18CSC205J – Operating Systems

– Dr. C.G.Balaji
Assistant Professor
Dept. of Computer Science and Engineering
SRM Institute of Science and Technology
Ramapuram, Chennai.

Course Learning Rationale (CLR)

- The **purpose** of learning this course is to:
  - Introduce the key role of an Operating System (OS)
  - Insist the **Process Management** functions of an OS
  - Emphasize the importance of **Memory Management** concepts of an OS
  - Realize the significance of **Device Management** part of an OS
  - Comprehend the need of **File Management** functions of an OS
  - Explore the services offered by the OS practically

Course Learning Outcomes (CLO)

- At the end of this course, you will be able to:
  - Identify the **need** of an Operating System (OS)
  - Know the **Process Management** functions of an OS
  - Understand the need of **Memory Management** of an OS
  - Find the significance of **Device Management** role of an OS
  - Recognize the essentials of **File Management** part of an OS
  - Gain an insight of importance of an OS through practical

UNIT I Syllabus

Operating System Objectives and functions – Gaining the role of Operating systems – The evolution of operating system, Major Achievements – Understanding the evolution of Operating systems from early batch processing systems to modern complex systems – Process Concept – Processes, PCB – Understanding the Process concept and Maintenance of PCB by OS – Threads – Overview and its Benefits – Understanding the importance of threads – Process Scheduling : Scheduling Queues, Schedulers, Context switch – Understanding basics of Process Scheduling – Operations on Process – Process creation, Process termination – Understanding the system calls – fork(), wait(), exit() – Inter Process communication : Shared Memory, Message Passing, Pipe() – Understanding the need for IPC – Process synchronization: Background, Critical section Problem – Understanding the race conditions and the need for the Process synchronization

What is an Operating System?

- An integrated set of specialized programs used to manage overall resources and operations of the computer
- Controls and monitors the execution of all other programs that reside in the computer
- application programs
- other system software
- An interface between applications and hardware

Objectives of an Operating System

- To make the computer system convenient to use in an efficient manner
- To hide the details of the hardware resources from the users
- To provide users a convenient interface to use the computer system
- To act as an intermediary between the hardware and its users, making it easier for the users to access and use other resources
- To manage the resources of a computer system

Objectives of an Operating System

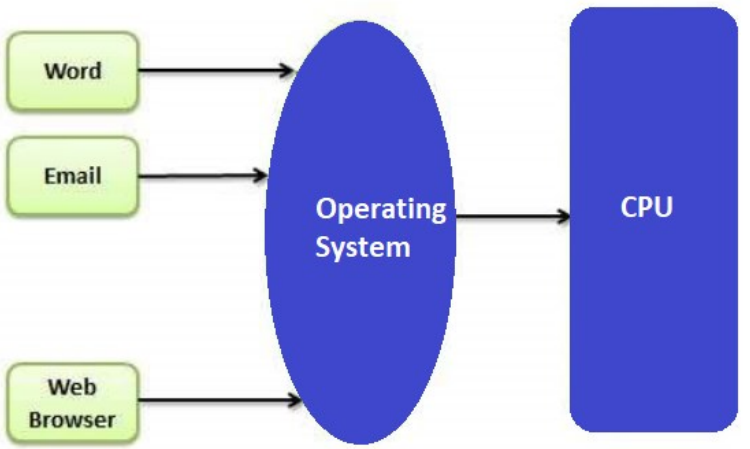
- To keep track of who is using which resource, granting resource requests, and mediating conflicting requests from different programs and users
- To provide efficient and fair sharing of resources among users and programs

Operating System Services

- Program development
- Program execution
- Access I/O devices
- Controlled access to files
- System access
- Error detection and response
- Accounting

Tasks performed by Operating System

- It manages computer hardware
- It controls and coordinates the computer H/W
- It specifies, how various resources like hardware and software can be used in an efficient manner.



Two view points of Operating System

- **User's View**
  - Users want ease of use, good performance and high convenience
  - Don't care about resource utilization
- **System's View**
  - The computer system, need to consider CPU time, memory, I/O Devices.
  - OS is a resource allocator
  - OS is a control program

The Role of an Operating System

- A set of programs that
  - manage computer hardware resources
  - provide common services for application software
- Acts as an interface between the hardware and the programs requesting I/O
- The most fundamental of all system software programs

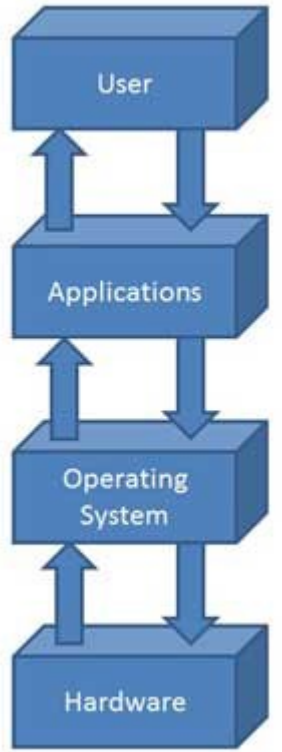


Figure: Computer System stack

Gaining the Role of an Operating System

- **The OS as an User/Computer Interface**
- The OS provides a user interface (UI), an environment for the user to interact with the machine.
- The UI is either graphical or text-based.

Gaining the Role of an Operating System

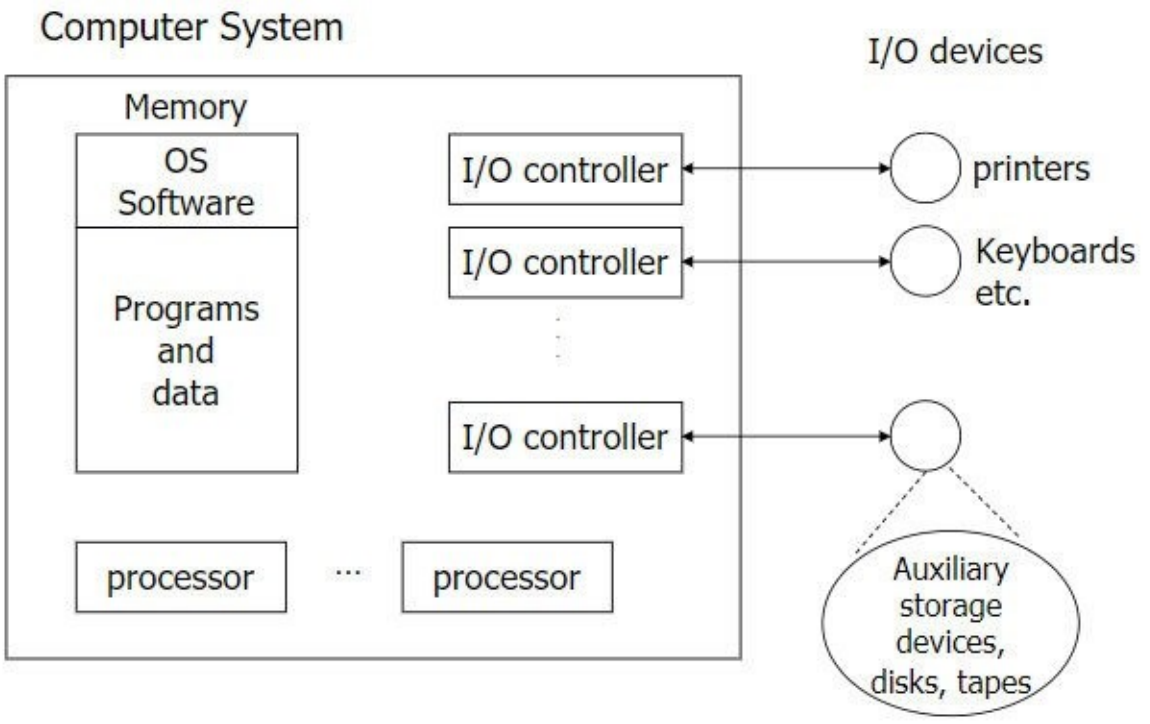
- **Graphical user interface (GUI)**
  - The OS on most computers and smartphones provides an environment with tiles, icons and/or menus.
  - The user interacts with images through a mouse, keyboard or touchscreen.
- **Command line interface (CLI)**
  - This is a text-only service with feedback from the OS appearing in text.
  - Using a CLI requires knowledge of the commands available on a particular machine.

Gaining the Role of an Operating System

- **The OS as a resource manager**
  - A computer system has many resources (hardware and software)
  - The commonly required resources are input/output devices, memory, file storage space, CPU, etc.
  - The OS acts as a manager of the above resources and allocates them to specific programs and users, whenever necessary to perform a particular task.
  - The resources are processor, memory, files, and I/O devices. In simple terms, an OS is an interface between the computer user and the machine.

