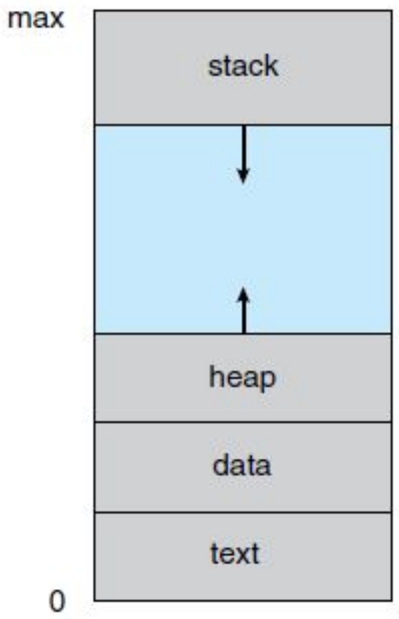
Process

A process is a program that is in execution.

But, it is more than the program codes. Program code is known as "text section" of a process.

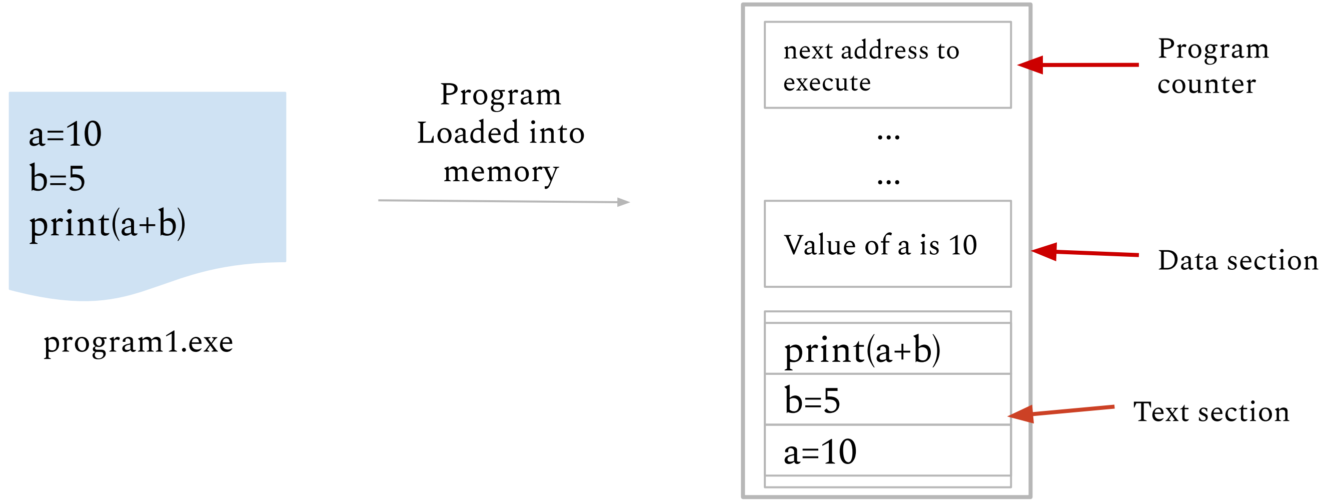
Besides code of the program, it contains -

- **Program Counter and Registers:** stores current activity of the process
- **Stack:** Temporary data (function parameter, local variables, return addresses etc.)
- **Data Section:** Global Variables
- **Heap:** dynamically allocated memory during runtime



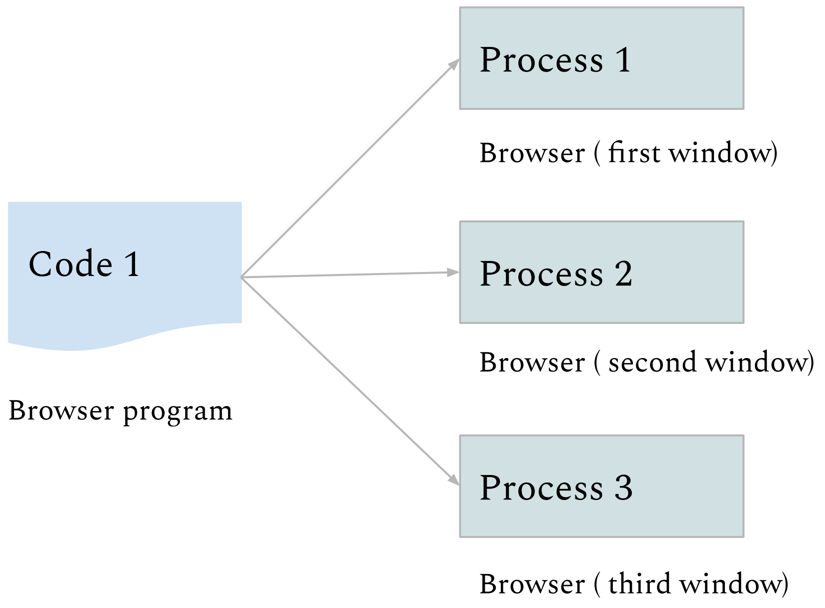
Program Vs Process

- Program is a collection of instructions that can be executed
- A program is a **passive** entity.
- A process is an **active** entity.
- A program becomes a process when it is loaded into memory for execution.



Corresponding process in memory

Same program, Different Process



- Program code is same
- Data, Heap, Stacks contains different information

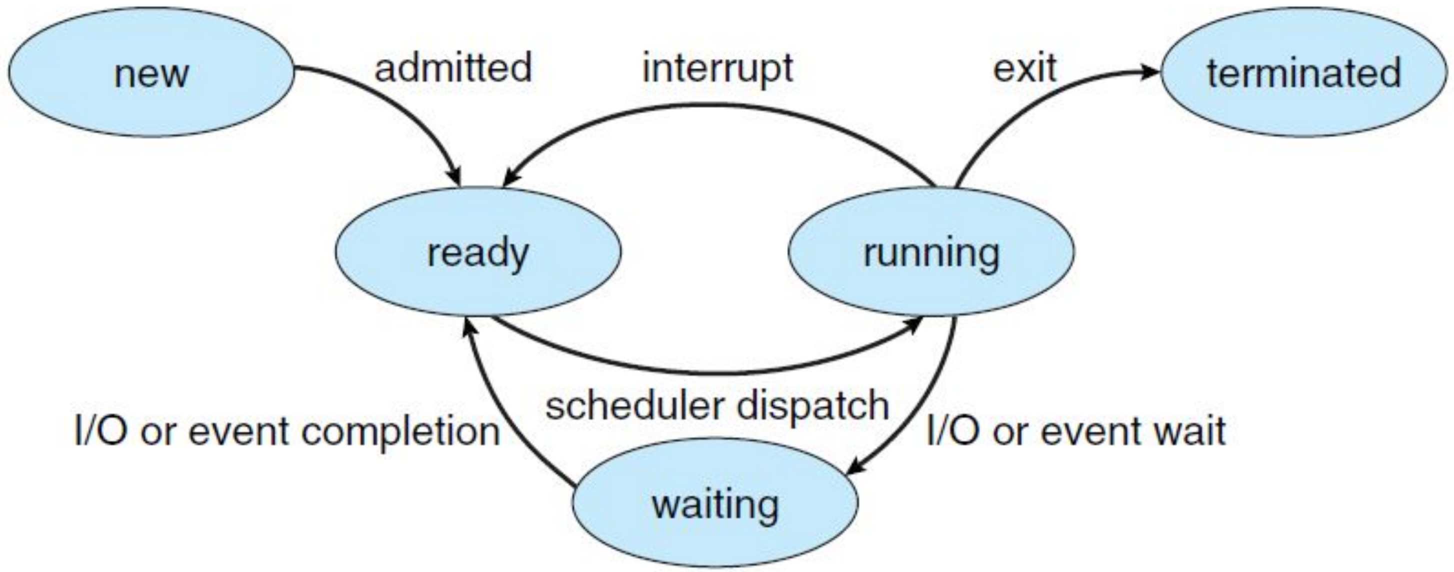
States of a Process

A process state defines the current activity of that process.

The states a process can be:

- ❏ **New**: Process is being created
- ❏ **Running**: Instructions are being executed
- ❏ **Waiting**: Process is waiting for some event to occur
- ❏ **Ready**: Waiting to be assigned to a processor
- ❏ **Terminated**: Process has finished execution

Process State Diagram



Representation of Processes in OS

Each process is represented in the operating system by a ***Process Control Block (PCB)***

PCB is a data structure to store information of Processes such as -

Process state

Program counter

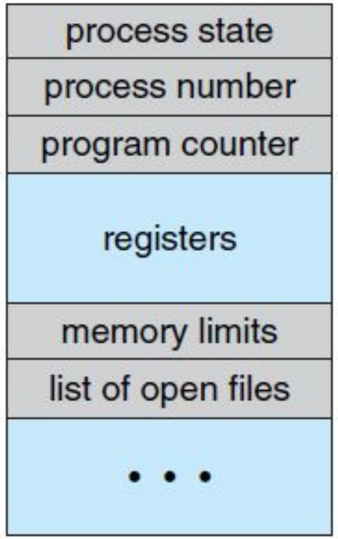
CPU registers

CPU scheduling information

Memory-management information

Accounting information

I/O status information



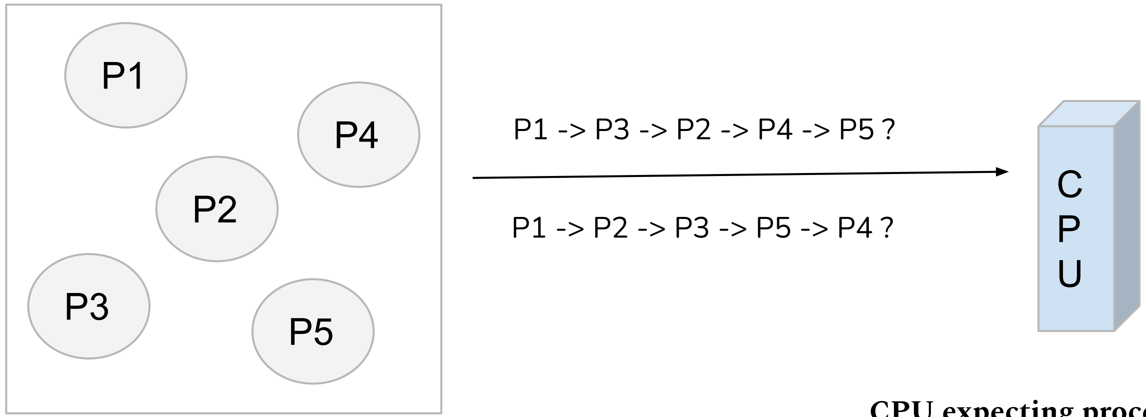
Operating Systems

Process Scheduling

FBA

Process Scheduling

Multiple process is ready to execute.
But, which Process should be executed first?



