# Operating Systems (UE18CS302)
Unit 1

Aronya Baksy

August 2020

## 1 What is an Operating System

**Definition 1.** A system program that acts as an intermediary between user and the computer hardware being used.

### 1.1 Goals of an OS

1. Execute user programs, make problem solving easier for the user
2. Make the system convenient to use
3. Use computer hardware resources in the most efficient manner possible

User convenience and optimized resource usage most of the time are contradictory goals, which involves a trade-off to achieve the optimum balance.

### 1.2 The User's view of an OS

The end user of a computer system defines the Operating System in the following manner:

- Offers **convenient** access to computational power, while being **easy to use** and offering **good performance**
- The end user does not care much about actual resource usage in the computer.

The user level definition of an OS may even vary based on the computation environment.

- Workstations and mainframe computers must share server resources fairly among all users.
- A handheld system is resource poor and must be optimized for high battery life and ease of use
- An embedded system or an IoT device does not have any User Interface (UI) and must be optimized for a specific task only.

### 1.3 The system view of an OS

From the point of view of actual computing, the OS may be defined as the following

- The OS is a **resource manager**. It allocates the computing resources in a fair and efficient manner to all programs that request for access to these resources.
- The OS is a **control program**. It controls the execution of programs and prevents errors/improper use of the computing resources by these programs.

### 1.4 Definition of an OS

The OS may be defined from either the system or user POVs. Hence there are no fixed definitions of an OS.
The following are two loose and intuitive definitions that may be used:

- ”All the software shipped by a vendor while selling an operating system”: this defn is suitable for a program like Microsoft Windows that is sold as a single unit comprising of application and system software.
- ”A kernel program that runs at all times on a computer system”: this defn is suitable for OSes belonging to the UNIX and Linux families that are available only as kernels without any extra application software on top.

## 2 Computer System Organization

A computer consists of hardware resources which are to be managed by the OS.
A program known as the **bootstrap** program (stored in read-only memory) initializes all the aspects of the system (CPU registers, I/O Devices) on startup/reboot, and loads the actual kernel of the OS to start execution.
Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become system processes.
Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

### 2.1 Interrupts

An event occurrence is defined by a signal known as an **interrupt**.
An interrupt is a signal provided by a peripheral device to the CPU requesting for some access to the CPU’s computation to serve its needs.
When an interrupt is received by the CPU, it transfers control to the appropriate **Interrupt Service Routine** (ISR) which is a function in the OS that handles the functionalities of that device specifically.
The **Interrupt Vector Table** (IVT) stores the address of the ISR corresponding to each type of device.
An **exception** is a software generated interrupt that is raised upon an error or at the request of the user.
The following mechanisms are involved in handling of Interrupts

- Storing the current PC and register values before interrupt was received by CPU
- Interrupt handling methods:
  1. **Polling:** Read the state (busy/available) of each I/O device in a list and then service the one who makes a request. Repeat continuously while CPU has power.
  2. **Vectored Interrupts:** The device notifies the CPU that it needs attention. The CPU reads the address of the appropriate ISR from the IVT and pauses the current program execution to handle the ISR.

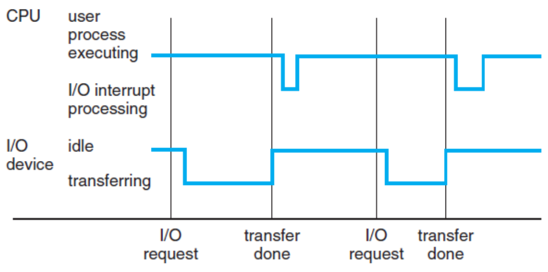


Figure 1: Interrupt Timeline for a single process

## 2.2 CPU

- $\geq 1$ CPUs, along with I/O device controllers that exist on a single bus to access the shared memory.
- CPUs and I/O devices compete for memory access cycles.
- The I/O devices and CPUs execute **concurrently**

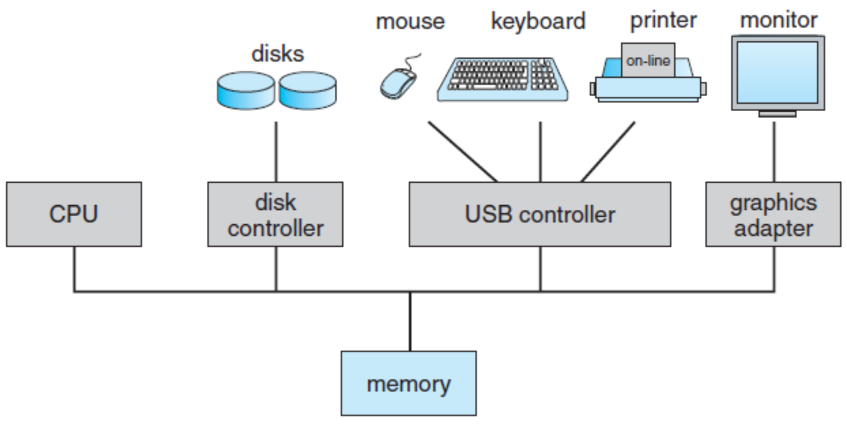


Figure 2: Hardware Resource Organization

## 2.3 Storage:

Every computer has the following levels of storage in it's hierarchy.