Gaining the Role of an Operating System

- **The OS as a resource manager**



Evolution of Operating System

Operating System is divided into four generations

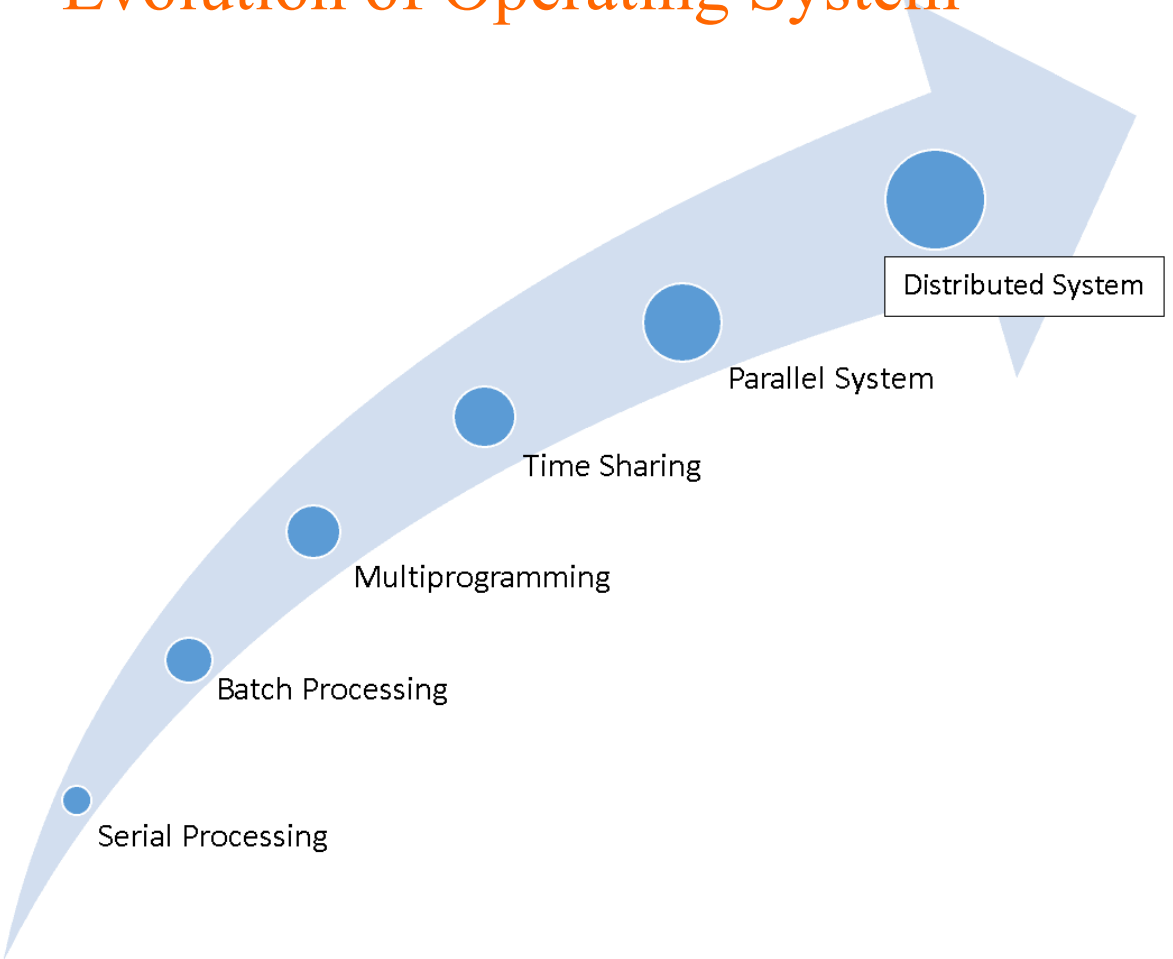
- **First Generation (1945-1955)**
  - In this generation there is no operating system, so the computer system is given instructions which must be done directly.
- **Second Generation (1955-1965)**
  - The Batch processing system was introduced in the second generation.
  - a job or a task that can be done in a series, and then executed sequentially.

Evolution of Operating System

Operating System is divided into four generations

- **Third Generation (1965-1980)**
  - OS was developed to serve multiple users at once in the third generation.
  - Users can communicate through an online terminal to a computer.
- **Fourth Generation (1980-Now)**
  - OS is used for computer networks where users are aware of the existence of computers that are connected to one another.
  - With the onset of new devices like wearable devices, which includes Smart Watches, Smart Glasses, VR gears etc, the demand for unconventional operating systems is rising.

Evolution of Operating System



Understanding the Evolution of Operating System

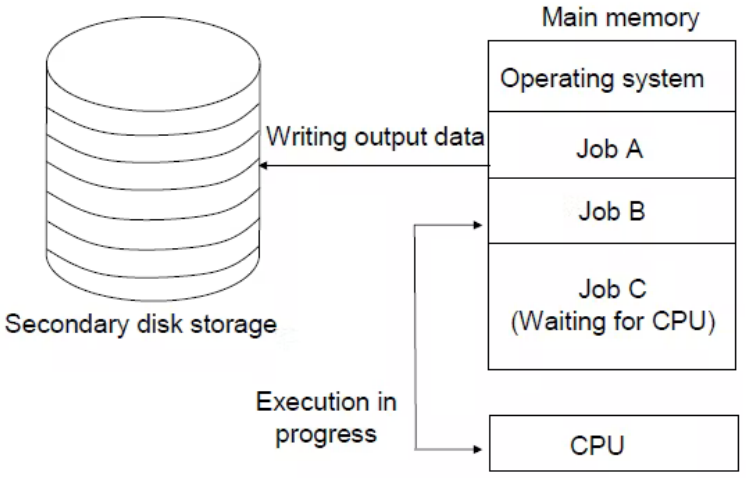
- The **Serial Processing** OS are those which Performs all the instructions into a Sequence Manner.
- Program Counter is used for **executing** all the Instructions.
- **Punch Cards** are used where stiff paper holds digital data represented by the presence or absence of holes in predefined positions.

Understanding the Evolution of Operating System

- In **Batch Processing**, similar Types of jobs are first prepared and stored on the Card.
- The System perform all the operations on the Instructions one by one.
- OS will use the LOAD and RUN Operations.
- Jobs prepared for Execution must be the Same Type

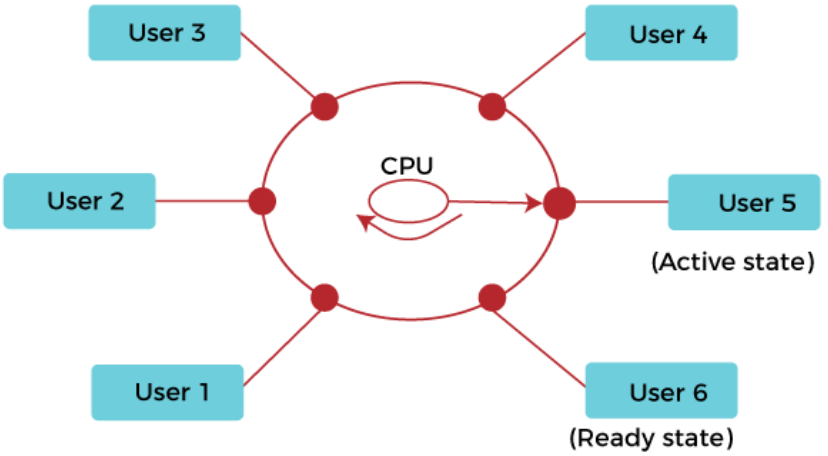
Understanding the Evolution of Operating System

- With the help of **Multi programming** we can execute multiple programs on the System at a time.
- They never use any cards because the Process is entered on the Spot by the user.
- There must be proper management of all the Running Jobs.



Understanding the Evolution of Operating System

- **Time-sharing OS** enables many peo located at various terminals, to use particular computer system at the same tim
- Multiple jobs are implemented by the C by switching between them, but the switc occur so frequently.
- So, the user can receive an immediate response.

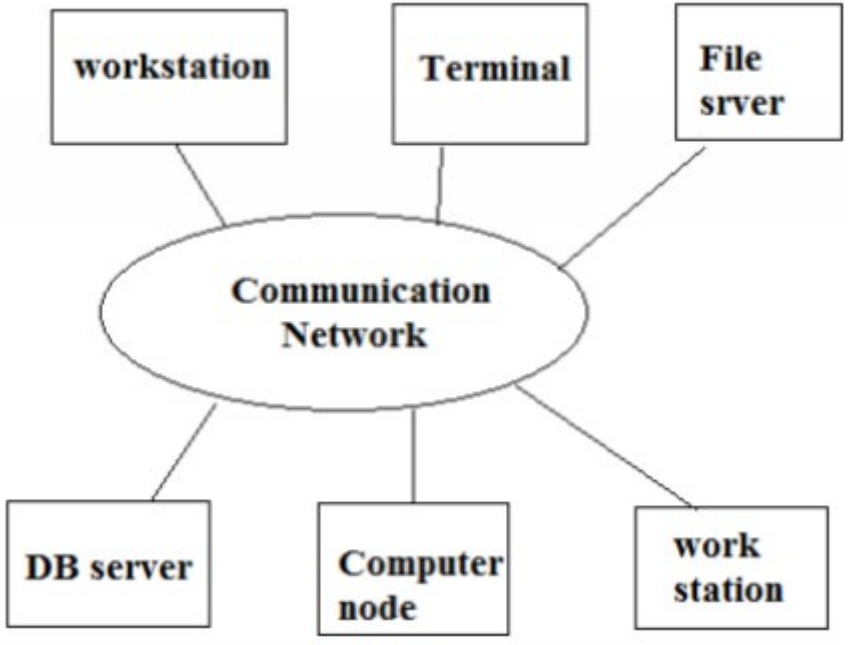


Understanding the Evolution of Operating System

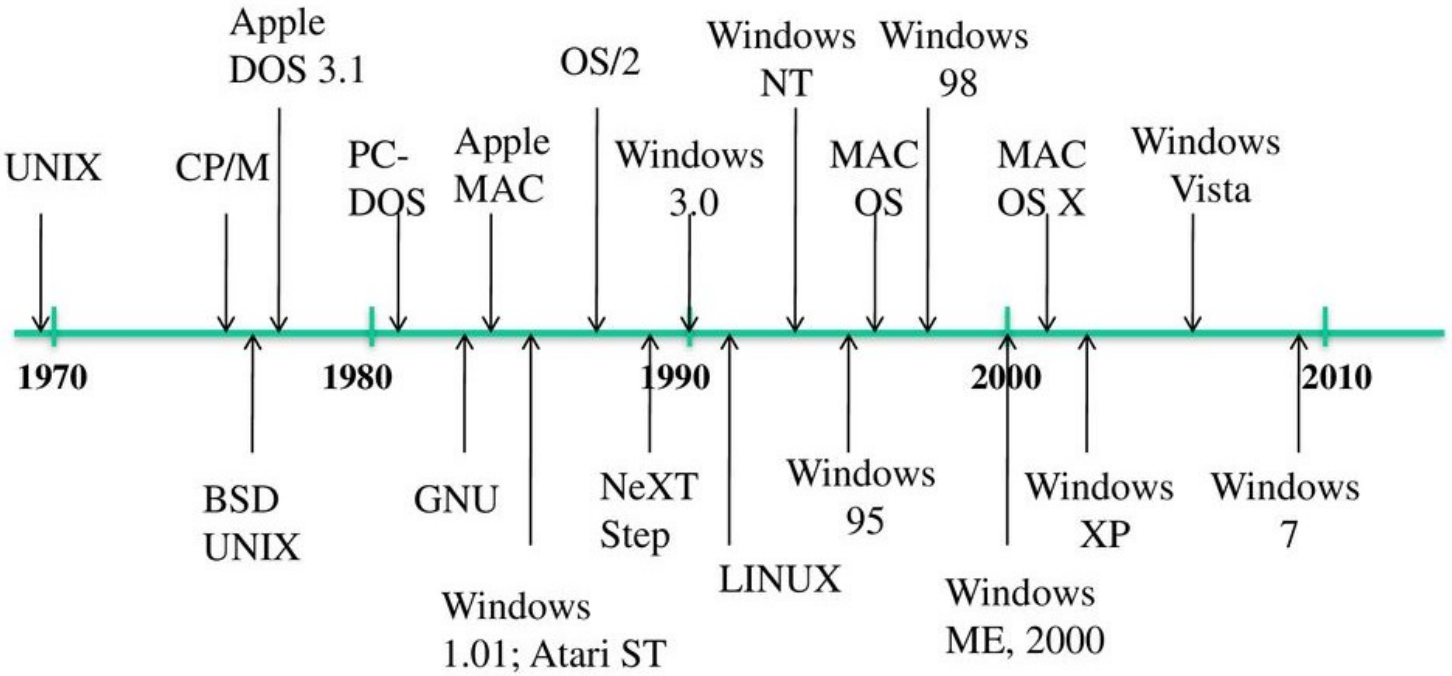
- **Parallel OS** are a type of computer processing platform that breaks large tasks into smaller pieces.
- Executed at the same time in different places and by different mechanisms. They are sometimes also described as “multi-core” processors.
- Parallel OS are used to interface multiple networked computers to complete tasks in parallel.