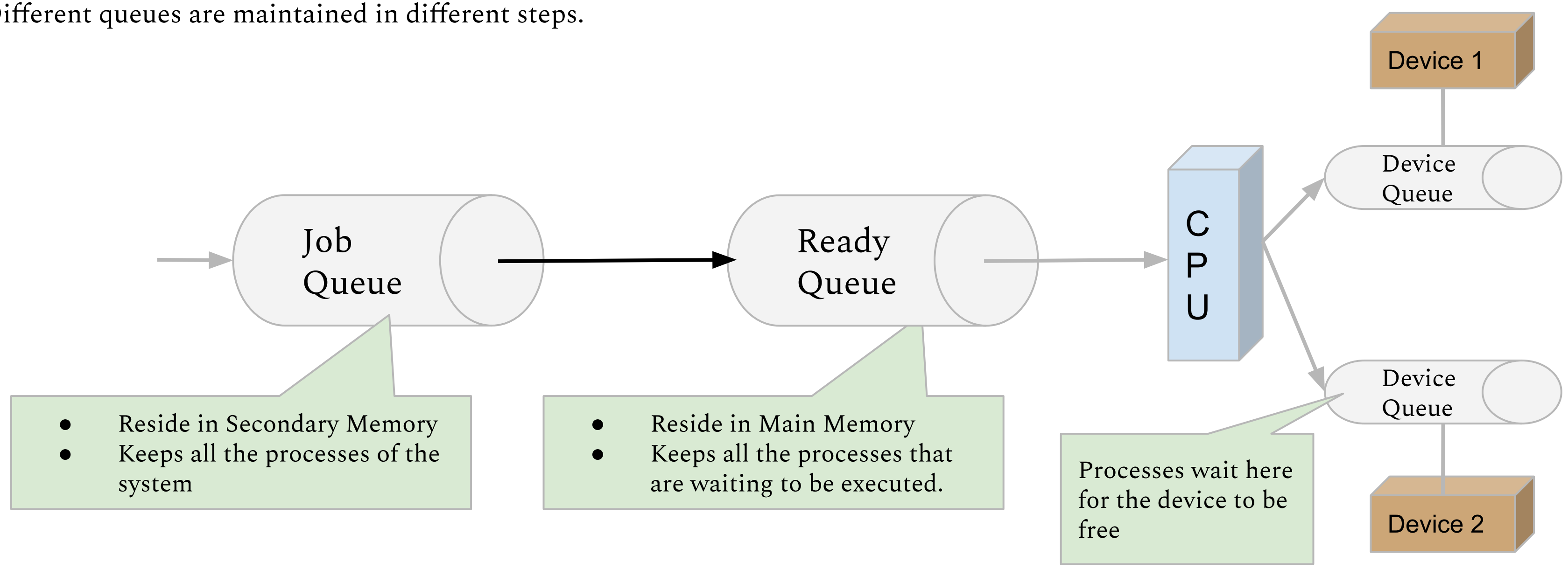
Processes needs to be executed

CPU expecting processes to execute

Scheduling Queue

Stores the processes in different steps of OS.

Different queues are maintained in different steps.



- Reside in Secondary Memory
- Keeps all the processes of the system

- Reside in Main Memory
- Keeps all the processes that are waiting to be executed.

Processes wait here for the device to be free

Queueing Diagram

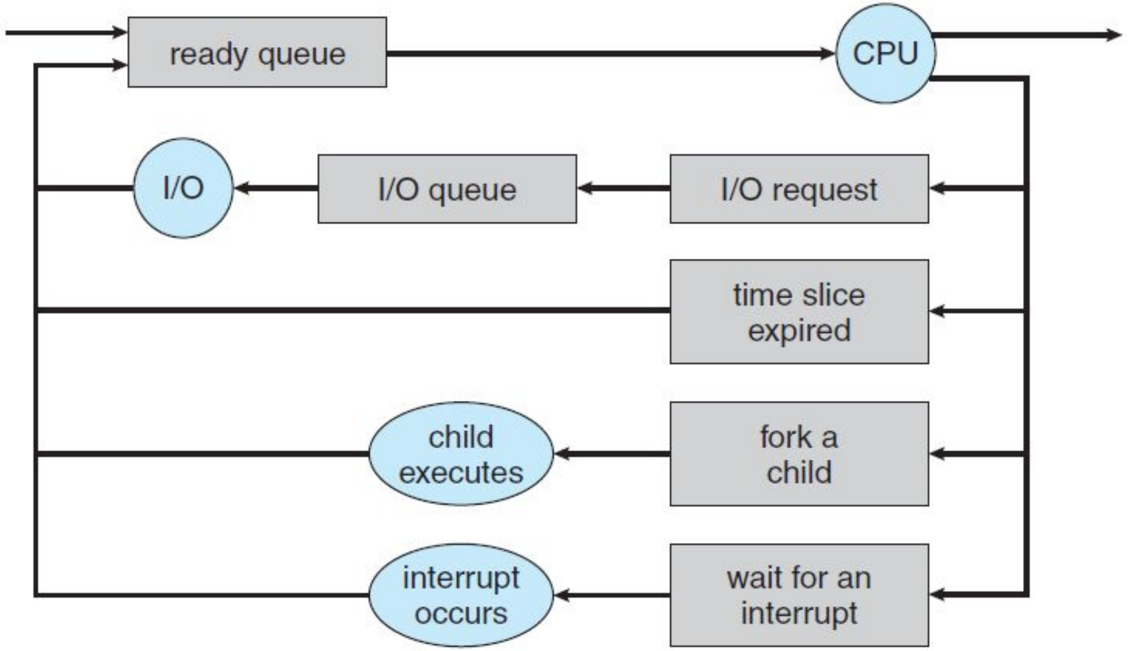
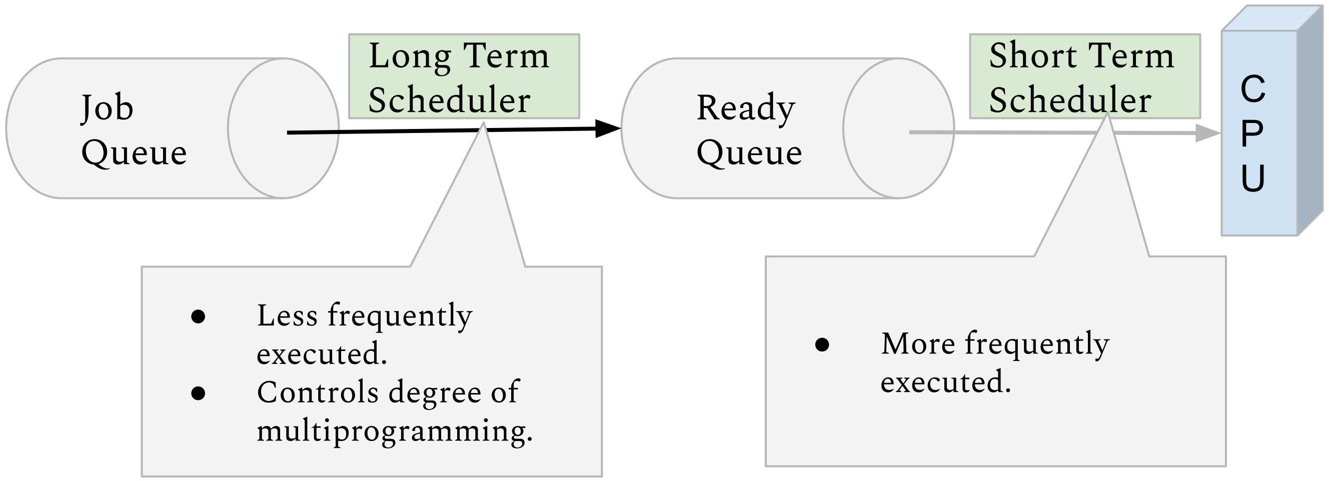


Fig: Representation of Process Scheduling using Queueing-Diagram

Schedulers

Schedulers select processes from different queues to be passed to the next phase.



- Less frequently executed.
- Controls degree of multiprogramming.

- More frequently executed.

CPU Bound Vs I/O Bound Process

- CPU bound processes spend more time doing computation using processors than I/O.
- I/O bound processes spend more time in I/O than CPU.

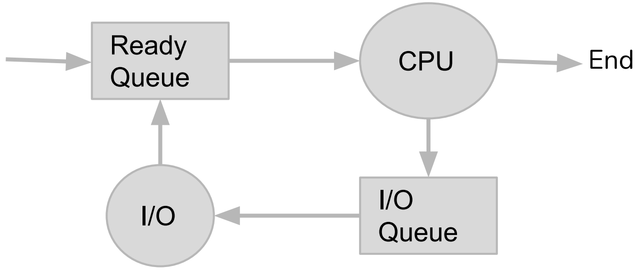
Long Term Scheduler must select wisely !

- What will happen if all processes are I/O bound ?

=> Empty ready queue

- What will happen if all processes are CPU bound ?

=> Empty waiting queue



Medium Term Scheduler

- Time-sharing system may use this scheduler.
- Swapping reduce the degree of multiprogramming.