Understanding the Evolution of Operating System

- A **Distributed OS** is system software over a collection of independent, networked, communicating, and physically separate computational nodes.
- They handle jobs which are serviced by multiple CPUs.
- Each individual node holds a specific software subset of the global aggregate operating system.



Understanding the Evolution of Operating System



Major Achievements of OS

- Operating Systems are among the most complex pieces of software ever developed.
- Four major theoretical advances in the development of operating systems:
  - Process
  - Memory management
  - Information protection and security
  - Scheduling and resource management

Process

- A program in execution **(OR)**
- An instance of a program running on a computer **(OR)**
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by
  - A single sequential thread of execution
  - A current state
  - An associated set of system resources

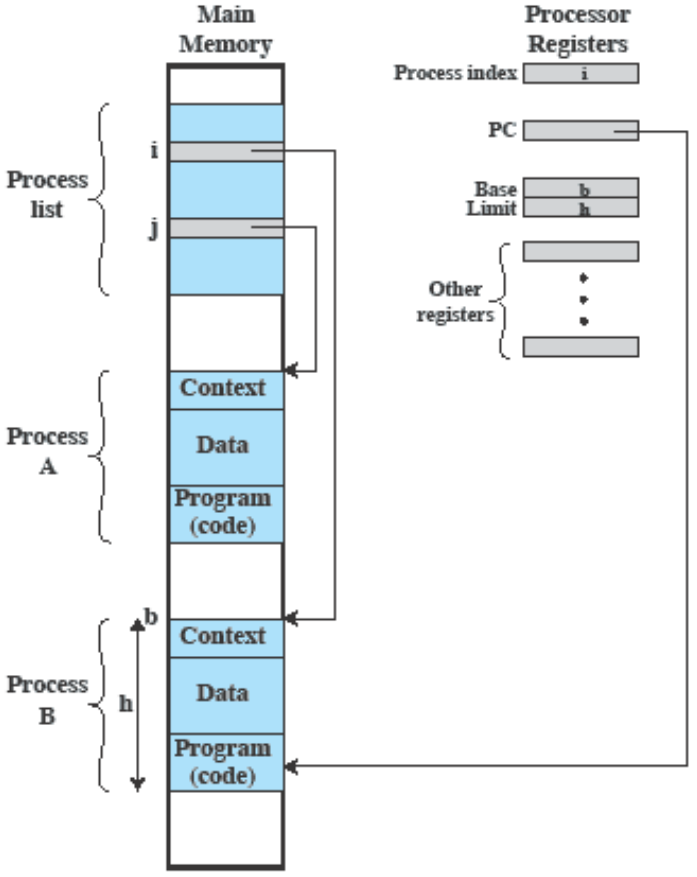
Process

- Three major computer system development created **problems in timing and synchronization** that contributed to the development of concept of the process:
  - Multiprogramming batch operation
  - Time sharing
  - Real time transaction systems

Process

- Difficulties with Designing System Software
  - Improper synchronization
  - Failed mutual exclusion (mutual exclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions.)
  - Non-determinate program operation
  - Deadlocks
- Process consists of three components
  - An executable program
  - Associated data needed by the program
  - Execution context of the program

Process



A Typical Process Implementation

- Two process A and B exists in memory
- Block of memory allocated to each process
  - Program
  - Data
  - Context – program counter, state,..
- OS builds and maintains a process list
  - Pointer to the location of the block of memory
  - Part or all of the execution context of the process

Memory Management

- Process isolation
- Automatic allocation and management
- Protection and access control

Virtual Memory

- Allows programmers to address memory from a logical point of view
- Another layer of indirection
- Allow the illusion of operating with a larger memory space than what is available in reality
- By storing some of the information on the file system

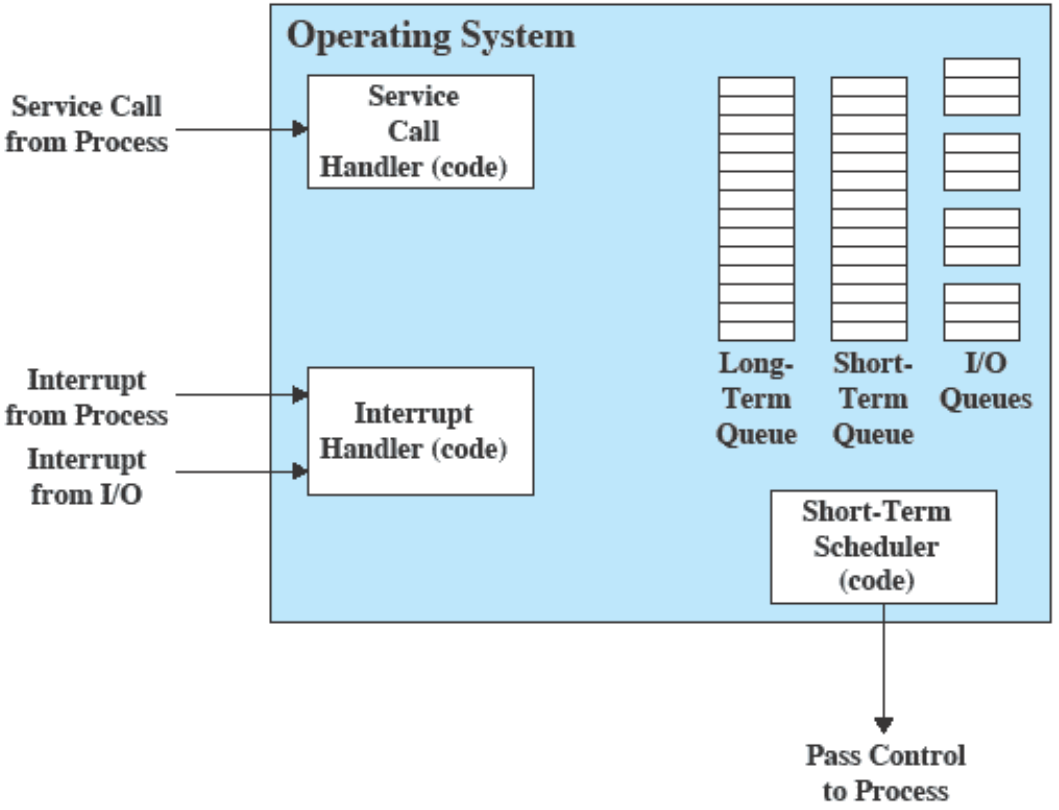
Paging

- One way to implement virtual memory
  - Allows the process to be comprised of a number of fixed-size blocks, called pages
  - Virtual address is a page number and an offset within the page
  - Each page may be located anywhere in main memory
  - Real address or physical address is the main memory address

Scheduling and Resource Management

- Fairness
  - Give equal and fair access to resources
- Differential responsiveness
  - Discriminate among different classes of jobs
- Efficiency
  - Maximize throughput, minimize response time, and accommodate as many uses as possible

Key Elements of an Operating System



Key elements of OS for Multi-programming

Understanding the evolution of Operating systems

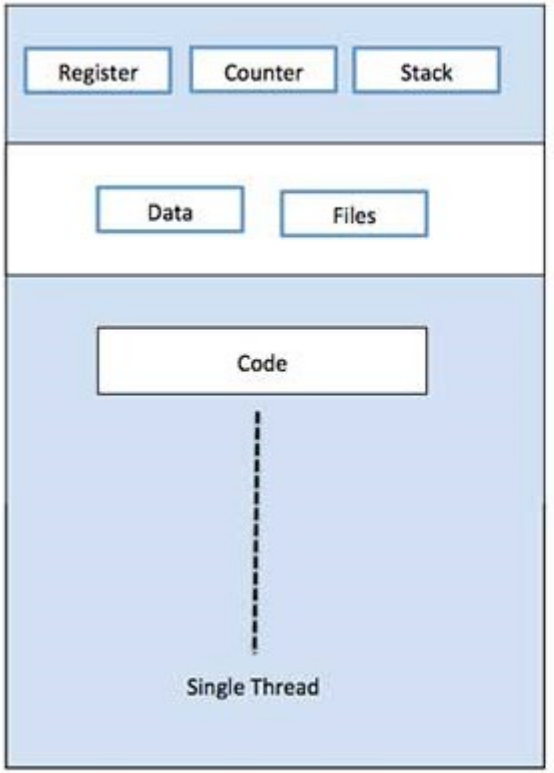
- Microkernel architecture
- Multithreading
- Symmetric multiprocessing (SMP)
- Distributed operating systems

Understanding the evolution of Operating systems

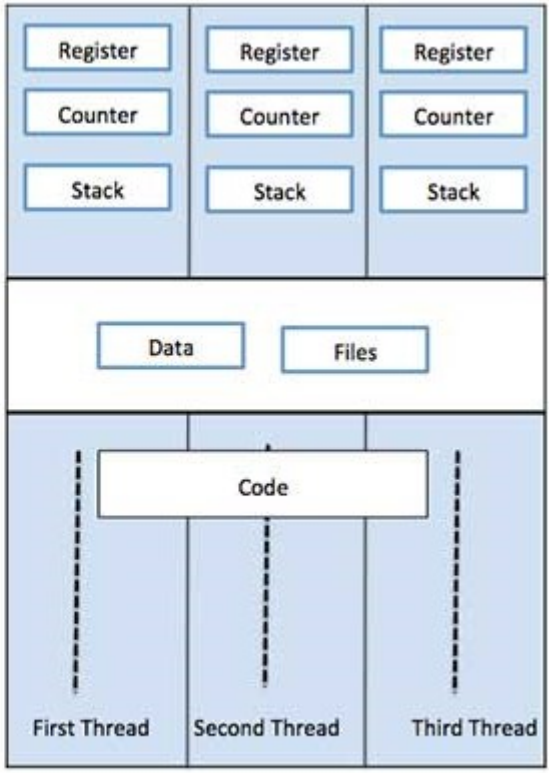
- Microkernel architecture
  - Assigns only a few essential functions to the kernel
    - Address spaces
    - Interprocess communication (IPC)
    - Basic scheduling
  - Was a hot topic in the 1990s
  - Working examples Mach, QNX
  - Current operating systems: Windows, Linux, Mac OS are not microkernel based

Understanding the evolution of Operating systems

- **Multithreading**
  - Process is divided into threads that can run concurrently
  - Thread
    - Dispatchable unit of work
    - Executes sequentially and is interruptible
  - Process is a collection of one or more threads



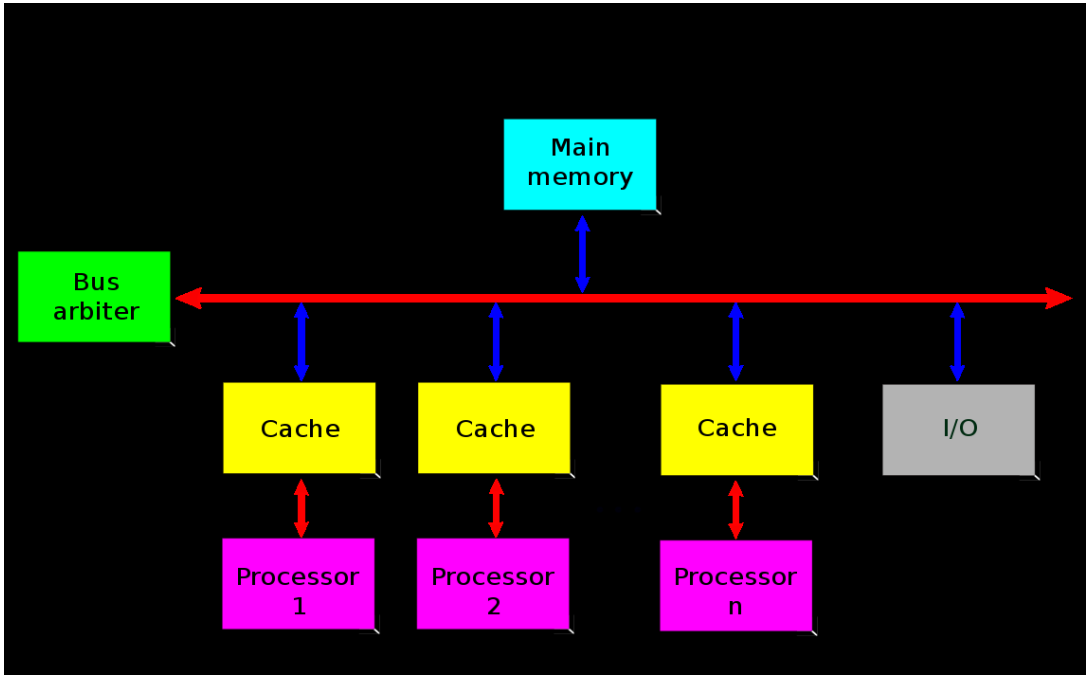
Single Process P with single thread



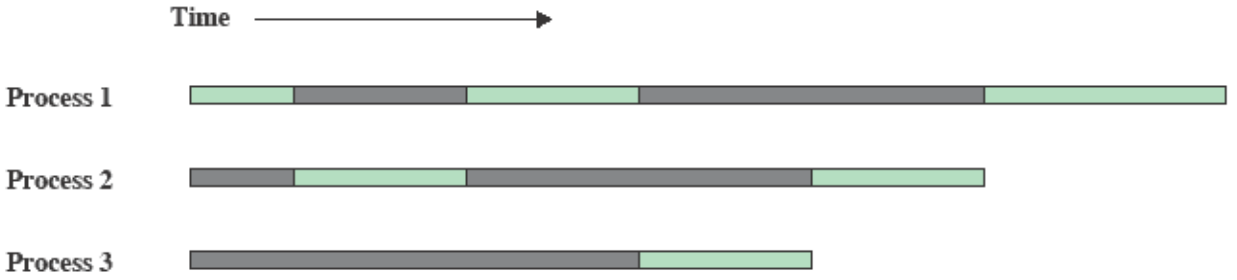
Single Process P with three threads

Understanding the evolution of Operating systems

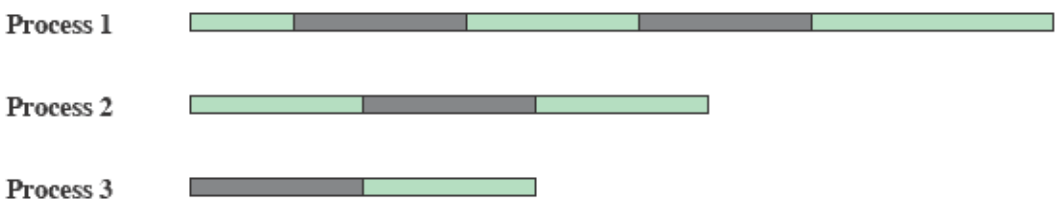
- **Symmetric multiprocessing (SMP)**
  - There are multiple processors
  - These processors share same main memory and I/O facilities
  - All processors can perform the same functions



Understanding the evolution of Operating systems



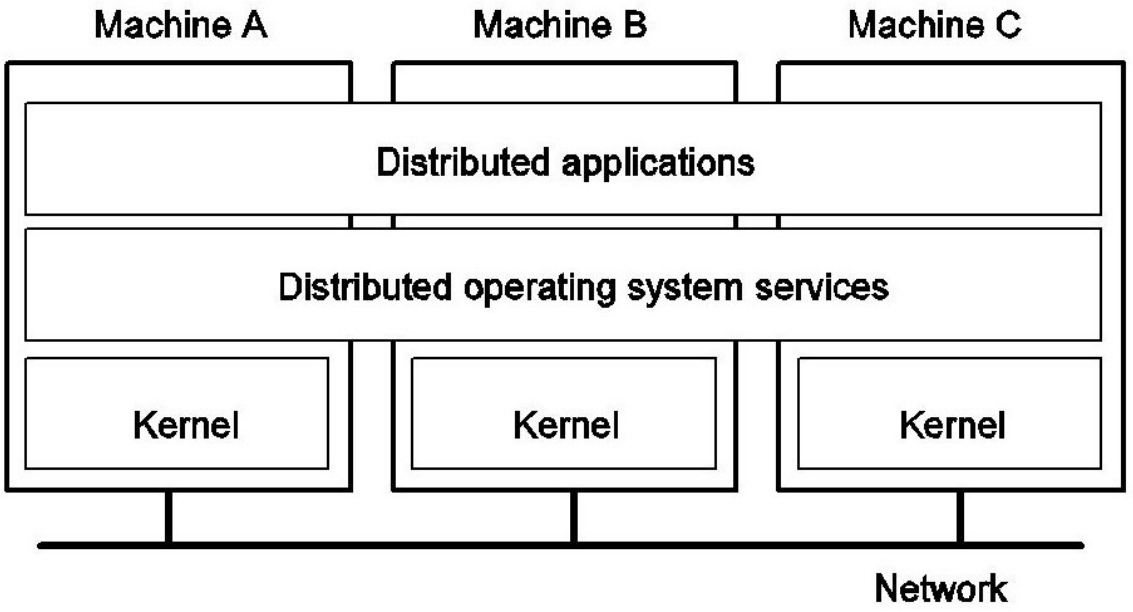
(a) Interleaving (multiprogramming, one processor)



(b) Interleaving and overlapping (multiprocessing; two processors)

Understanding the evolution of Operating systems

- **Distributed operating systems**
  - Provides the illusion of a single main memory space and single secondary memory space
  - Eg. Amoeba by Andrew Tannembaum



OS DESIGN CONSIDERATIONS FOR MULTIPROCESSOR AND MULTICORE

Symmetric Multiprocessor OS Considerations

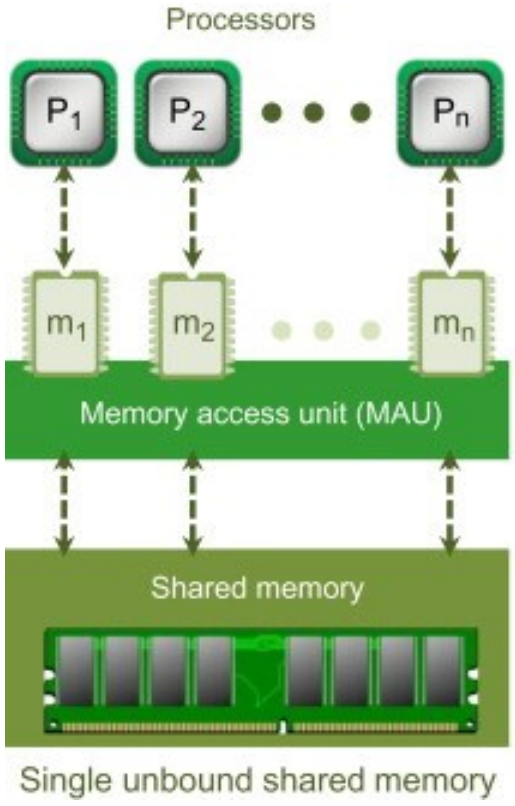
- Kernel can execute on any processor, and typically each processor does self-scheduling from the pool of available processes or threads.
- The kernel can be constructed as multiple processes or multiple threads, allowing portions of the kernel to execute in parallel. The SMP approach complicates the OS.
- The OS designer must deal with the complexity due to sharing resources (like data structures) and coordinating actions (like accessing devices) from multiple parts of the OS executing at the same time.
- Techniques must be employed to resolve and synchronize claims to resources.

OS DESIGN CONSIDERATIONS FOR MULTIPROCESSOR AND MULTICORE

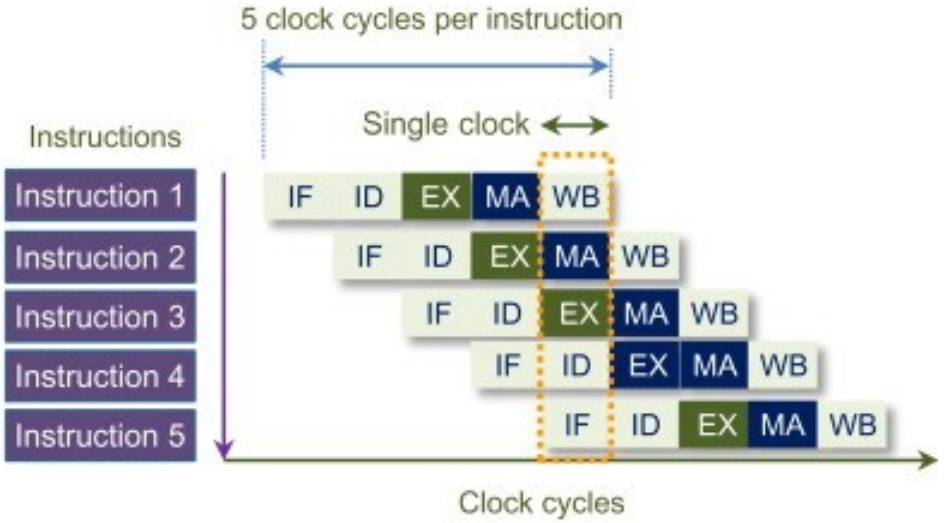
Symmetric Multiprocessor OS Considerations

- An SMP operating system manages processor and other computer resources so that the user may view the system in the same fashion as a multiprogramming uniprocessor system.
- A user may construct applications that use multiple processes or multiple threads within processes without regard to whether a single processor or multiple processors will be available.
- Thus, a multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors.