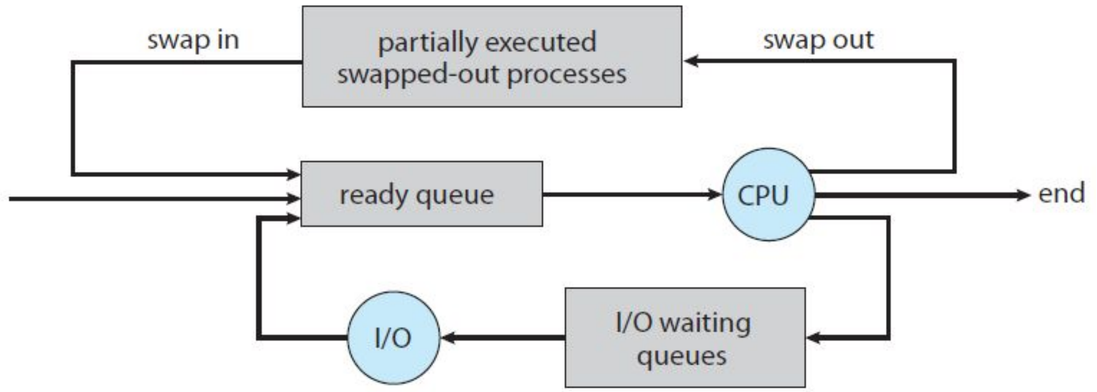
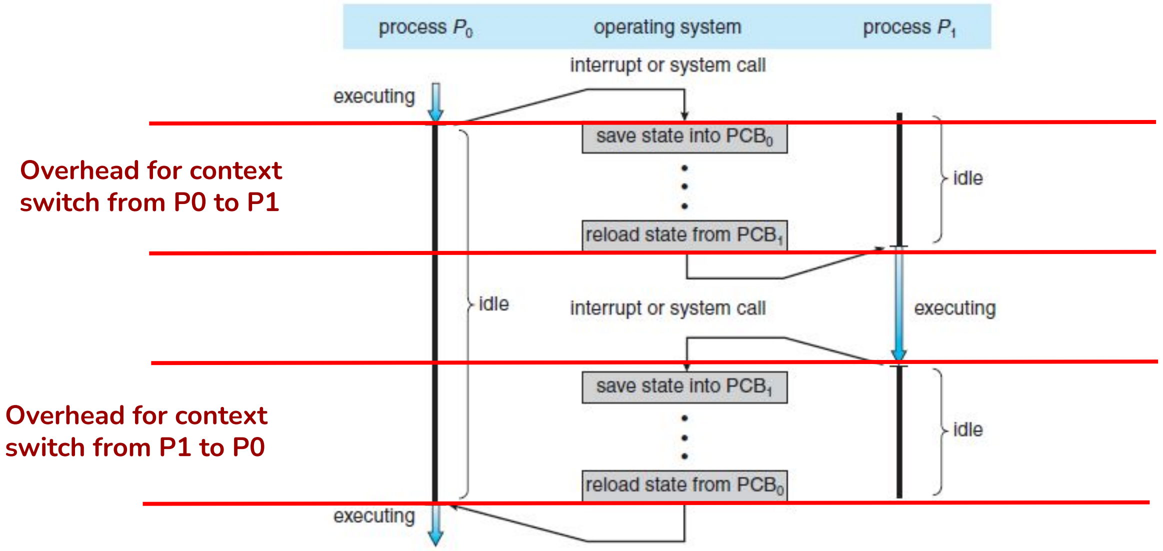


Fig: Addition of swapping in Queueing-Diagram

Context Switch

When an interrupt occurs, the system needs to save the current **context** (state) of the process running on the CPU.

Context Switch: 1. Storing currently executed process context
2. Restoring the next process context to execute



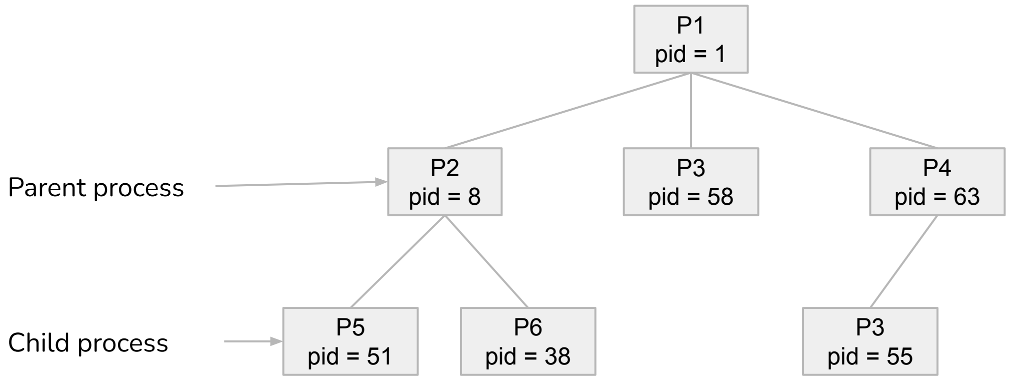
Operating Systems

Operations on Process

FBA

Process Creation

- A process is identified by a unique PID (Process Identifier) in the OS.
- A process may create new processes.



- Child process obtain resources from OS or are restricted to Parent's resources
- Parent process may pass initializing data to child process

Process Creation

- When a process creates new process -

  The parent continues to execute concurrently with its children
  Or,
  The parent waits until some or all of its children have terminated

- Two address-space possibilities for the new process -

  The child process is a duplicate of the parent process
  Or
  The child process has a new program loaded into it.

Process creation in UNIX

System Call: offers the services of the operating system to the user programs.

***fork()*:** create a new process, which becomes the child process of the caller

***exec()*:** runs an executable file , replacing the previous executable

***wait()*:** suspends execution of the current process until one of its children terminates.

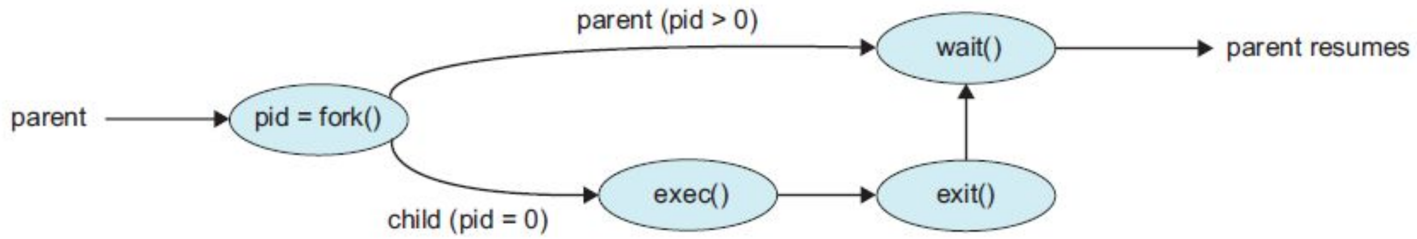


Fig: Process creation using fork() system call

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

int main()
{
pid_t pid;

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

   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     return 1;
   }
   else if (pid == 0) { /* child process */
     execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
     /* parent will wait for the child to complete */
     wait(NULL);
     printf("Child Complete");
   }

   return 0;
}
```

Figure 3.9 Creating a separate process using the UNIX `fork()` system call.

```
int main(){
   fork();
   fork();
   printf(“A”);
}
```

```
int main(){
   fork();
   fork();
   fork();
   printf(“A”);
}
```

```
int main(){
   a = fork();
   if(a==0) fork();
   fork();
   printf(“A”);
}
```

```
int main(){
   fork();
    a = fork();
    if(a==0) fork();
   printf(“A”);
}
```

```
int main(){
   int x = 1;
   a = fork();
   if(a==0){
       x = x -1;
       printf(“value of x is: %d”, x);
   }
   else if (a>0){
       wait(NULL);
       x = x +1;
       printf(“value of x is: %d”, x);
   }
}
```

Process Termination

A process is terminated when -

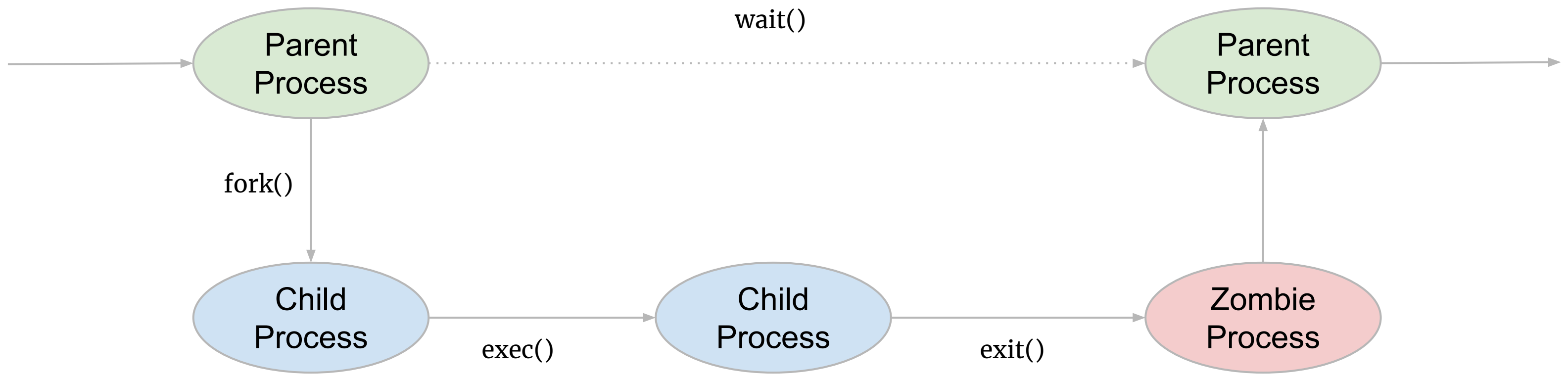
It executes its last statement
Or
Termination cause by another process

When a process is terminated, the resources are deallocated.

A parent may terminate its child if -

1. Child has exceeded the usage of resources
2. Task assigned to child is no longer needed
3. Parent is exiting (cascading termination)

Zombie Process in UNIX



Operating Systems

Interprocess Communication

FBA

Processes in the system

Processes running concurrently may be -

Independent (cannot affect or be affected by other process)

Or

Cooperating (can affect or be affected by other process)

Process cooperation is needed for -

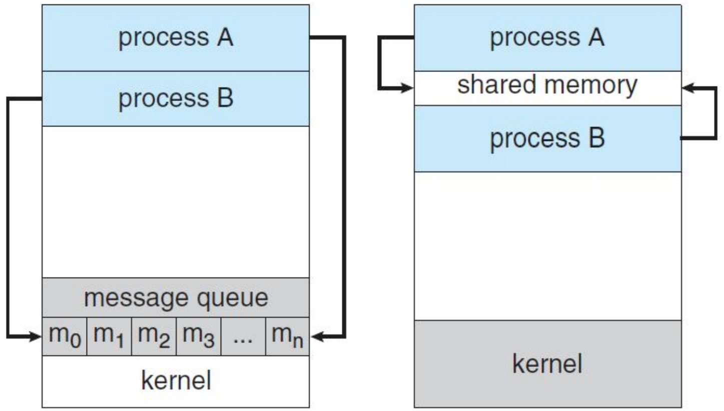
- ➔ Information sharing
- ➔ Computational speedup
- ➔ Modularity
- ➔ Convenience

Inter Process Communication

IPC is a ***mechanism*** to exchange data and information among processes.

Two fundamental model of IPC -

1. Shared Memory
2. Message Passing

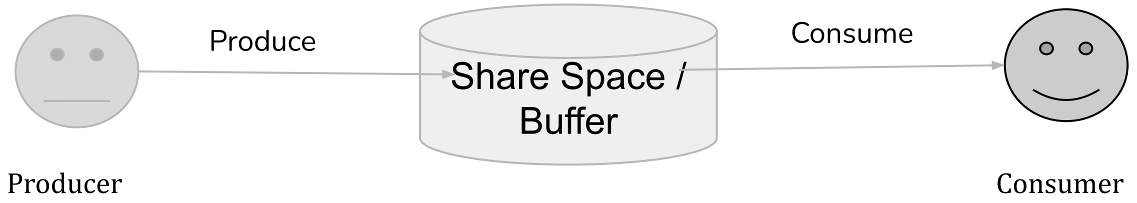


Shared Memory System

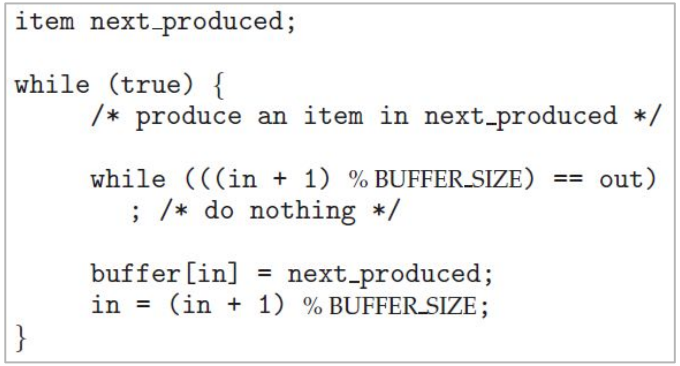
(Producer-Consumer Problem)

Producer: produces products for consumer

Consumer: consumes products provided by producer



Producer-Consumer Problem (Producer)



```
item next_produced;

while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

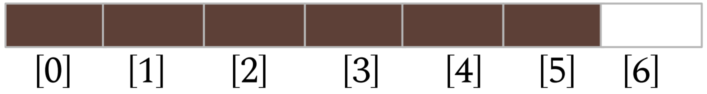
in: next free position in buffer
out: first full position in buffer

Both initialized with 0.

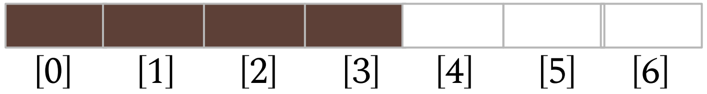
in = 0
out = 0

Here, BUFFER_SIZE = 7

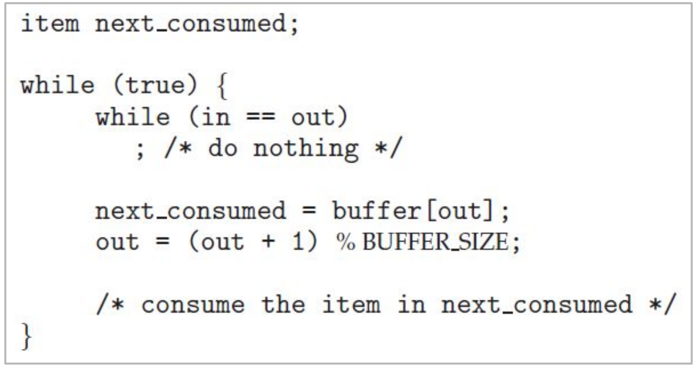
When buffer is full,
in = 6 , out = 0



When buffer is not full,
In = 4, out = 0



Producer-Consumer Problem (Consumer)



```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

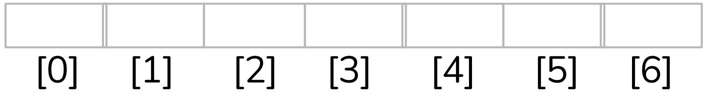
in: next free position in buffer
out: first full position in buffer

Both initialized with 0.

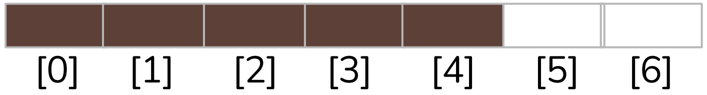
in = 0
out = 0

Here, BUFFER_SIZE = 7

When buffer is empty,
in = 0 , out = 0



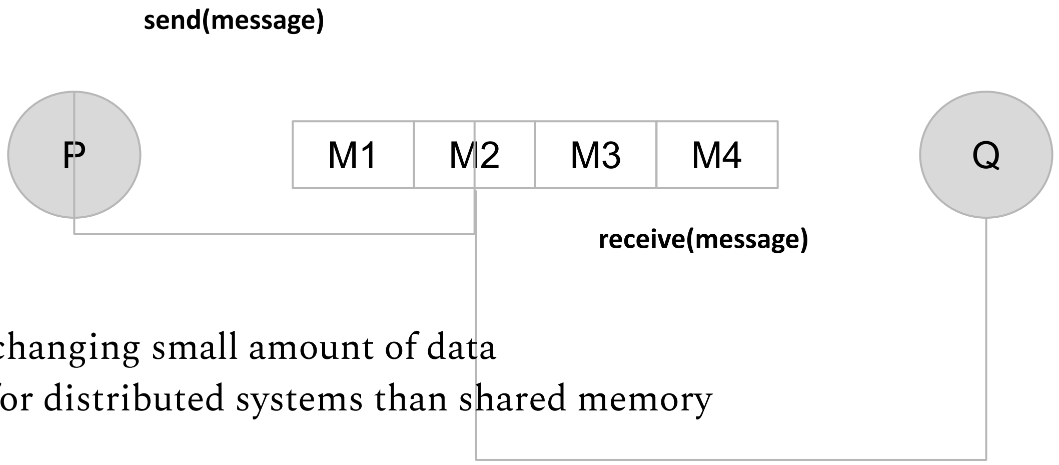
When buffer is not empty,
In = 5, out = 0



Message Passing System

If processes P and Q want to communicate, they must *send* messages to and *receive* messages from each other.

A communication link must exist between P and Q.



- Useful for exchanging small amount of data
- More suited for distributed systems than shared memory

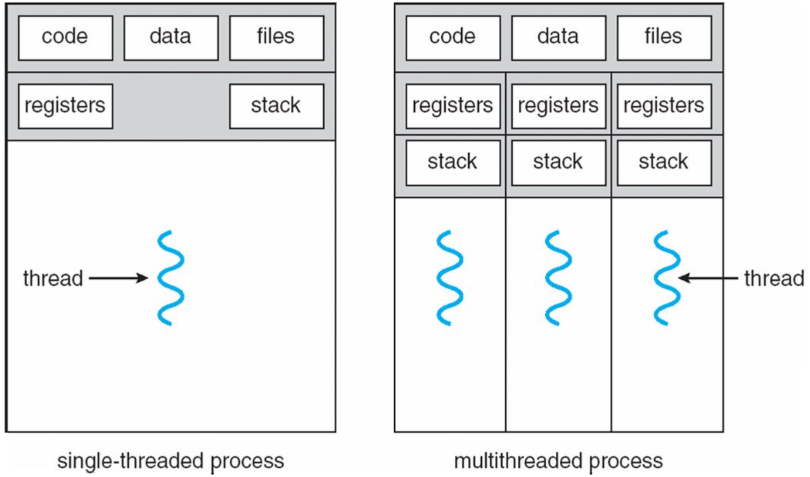
OPERATING SYSTEMS

Threads

FBA

Thread

- ➢ A thread is a path of execution within a process.



- A thread contains -
  - Thread ID
  - Program Counter
  - Register Set
  - Stack
- Shares with other threads belonging to the same process -
  - Code Section
  - Data Section
  - OS resources

- ➢ A traditional process has a single thread of control (Single Threaded Process)
- ➢ Process with multiple threads of control, can perform more than one task at a time (Multi Threaded Process)

Benefits

There are four major categories of benefits to multi-threading:

1. **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.

1. **Resource sharing** - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.

1. **Economy** - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.

1. **Scalability, i.e. Utilization of multiprocessor architectures** - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. (Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)

Multicore Programming

Multicore or **multiprocessor** systems putting pressure on programmers, challenges include:

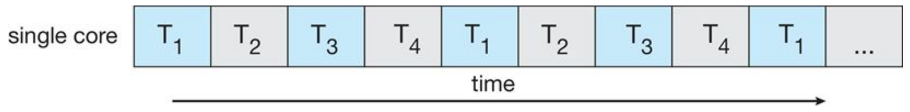
- ➔ Dividing activities
- ➔ Balance
- ➔ Data splitting
- ➔ Data dependency
- ➔ Testing and debugging

Parallelism implies a system can perform more than one task simultaneously

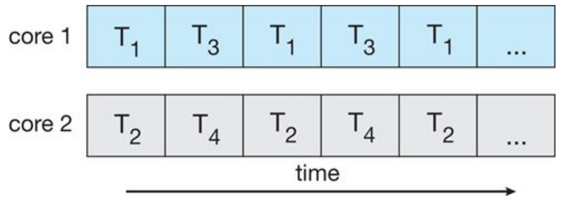
Concurrency supports more than one task making progress

- Single processor / core, scheduler providing concurrency

Concurrent execution on single-core system:



Parallelism on a multi-core system:

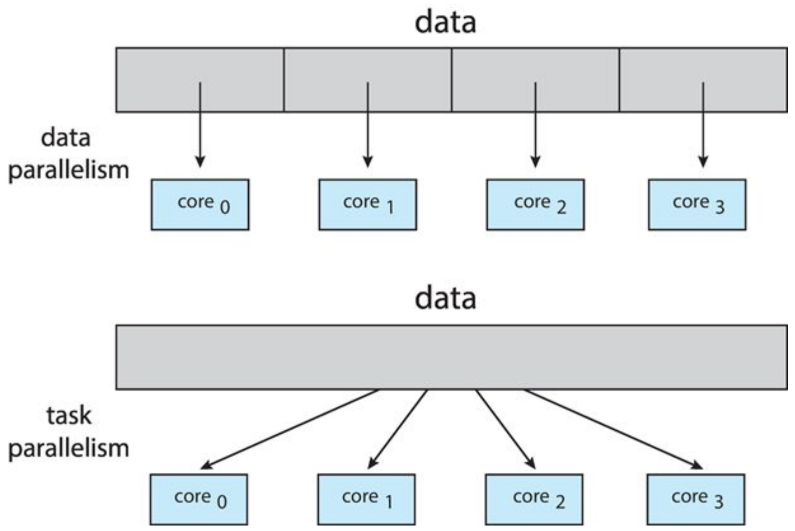


core 1
T_1 T_3 T_1 T_3 T_1 ...
core 2
T_2 T_4 T_2 T_4 T_2 ...
time

Multicore Programming

Data parallelism – distributes subsets of the same data across multiple cores, same operation on each

Task parallelism – distributes threads across cores, each thread performing unique operation



Amdahl's Law

Identifies performance gains from adding additional cores to an application that has both serial and parallel components

- ➔ S is serial portion
- ➔ N is number of processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

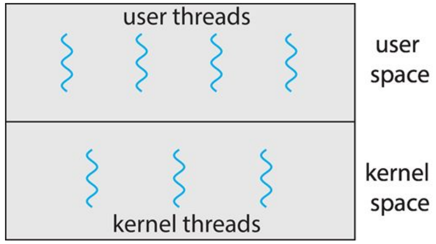
OPERATING SYSTEMS

Multithreading Models

FBA

User and Kernel Threads

- There are two types of threads to be managed in a modern system: User threads and kernel threads.
- In a specific implementation, the user threads must be mapped to kernel threads.



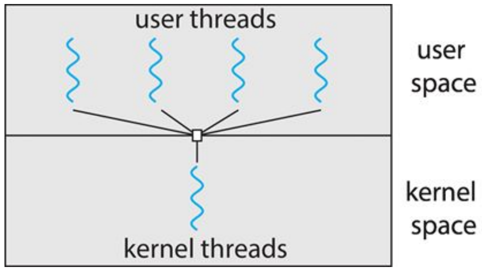
- **User threads** are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.

- **Kernel threads** are supported within the kernel of the OS itself. All modern OS support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

Multithreading Models

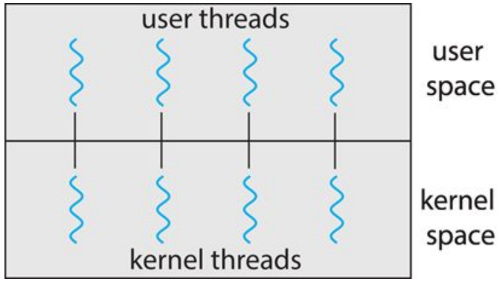
Many-to-One:

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



One to One:

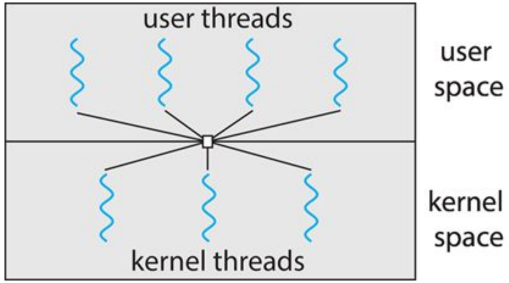
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



Multithreading Models

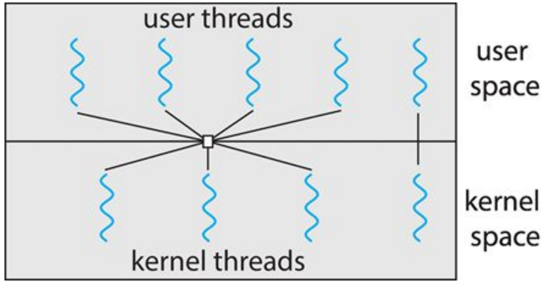
Many-to-Many:

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the ThreadFiber package
- Otherwise not very common



Two-level Model:

- Similar to M:M, except that it allows a user thread to be bound to kernel thread



OPERATING SYSTEMS

Thread Libraries

FBA

Thread library

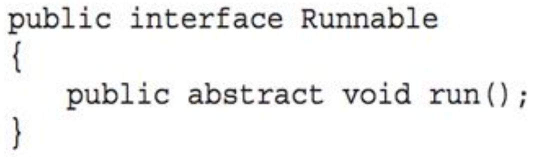
- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls, and requires a kernel with thread library support.
- There are three main thread libraries in use today:
  - POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
  - Win32 threads - provided as a kernel-level library on Windows systems.
  - Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

Pthreads

- The POSIX standard (IEEE 1003.1c) defines the specification for pThreads, not the implementation.
- pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
- Global variables are shared amongst all threads.
- One thread can wait for the others to rejoin before continuing.
- pThreads begin execution in a specified function.
- Common in UNIX operating systems (Linux & Mac OS X)

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface



```java
public interface Runnable
{
    public abstract void run();
}
```

➔ Standard practice is to implement Runnable interface

OPERATING SYSTEMS

Threading Issues

FBA

Threading Issues

- **fork() and exec() System Calls:** Duplicate all the threads or not?
- **Thread cancellation:** Thread cancellation is the task of terminating a thread before it has completed.
- **Signal Handling:** Where should a signal be delivered?
- **Thread Pool:** Create a number of threads at the process start-up.
- **Thread Specific data:** Each thread might need it’s own copy of certain data.

OPERATING SYSTEMS

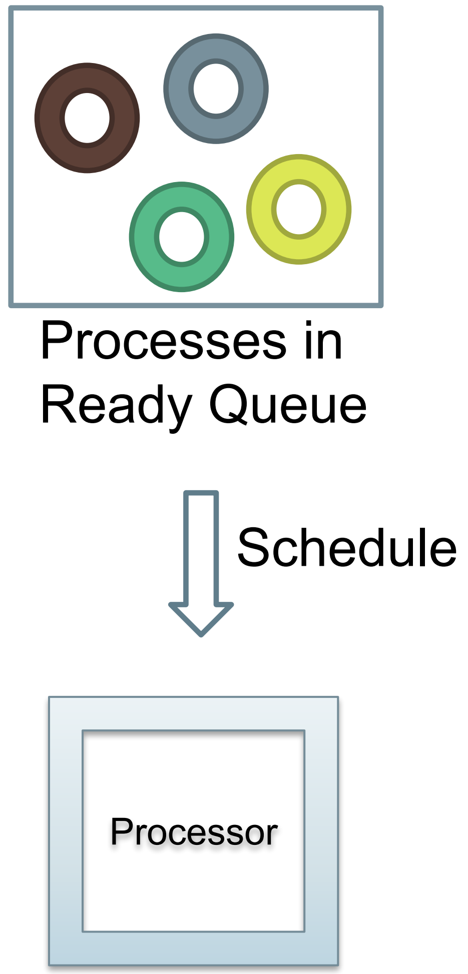
CPU Scheduling

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming

- Continuous Cycle :
  - one process has to wait (I/O)
  - Operating system takes the CPU away
  - Give CPU to another process
  - This pattern continues

- CPU–I/O Burst Cycle : Process execution consists of a *cycle* of CPU execution and I/O wait



CPU Scheduler

- **Selects from among the processes in ready queue, and allocates the CPU to one of them**
  - FIFO queue
  - Priority queue
  - Tree
  - Unordered linked-list
- **CPU scheduling decisions may take place when a process:**
  1. Switches from running to waiting state (I/O request)
  2. Switches from running to ready state (e.g. when interrupt occurs)
  3. Switches from waiting to ready (e.g. at completion of I/O)
  4. Terminates
- Scheduling under 1 and 4 is ***nonpreemptive***
- All other scheduling is ***preemptive***
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time**
  – amount of time to execute a particular process
  -- the interval from the time of submission of a process to the time of the completion.