OS DESIGN CONSIDERATIONS FOR MULTIPROCESSOR AND MULTICORE



Single unbound shared memory



Instruction flow of a pipelined processor

IF: Instruction fetch
ID: Instruction decode
EX: Instruction execution
MA: Memory access to target address
WB: Write execution result to registers

OS DESIGN CONSIDERATIONS FOR MULTIPROCESSOR AND MULTICORE

Symmetric Multiprocessor OS Considerations

- The key design issues include the following:
  - Simultaneous concurrent processes or threads
    - avoid data corruption or invalid operations
  - Scheduling
    - enforcing a scheduling policy
  - Synchronization
    - enforce mutual exclusion and event ordering
  - Memory management
  - Reliability and fault tolerance
- **multiprocessor OS design = multiprogramming uniprocessor design**

OS DESIGN CONSIDERATIONS FOR MULTIPROCESSOR AND MULTICORE

Multicore OS Considerations

- Harness the multicore processing power and intelligently manage the substantial on-chip resources efficiently
- Match the inherent parallelism of a many-core system with the performance requirements of applications
- The potential for parallelism in fact exists at three levels
  - **hardware parallelism** within each core processor, known as instruction level parallelism
  - potential for **multiprogramming** and **multithreaded** execution within each processor
  - potential for a **single application** to execute in **concurrent processes** or threads across multiple cores

OS DESIGN CONSIDERATIONS FOR MULTIPROCESSOR AND MULTICORE

Multicore OS Considerations

•A variety of approaches are being explored for next-generation operating systems:

- **PARALLELISM WITHIN APPLICATIONS**
  - applications can be subdivided into multiple tasks that can execute in parallel
  - these tasks are implemented as multiple processes, perhaps each with multiple threads.
- **VIRTUAL MACHINE APPROACH**
  - allow one or more cores to be dedicated to a particular process
  - leave the processor alone to devote its efforts to that process alone
  - we avoid much of the overhead of task switching and scheduling decisions

PROCESS

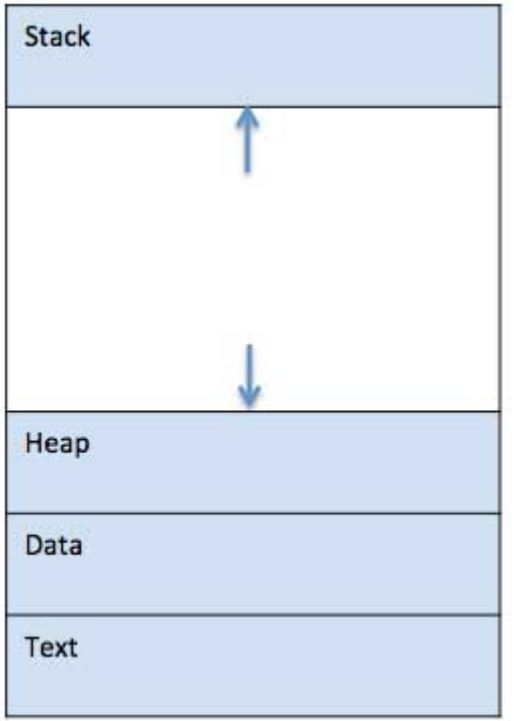
- To introduce the notion of a process – a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems

PROCESS CONCEPT

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

- When a program is loaded into the memory and it becomes a process, it can be divided into four sections
  - stack, heap, text and data.
- **Stack**
  - The process Stack contains the temporary data such as method/function parameters, return address and local variables.
- **Heap**
  - This is dynamically allocated memory to a process during its run time.
- **Text**
  - This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
- **Data**
  - This section contains the global and static variables.



PROCESS CONCEPT

- Difference between a Program and Process:
  - Program is ***passive*** entity stored on disk (**executable file**)
  - Process is ***active*** *entity*
  - 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

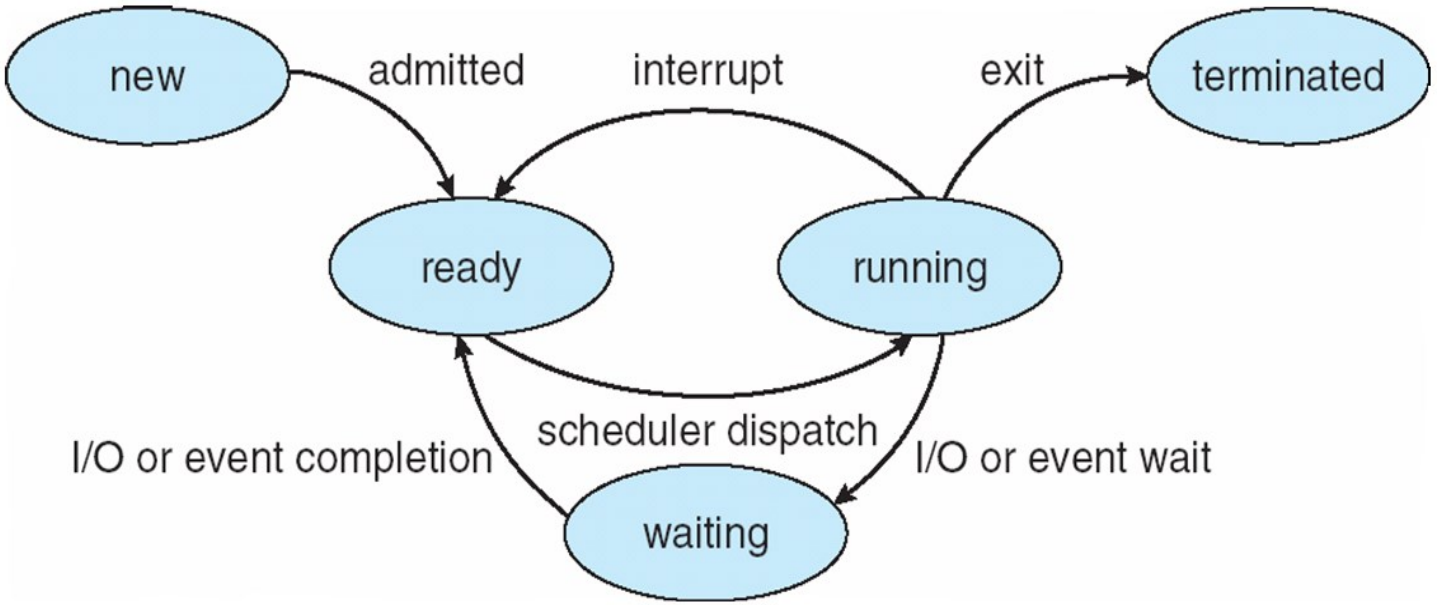
PROCESS CONCEPT

- **Process States**
- When a process executes, it passes through different states.
- These stages may differ in different operating systems, and the names of these states are also not standardized.

S.N.	State & Description
1	**New** This is the initial state when a process is first started/created.
2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	**Running** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	**Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5	**Terminated or Exit** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

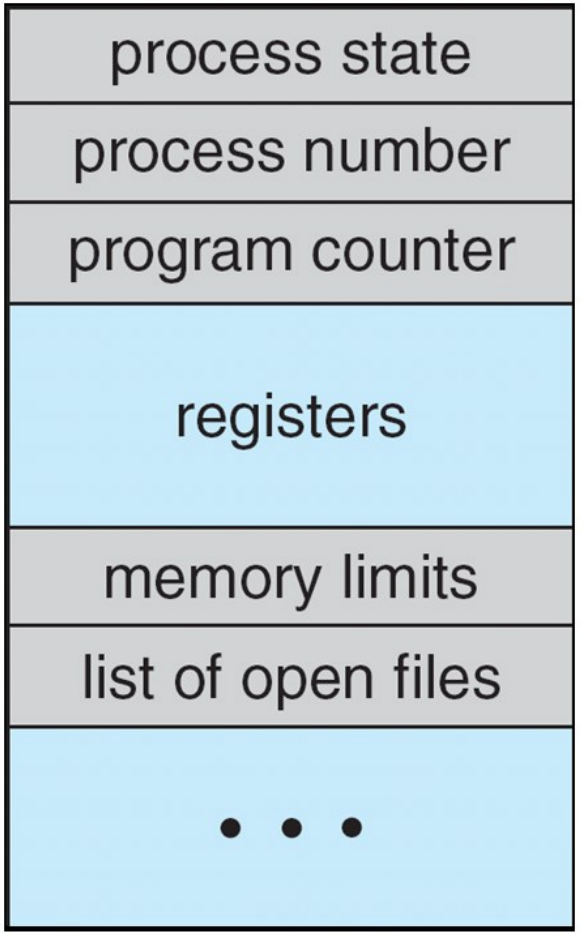
PROCESS CONCEPT

- **Process States**



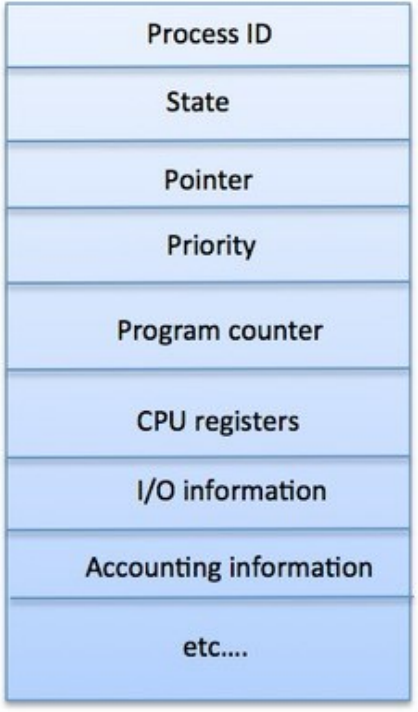
PROCESS CONCEPT

- Process Control Block (PCB) – Information associated with each process (a.k.a. **task control block**)
  - Process state – 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
  - Memory-management information – memory allocated to the process
  - Accounting information – CPU used, clock time elapsed since start, time limits
  - I/O status information – I/O devices allocated to process, list of open files



Process Control Block (PCB)

- The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems.



Process Control Block (PCB)

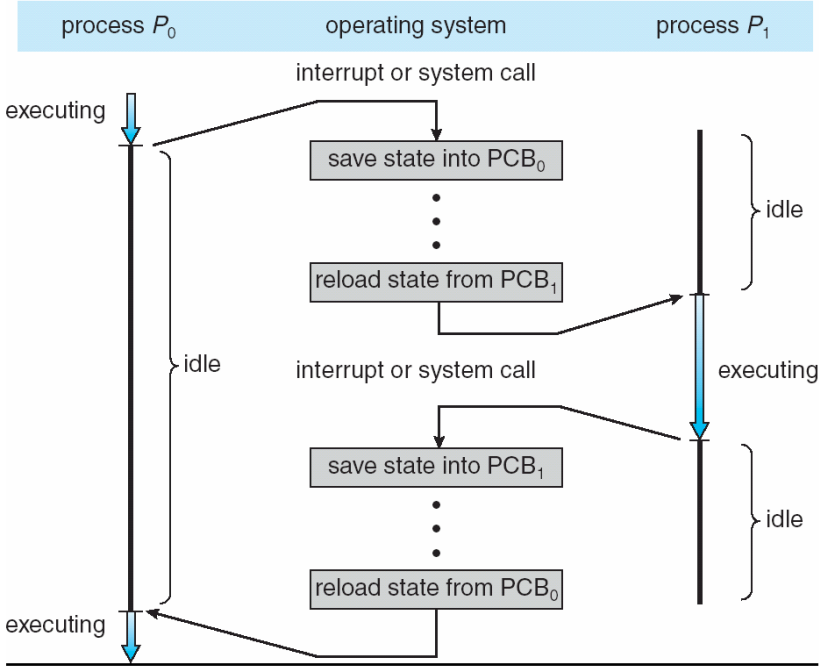
A PCB keeps all the information needed to keep track of a process as listed:

- Process state – The current state of the process - running, waiting, etc
- Process privileges - This is required to allow/disallow access to system resources.
- Process ID - Unique identification for each of the process in the operating system.
- Pointer - A pointer to parent process.
- Program Counter - Program Counter is a pointer to the address of the next instruction to be executed for this process.
- CPU registers - Various CPU registers where process need to be stored for execution for running state.
- CPU Scheduling Information - Process priority and other scheduling information which is required to schedule the process.
- Memory management information - This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
- Accounting information - This includes the amount of CPU used for process execution, time limits, execution ID etc.
- I/O status information - This includes a list of I/O devices allocated to the process.

CPU Switch from Process to Process

What is Context Switching?

- Involves switching of the CPU from one process or task to another.
- The execution of the process that is present in the **running state is suspended** by the kernel and another process that is present in **the ready state is executed** by the CPU.
- It is one of the essential features of the multitasking operating system.
- The processes are switched so fast – gives an illusion to the user that all the processes are being executed at the same time.
- Context switching can happen due to the following reasons:
  - a process of high priority comes in the ready state
  - an interruption occurs–process in the running state should be stopped
  - transition between the user mode and kernel mode is required

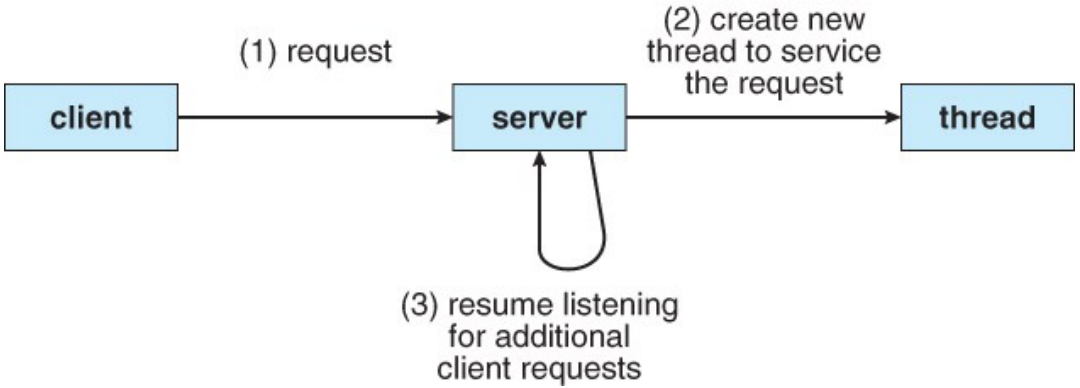


Threads

- So far, process has a single thread of execution
- A thread is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)
- Example: multithreaded processes (we saw in previous class)
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- For example in a **word processor**, a background thread may **check spelling and grammar** while a foreground thread processes **user input (keystrokes),** while yet a third thread **loads images from the hard drive**, and a fourth does **periodic automatic backups of the file being edited**.
- Another example is a web server

- Will be discussed in the next chapter.

Threads



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

- **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
- **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.
- **Economy** - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
- **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.

Process Scheduling

- The aim is to assign processes to be executed by the processor or processors over time, in a way that meets system objectives such as **response time, throughput, and processor efficiency**.
- In many systems, this scheduling activity is broken down into three separate functions such as **long-, medium-, and short term scheduling.**
- Relative time scales with which these functions are performed.
- Maximize CPU use, quickly switch processes onto CPU for time sharing

Process Scheduling

- **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 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

Process Scheduling

- 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
- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
- Remove process from memory, store on disk, bring back in from disk to continue execution: swapping
- A part of the swapping function.

Process Scheduling

- **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
  - **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Process Scheduling

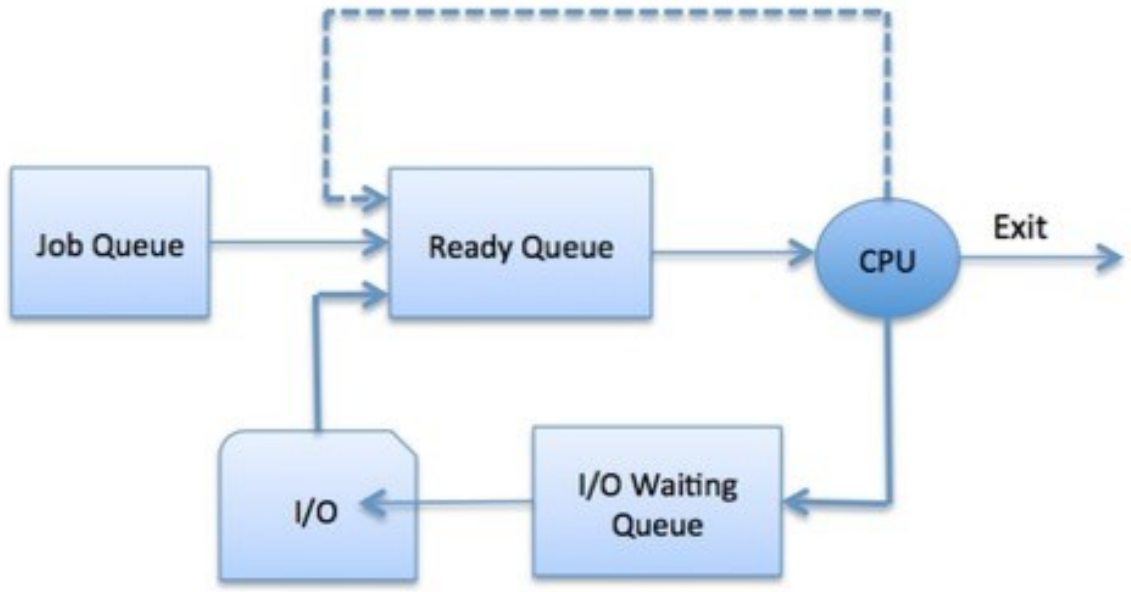
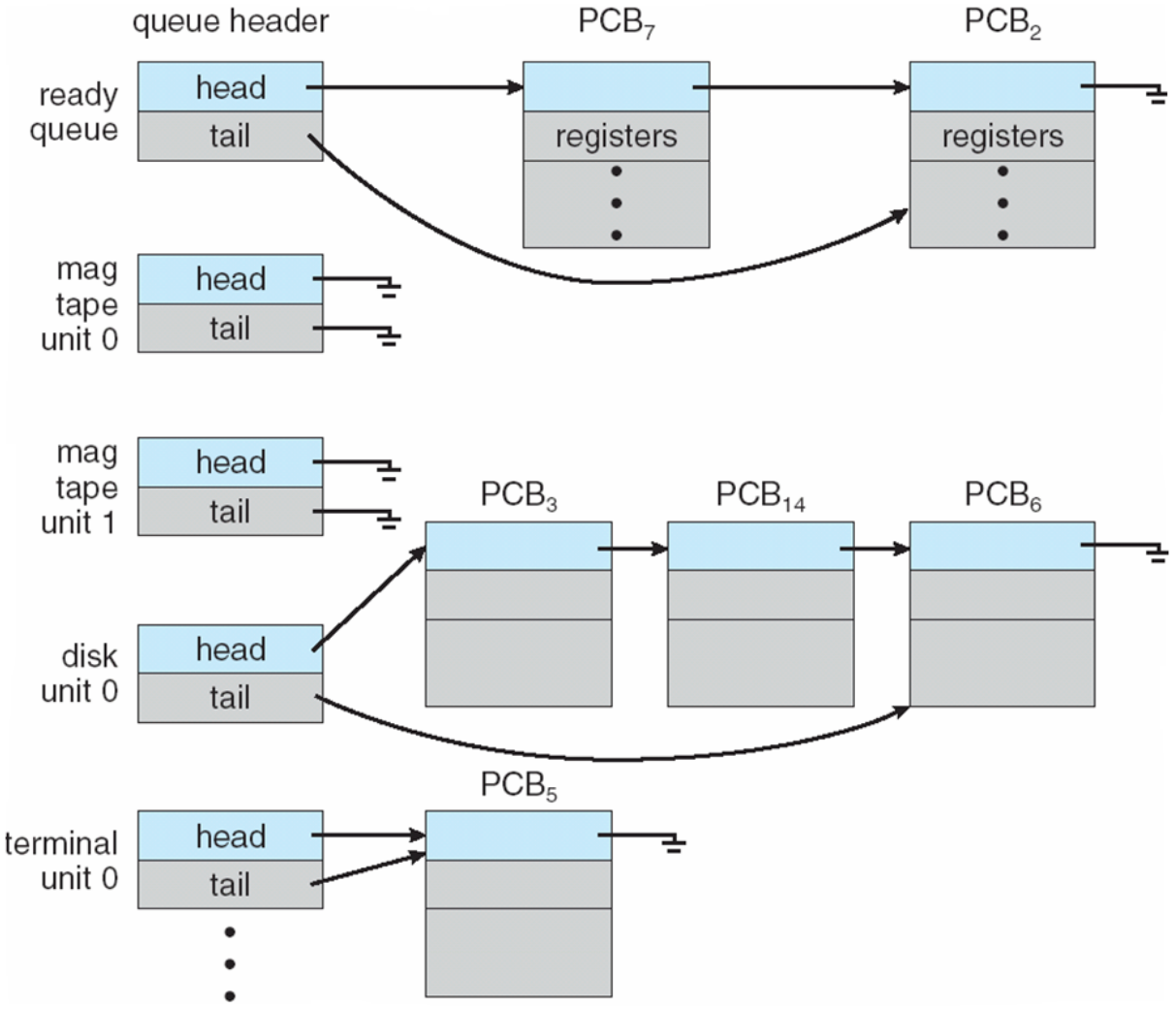