  -- sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, doing I/O
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

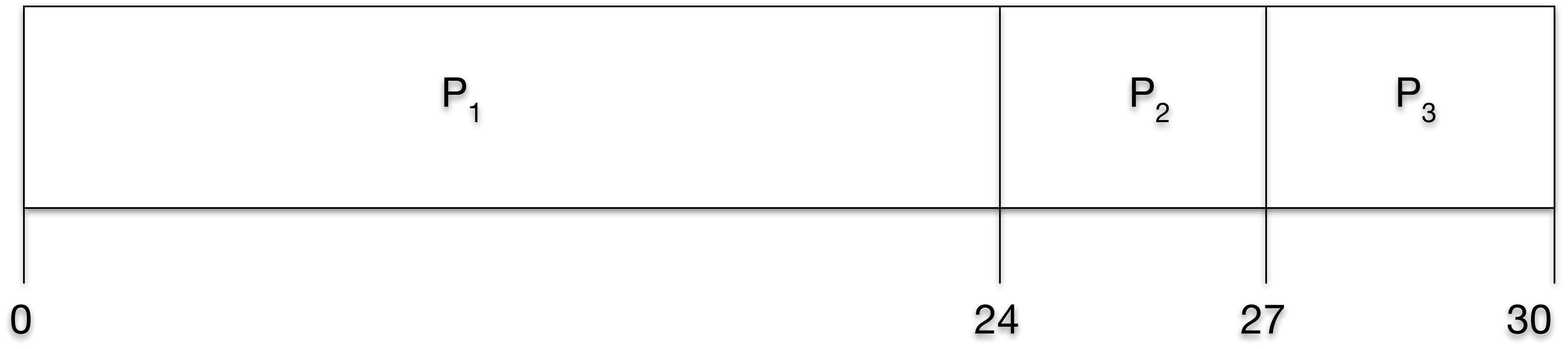
OPERATING SYSTEMS

CPU Scheduling Algorithms - First Come First Serve (FCFS)

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

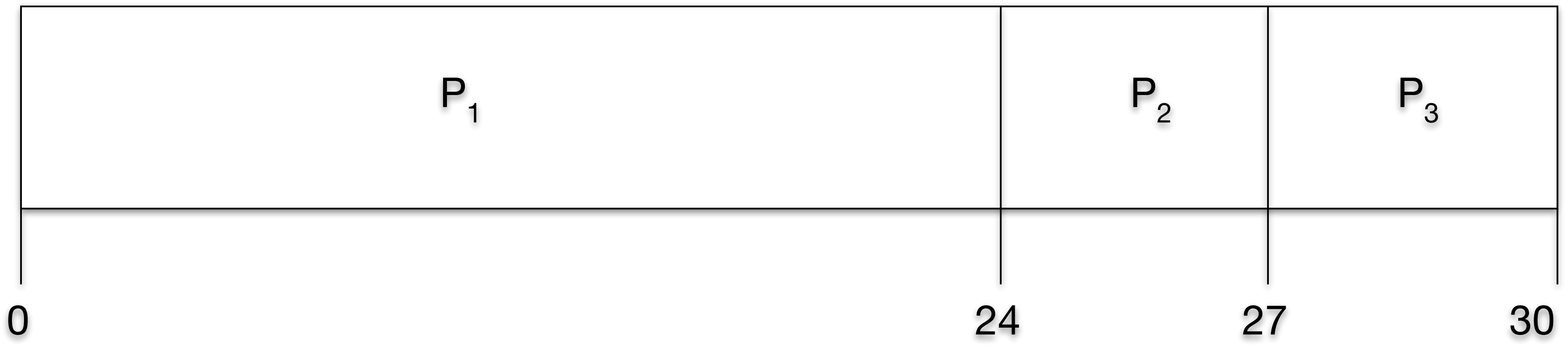
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



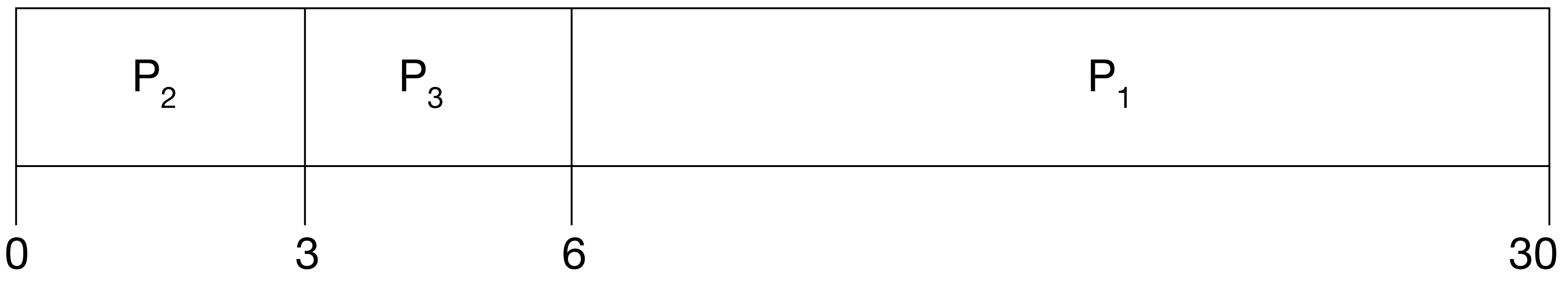
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Turnaround time $P_1 = 24$; $P_2 = 27$; $P_3 = 30$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

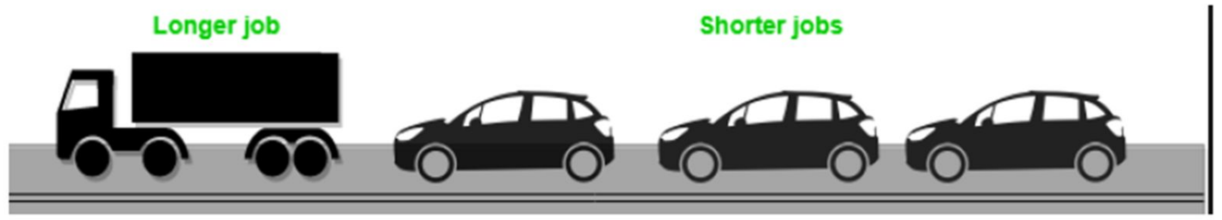
P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
Average waiting time: (6 + 0 + 3)/3 = 3
Much better than previous case

Convoy effect *- short process behind long process.*
Consider one CPU-bound and many I/O-bound processes



OPERATING SYSTEMS

CPU Scheduling Algorithms - Shortest Job First (SJF)

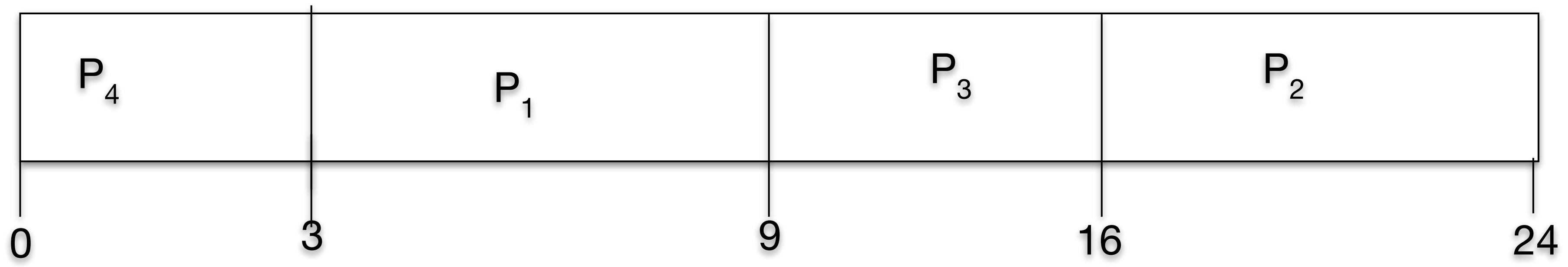
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - **Non-preemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst
  - **preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- <u>SJF is optimal </u>– gives minimum average waiting time for a given set of processes

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

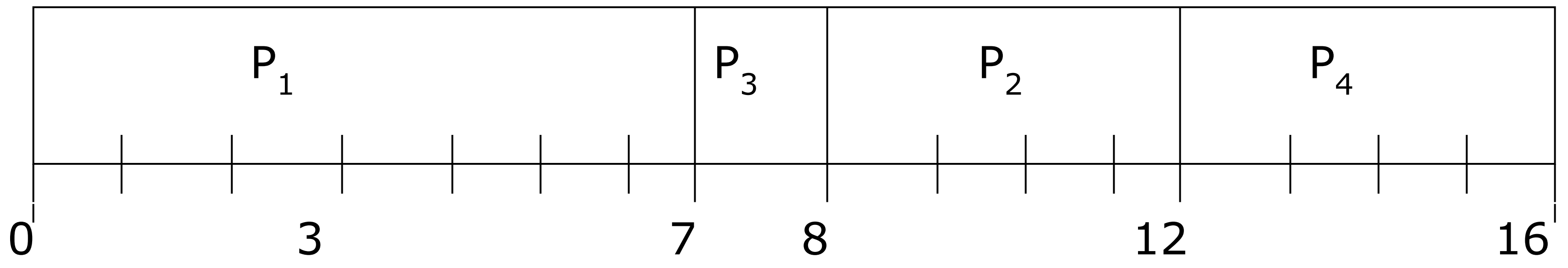
- SJF scheduling chart



Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

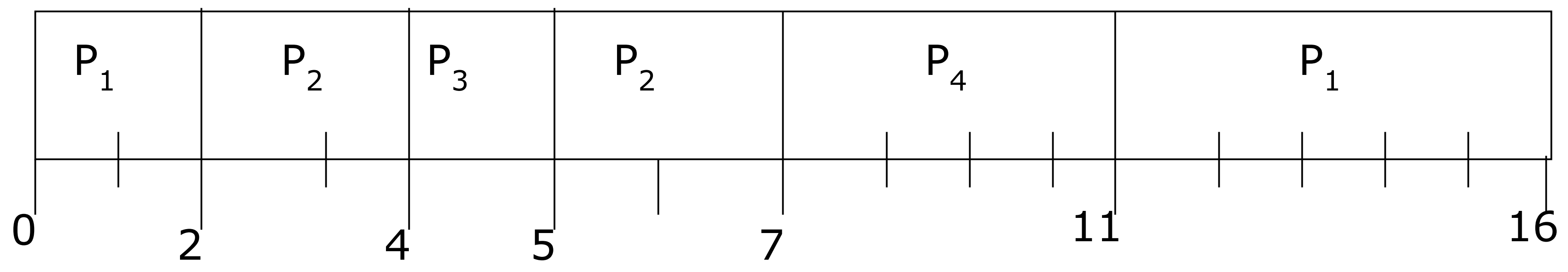
- SJF (non-preemptive)



Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)

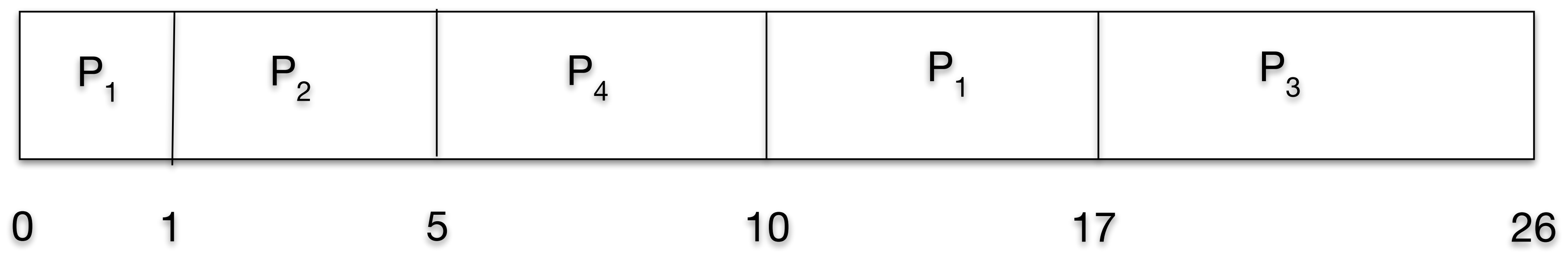


Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- *Preemptive* SJF Gantt Chart



OPERATING SYSTEMS

CPU Scheduling Algorithms - Priority Scheduling