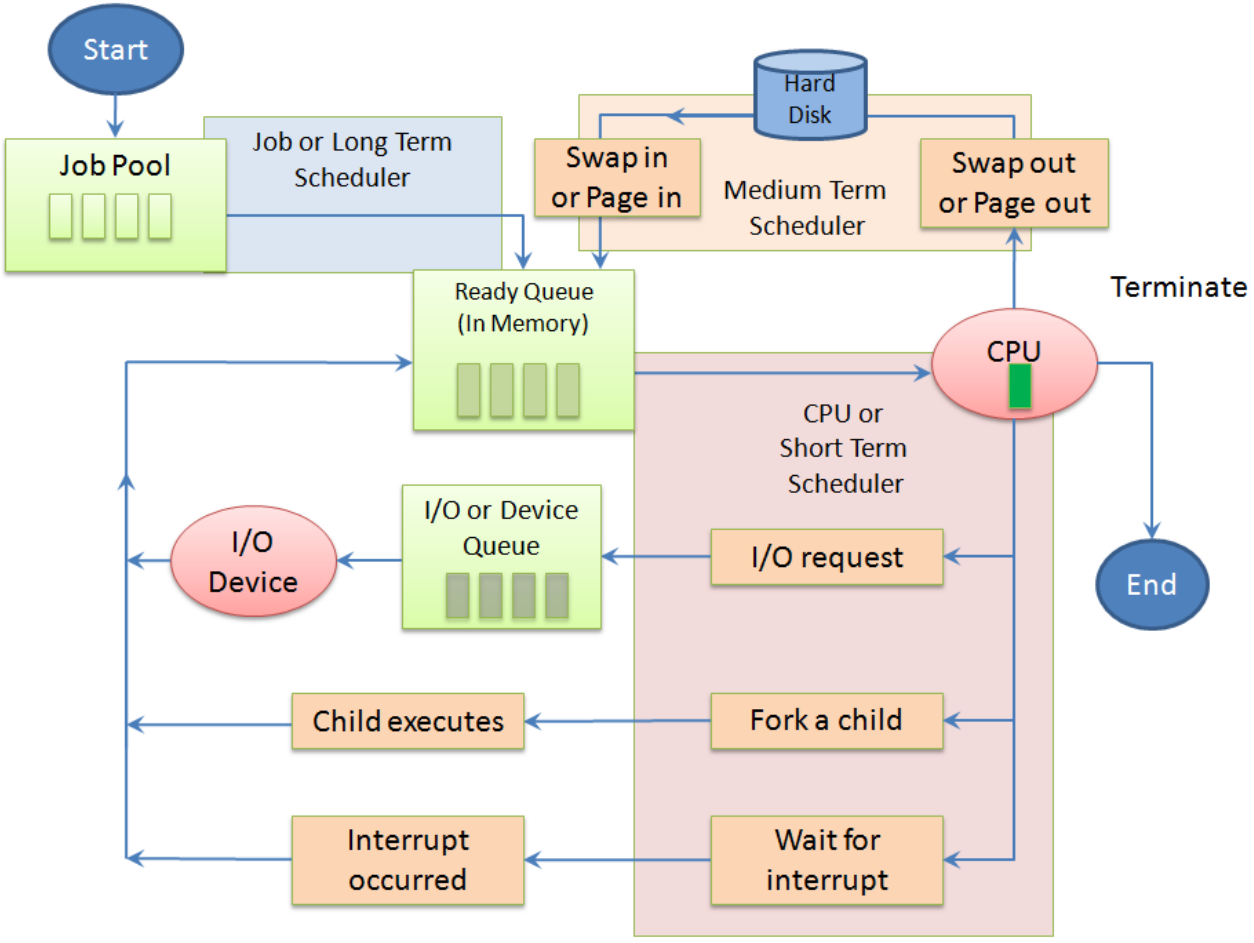
Figure: Ready Queue And Various I/O Device Queues

Process Scheduling

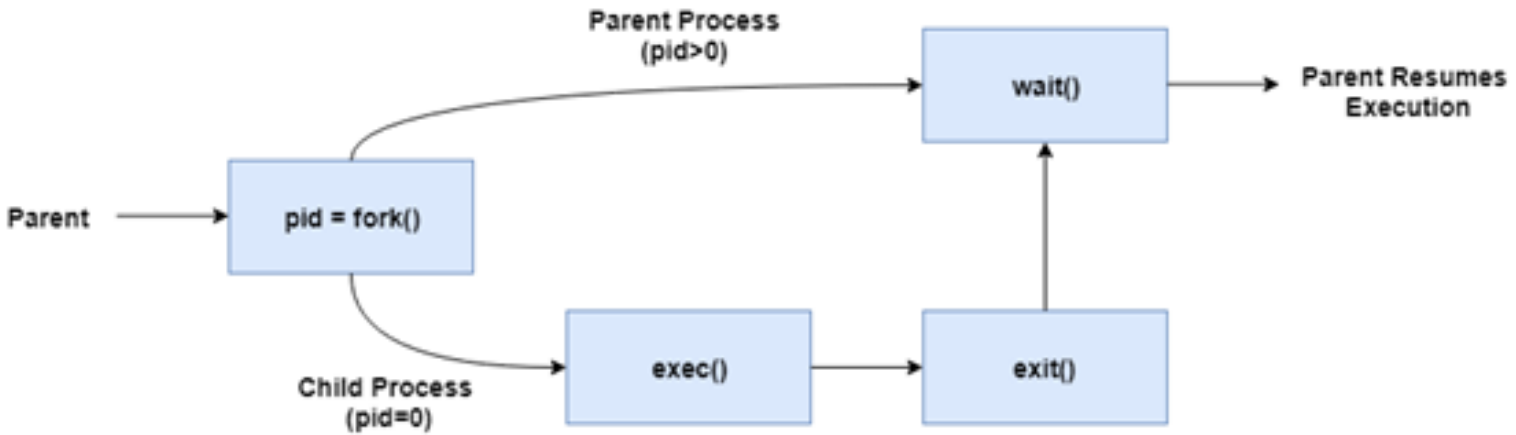
- Ready Queue And Various I/O Device Queues



Process Scheduling



Process Creation vs Process Termination



Process Creation Using fork()

Process Creation vs Process Termination

- A process may be created in the system for different operations. Some of the events that lead to process creation are as follows:
  - User request for process creation
  - System Initialization
  - Batch job initialization
  - Execution of a process creation system call by a running process

Process Creation vs Process Termination

fork():

- A process may be **created** by another process using fork().
- The creating process is called the parent process and the created process is the child process.
- A child process can have only one parent but a parent process may have many children.
- Both the parent and child processes have the same memory image, open files and environment strings.
- However, they have distinct address spaces.

Process Creation vs Process Termination

fork():

- The fork() system call returns either of the three values −
  - **Negative** value to indicate an error, i.e., unsuccessful in creating the child process.
  - Returns a **zero** for child process.
  - Returns a **positive** value for the parent process. This value is the process ID of the newly created child process.

Process Creation vs Process Termination

fork():

•Usually after fork() call, the child process and the parent process would perform different tasks.

•If the same task needs to be run, then for each fork() call it would run 2 power n times, where n is the number of times fork() is invoked.

Process Creation vs Process Termination

- When fork() is called **once**, the output is printed **twice** (2 power 1).
- If fork() is called, say 3 times, then the output would be printed 8 times (2 power 3).
- If it is called 5 times, then it prints 32 times and so on and so forth.
- getpid();
- getppid();

Process Creation vs Process Termination

```
#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;
}
```

Process Creation vs Process Termination

- **Process termination** occurs when the process is terminated.
- The **exit()** system call is used by most operating systems for process termination.
- Some of the causes of process termination are as follows:
  - A process may be terminated after its **execution is naturally completed**. This process leaves the processor and releases all its resources.
  - A child process may be terminated if its **parent process requests for its termination**.

Process Creation vs Process Termination

- Some of the causes of process termination are as follows:
  - A process can be terminated if it **tries to use a resource that it is not allowed to**. For example - A process can be terminated for trying to write into a read only file.
  - If an **I/O failure** occurs for a process, it can be terminated. For example - If a process requires the printer and it is not working, then the process will be terminated.
  - In most cases, if a **parent process is terminated** then its child processes are also terminated. This is done because the child process cannot exist without the parent process.
  - If a **process requires more memory** than is currently available in the system, then it is terminated because of memory scarcity.

Process Creation vs Process Termination

- A process can terminate in either of the two ways −
  - Abnormally, occurs on delivery of certain signals, say terminate signal.
  - Normally, using **_exit()** system call (or _Exit() system call) or **exit()** library function.
- The difference between _exit() and exit() is mainly the cleanup activity.
- The exit() **does some cleanup** before returning the control back to the kernel
- The _exit() (or _Exit()) would **return the control back to the kernel immediately**.

Process Creation vs Process Termination

- What happens if the parent process finishes its task early than the child process and then quits or exits?
- Now who would be the parent of the child process?
- The parent of the child process is init process, which is the very first process initiating all the tasks.
- To monitor the child process' execution state, to check whether the child process is running or stopped or to check the execution status, etc. the wait() system calls and its variants is used.

Process Creation vs Process Termination

- Following are the variants of system calls to monitor the child process/es −
  - wait()
  - waitpid()
  - waitid()
- The **wait()** system call would wait for one of the children to terminate and return its termination status in the buffer
- This call returns the process ID of the terminated child on success and -1 on failure.

Process Creation vs Process Termination

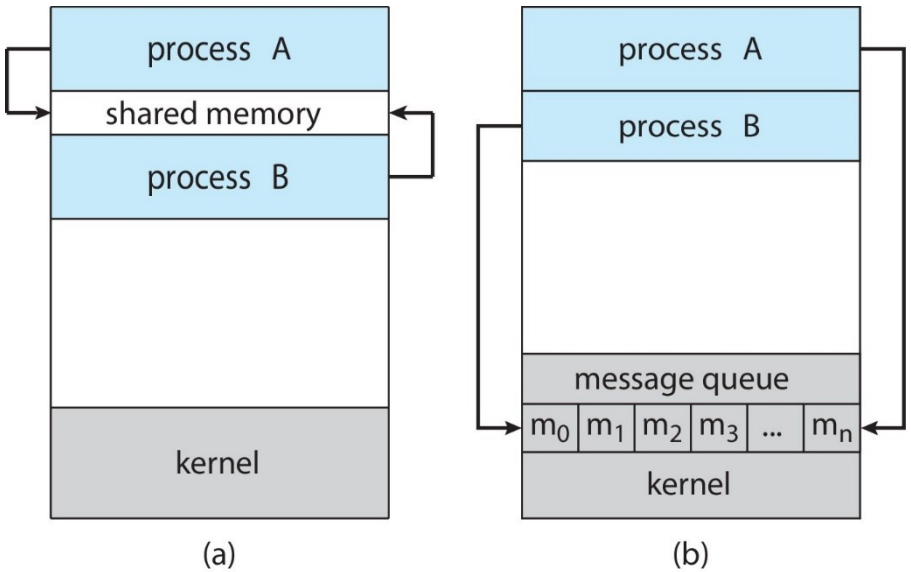
- The **waitpid()** system call suspends the execution of the current process and waits indefinitely until the specified children (as per pid value) terminates
- waitpid() system call waits only for the terminated children but this default behavior can be modified using the options argument
- **waitid()** call waits for the child process to change state.
  - **WCONTINUED –(SIGCONT)**
  - **WEXITED**
  - **WNOHANG**
  - **WSTOPPED**

Interprocess Communication

- A cooperating concurrent processes can affect each other, and independent concurrent processes cannot.
- Programmers want to write programs that implement cooperating processes.
- **Why?**
  - It may be a way for processes to share resources or information.
  - On a multiprocessor, multiple processes can get more work done on an application in less time.
  - Whether or not it is more efficient, it may be that one gets a better program design by using cooperating processes.
- For the reasons given above, OS should facilitate process cooperation by providing one or more interprocess communication (IPC) facilities.

Interprocess Communication

- The two basic models of IPC:
- **Shared-memory model** – somehow a region of memory is established that two different processes are able to use. Then the two processes communicate just by writing on and reading from areas in the shared memory.
- **Message-passing model** – messages are exchanged by using system calls, and thus the OS serves as an intermediary, a pick-up and delivery service, for the communicating processes.



Interprocess Communication

- The **shared memory** model of communication can be very fast
  - after the region of shared memory is established, usually with system calls, there is no further need for the processes to use system calls to communicate
- However communication is complicated by the fact that the processes have to properly synchronize their actions
- For example, if one process is writing and the other is reading, how does the reader keep from getting ahead of the writer? If the reader reads too fast, it could get ahead of the writer and start reading "garbage" from memory where nothing has been written yet.

Interprocess Communication

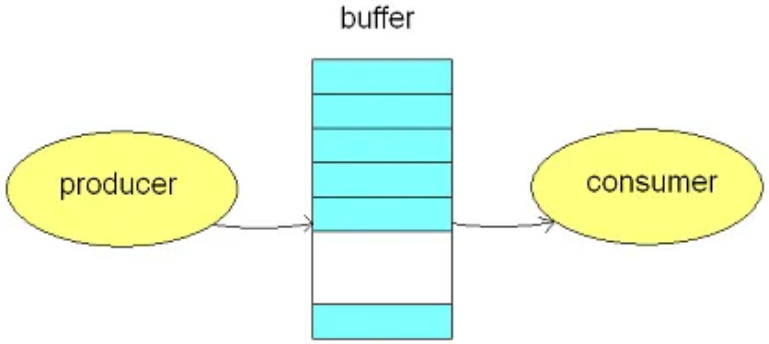
- **Message passing** is usually relatively easy to implement
- Going through the OS keeps the sender and receiver in sync with each other
- It can be inefficient when messages are large, because of the need to somehow copy the message from sender memory to recipient memory, and also because sending and receiving messages requires the overhead of a system call.

IPC In Shared-Memory Systems

- The sample producer-consumer code provides us with an example of a protocol that allows two processes to use a shared-memory method of communication and to keep in sync with each other.
- When two processes cooperate, it's common for them to have a producer/consumer relationship.
- One process creates a stream of items for the other process to use in some way.
- Errors can occur if the communicating processes don't stay in sync.

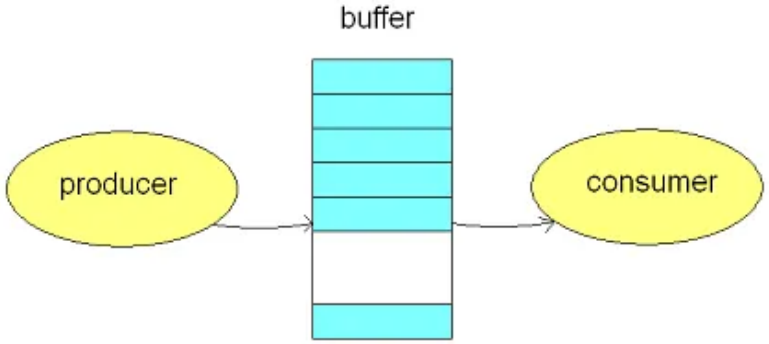
Producer-Consumer Problem

- It is also known as bounded buffer problem.
- In this problem we have two processes, producer and consumer, who share a fixed size buffer.
- Producer work is to produce data or items and put in buffer.
- Consumer work is to remove data from buffer and consume it.
- We have to make sure that producer do not produce data when buffer is full and consumer do not remove data when buffer is empty.



Producer-Consumer Problem

- The producer should go to sleep when buffer is full.
- Next time when consumer removes data it notifies the producer and producer starts producing data again.
- The consumer should go to sleep when buffer is empty.
- Next time when producer add data it notifies the consumer and consumer starts consuming data.
- This solution can be achieved using semaphores.



IPC In Shared-Memory Systems

- The sample producer-consumer code provides us with an example of a protocol that allows two processes to use a shared-memory method of communication and to keep in sync with each other.
- When two processes cooperate, it's common for them to have a producer/consumer relationship.
- One process creates a stream of items for the other process to use in some way.
- Errors can occur if the communicating processes don't stay in sync.
- Bounded Buffer–there is a fixed buffer size
- Unbounded Buffer–no practical limit on the size of the buffer

IPC In Shared-Memory Systems

Bounded Buffer:

Shared data

```
#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

IPC In Shared-Memory Systems

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;
}
```

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 */
}
```

Pipe ()

- Acts as a conduit allowing two processes to communicate

Issues:

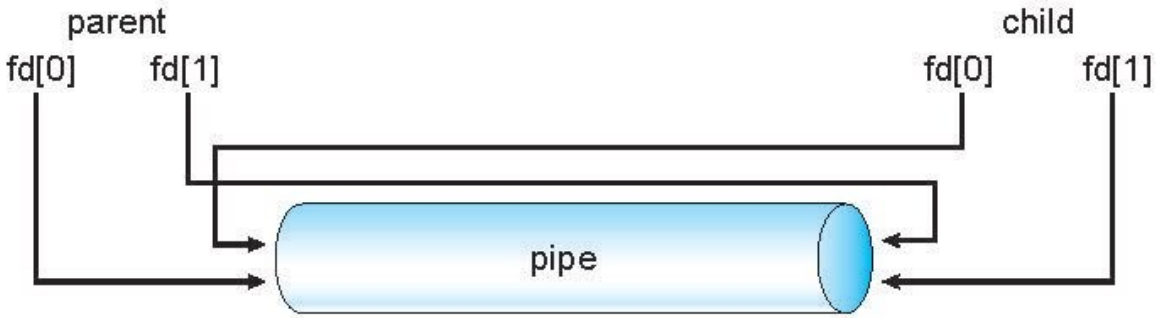
- Is communication unidirectional or bidirectional?
- In the case of two-way communication, is it half or full-duplex?
- Must there exist a relationship (i.e., parent-child) between the communicating processes?
- Can the pipes be used over a network?

Types of pipes:

- Ordinary pipes –
  - cannot be accessed from outside the process that created it.
  - Typically, a 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.

Pipe ()

- **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 therefore unidirectional
- Require parent-child relationship between communicating processes



Pipe ()

- **Named Pipes:**
- 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
- Provided on both UNIX and Windows systems

Process Synchronization

- Process Synchronization means **coordinating the execution of processes** such that no two processes access the same shared resources and data.
- It is required in a multi-process system where multiple processes run together, and more than one process tries to gain access to the same shared resource or data at the same time.
- It is necessary that processes are synchronized with each other as it helps avoid the inconsistency of shared data.
- For example:
  - A process P1 tries changing data in a particular memory location.
  - At the same time another process P2 tries reading data from the same memory location.
  - Thus, there is a high probability that the data being read by the second process is incorrect.

Process Synchronization

- **Sections of a Program in OS:**

  - **Entry Section:** This decides the entry of any process.
  - **Critical Section:** This allows a process to enter and modify the shared variable.
  - **Exit Section:** This allows the process waiting in the Entry Section, to enter into the Critical Sections and makes sure that the process is removed through this section once it's done executing.
  - **Remainder Section:** Parts of the Code, not present in the above three sections are collectively called Remainder Section.

Process Synchronization

- **Types of process in Operating System:**

- On the basis of synchronization, the following are the two types of processes:

  - **Independent Processes:** The execution of one process doesn't affect the execution of another.

  - **Cooperative Processes:** Execution of one process affects the execution of the other. Thus, it is necessary that these processes are synchronized in order to guarantee the order of execution.

Process Synchronization

- **What is Critical Section Problem?**

- A critical section is a segment of code which can be accessed by a **signal** process at a specific point of time.
- The section consists of shared data resources that is required to be accessed by other processes.
- The **entry** to the critical section is handled by the **wait()** function, and it is represented as P().
- The **exit** from a critical section is controlled by the **signal()** function, represented as V().

- In the critical section, only a single process can be executed. Other processes, waiting to execute their critical section, need to wait until the current process completes its execution.

Process Synchronization

- **Rules for Critical Section**

- **Mutual Exclusion:**
  - It implies that only one process can be inside the critical section at any time.
  - If any other processes require the critical section, they must wait until it is free.
- **Progress:**
  - If a process is not using the critical section, then it should not stop any other process from accessing it.
  - In other words, any process can enter a critical section if it is free.
- **Bounded Waiting:**
  - Bounded waiting means that each process must have a limited waiting time.
  - It should not wait endlessly to access the critical section.

Process Synchronization

- **Race Condition**

- Race condition is a situation where-
  - The final output produced depends on the execution order of instructions of different processes.

  - Several processes compete with each other.