Priority Scheduling

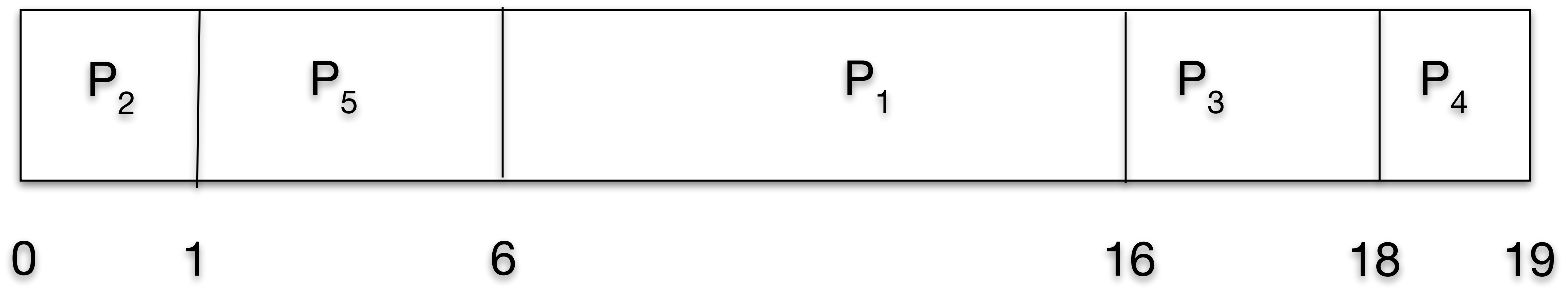
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Priority can be defined either internally or externally.
  - Factors for internal priority assignment:
    - Time limit, memory requirements, the number or open files etc.
  - Factors for external priority assignment:
    - Importance of the process, the type and amount of funds being paid for computer use, department sponsoring works etc.

Example of Priority Scheduling

Non Preemptive:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Problem ≡ **Starvation** – low priority processes may never execute

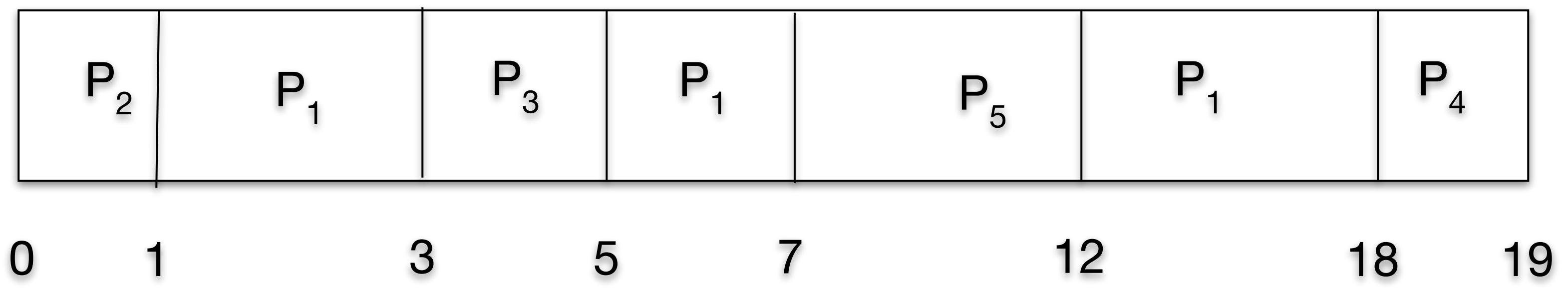
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Preemptive:

Process	Arrival Time	Burst Time	Priority
P_1	0	10	4
P_2	0	1	1
P_3	3	2	3
P_4	5	1	5
P_5	7	5	2

- Priority scheduling Gantt Chart



- Problem ≡ **Starvation** – low priority processes may never execute

- Solution ≡ **Aging** – as time progresses increase the priority of the process

OPERATING SYSTEMS

CPU Scheduling Algorithms - Round Robin (RR)

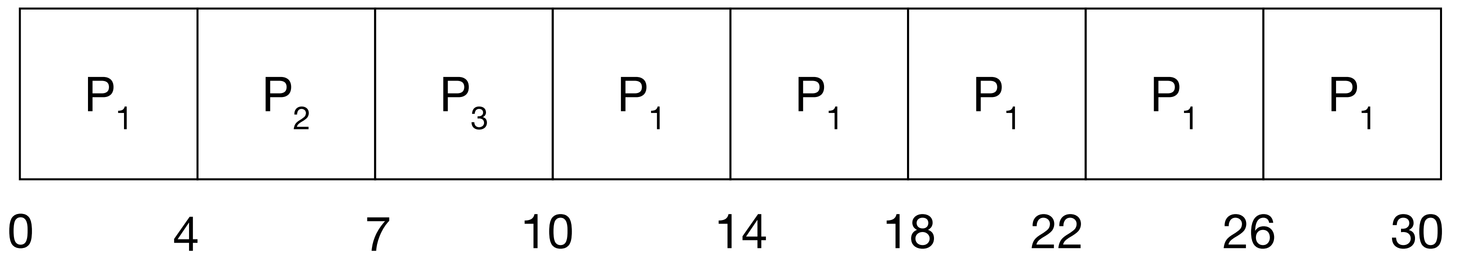
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
    - *q* large $\Rightarrow$ FIFO
    - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

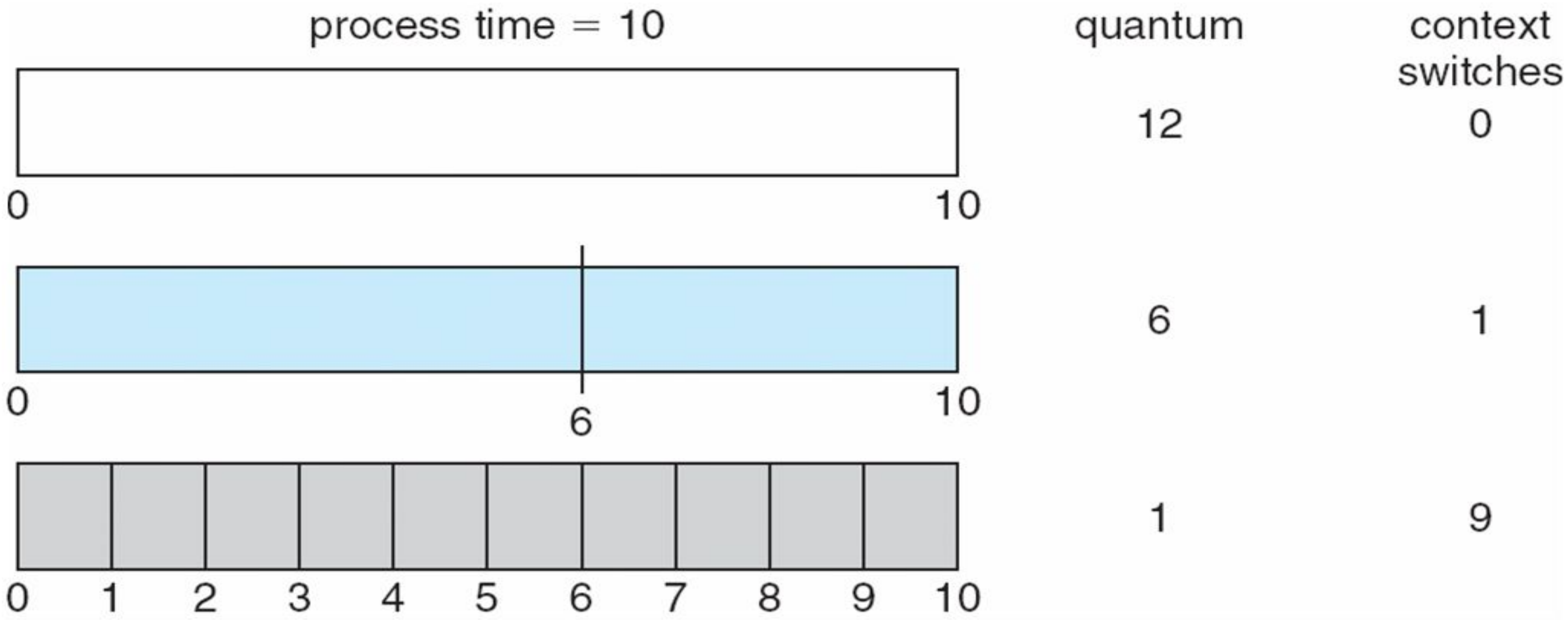
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

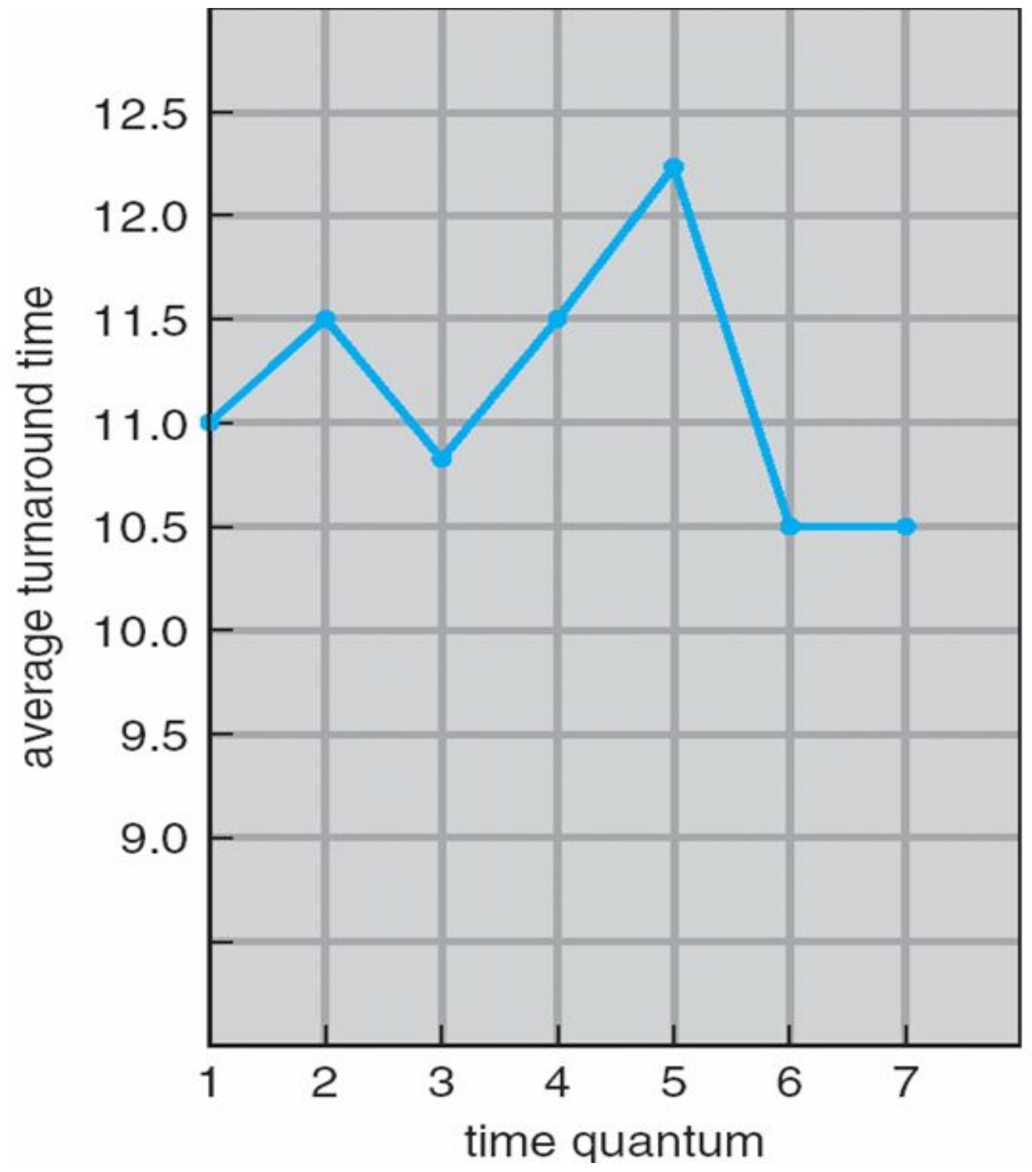


- Average waiting time is 17 / 3 = 5.66 milisecond
- Typically, higher average turnaround than SJF, but better *response*
- quantum should be large compared to context switch time
- Quantum usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than quantum

OPERATING SYSTEMS

CPU Scheduling - Multilevel Queue, Multilevel Feedback Queue

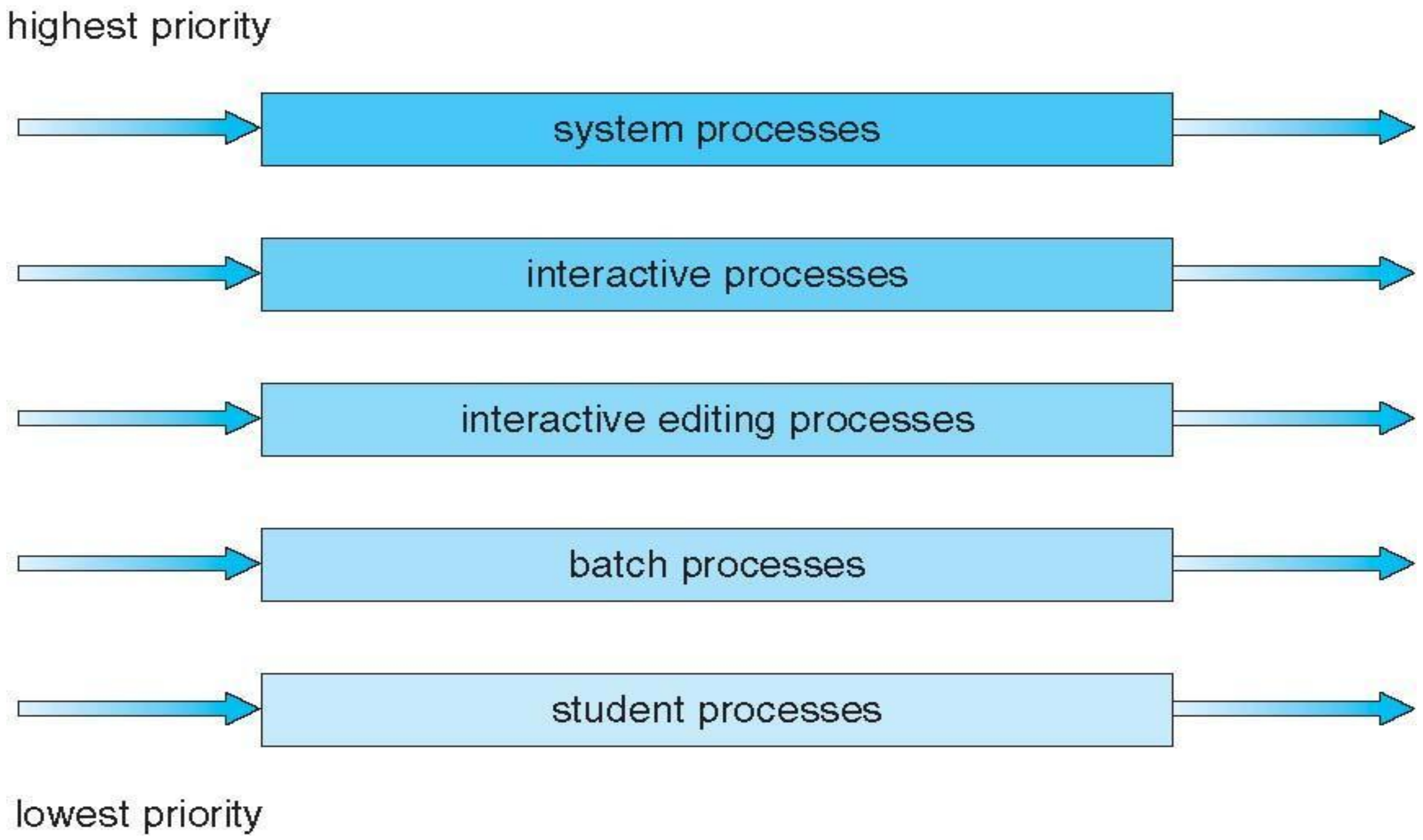
Multilevel Queue

- Another class of scheduling algorithm needs- in which processes are classified into different groups, e.g.:
    - foreground (interactive) processes
    - background (batch) processes
- They have different response time requirements-so different scheduling needs.
- Foreground processes may have priority over background processes.
- A multilevel queue-scheduling algorithm partitions the ready queue into several separate queues-we can see it in the figure of next slide:-

- Each queue has its own scheduling algorithm:
    - Foreground queue scheduled by – RR algorithm
    - Background queue scheduled by – FCFS algorithm

- Scheduling must be done between the queues:
    - Fixed priority preemptive scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
    - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., foreground queue can be given 80% of the CPU time for RR-scheduling among its processes, while 20% to background in FCFS manner.

Multilevel Queue Scheduling



Multilevel Feedback Queue scheduling

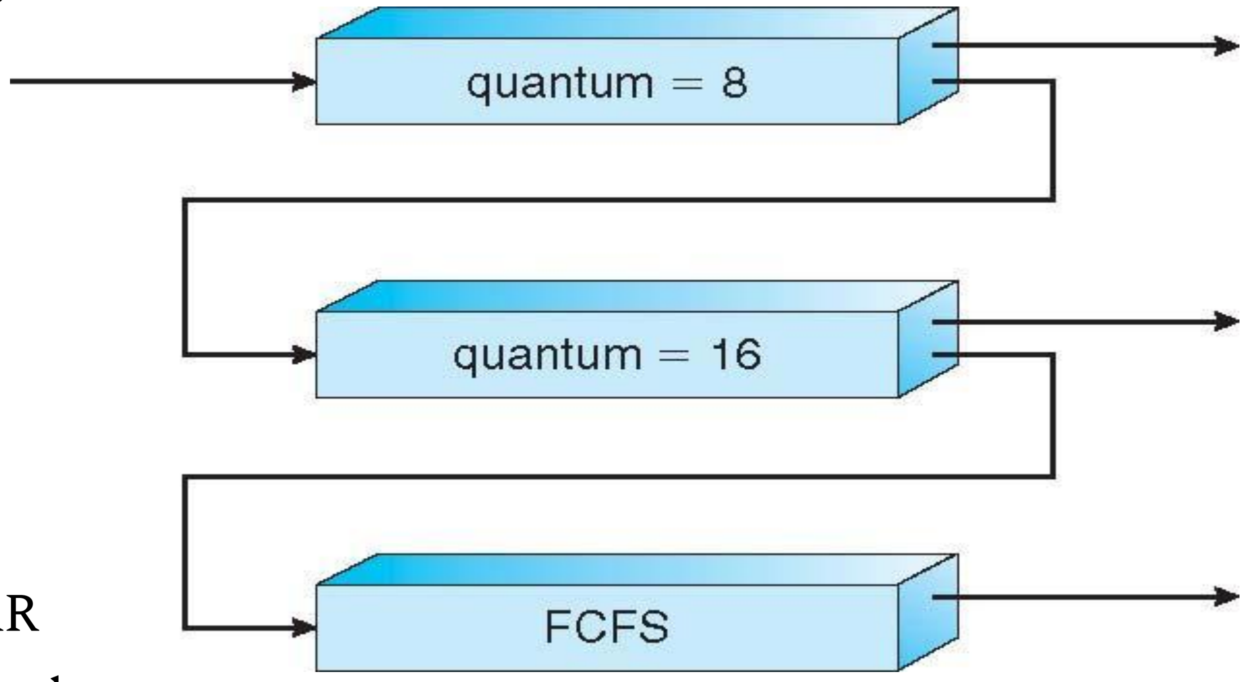
- Multilevel Feedback Queue scheduling, allows a process to move between queues.
- If a process uses too much CPU time, it will be moved to a lower priority queue.
- Similarly, a process that waits too long in a lower-priority queue may me moved to a higher-priority queue.

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues: (can see the figure in next slide)
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

- Scheduling
  - A new job enters queue Q_0 which is served for RR
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2


quantum = 8
quantum = 16
FCFS