1. **Main Memory:** Volatile in nature, random access
2. **Secondary Memory:** Large capacity, non volatile and acts as an extension to the main memory. This is used for permanent storage of user data & programs.

In addition, the CPU contains its own **cache** and **registers** that are used to reduce the number of memory accesses to the slower main/secondary memories and hence improve CPU performance.

# 3 Computer System Architecture

The following is a categorization of different system architectures based on the number of general-purpose processors they use

## 3.1 Singe Processor System

- A single general-purpose CPU executes a general-purpose instruction set, including instructions from **user processes.**
- Single processor systems may contain other processing units which are special purpose units. They may be device specific controllers (I/O devices) or I/O management processors that move data rapidly between components (typically found on mainframes).
- The special purpose processors run a very limited and specific instruction set, and they cannot run user process instructions.
- Sometimes, they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status.
- In other cases, these processors function entirely autonomously, without the involvement of the OS.

## 3.2 Multiprocessor System

- These are also called **parallel** 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.
- Multiprocessing architecture has the following advantages:
  - Increased Throughput
  - Economies of scale
  - Increased reliability
- **Increased reliability** is an important advantage in performance critical industry applications.
- **Graceful degradation** is a property of multiprocessing systems where the failure of a certain proportion of the total hardware resources results in only a proportional drop in performance.
- Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.
- Two main types of multiprocessing systems are in use nowadays.
  - **Asymmetric multiprocessing:** Each processor is assigned a separate, specific task, with a single *boss* processor handling the functioning of all the others.
  - **Symmetric multiprocessing:** Each processor is capable of performing all the tasks on the system. All processors are peers (there is no boss-worker relationship). Each processor maintains its own set of registers and local private cache, but a shared main memory common to all processors exists.

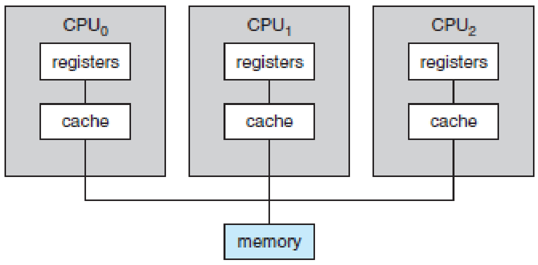


Figure 3: Symmetric Multiprocessing Architecture

- Some concerns in symmetric architectures arise from load distribution (one CPU idle, another one working at high load is undesirable), and I/O management (make sure the data reaches the right processor). These concerns are resolved by the sharing of certain data structures among all the processors.
- The following memory access models exist in Multiprocessing systems
  - **UMA:** Uniform Memory Access, meaning all CPUs take exactly the same time for a memory access (read/write) to the RAM.
  - **NUMA:** Non-Uniform Memory Access, meaning some parts of memory take longer to access than other parts (this leads to performance penalties).
- In contrast to multi-chip systems where each CPU is a discrete silicon wafer, the recent trend is towards **multi-core** systems where all the CPUs are organized on a single silicon wafer.
- Multicore architectures have the benefit of faster inter-CPU communication (intra chip vs inter chip) and lower power consumption (single chip vs multiple chips).

- All multicore systems are multiprocessor systems, but all multiprocessor systems are not multicore systems.
- **Blade servers** are systems in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between types of computers. In essence, these servers consist of multiple independent multiprocessor systems.

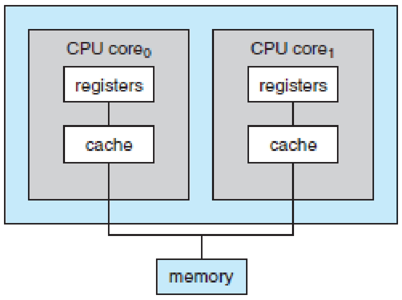


Figure 4: Multi-core Architecture (dual core)

## 3.3 Clustered Systems

- These consist of multiple systems called *nodes* connected to each other.
- Each node may be a single processor or a multiprocessor system.
- The generally accepted definition of clustered systems is that they share a common memory and are connected by an interconnect protocol like LAN or InfiniBand.
- The main application of clustered systems is in **high-availability** scenarios, where the service delivery must continue even when one or more nodes fail. This is done by building in various redundancies into the system. A software layer monitors all the clusters, and in case one node fails, the node that monitors can take up the memory and tasks done by the failed node and resume them.
- Clustered systems may be organized as follows
  - **Asymmetric:** One machine is in **hot-standby** mode while another runs an application. In case of failure of the latter, the former takes on the role of the active system.
  - **Symmetric:** $\geq 2$ hosts run applications and monitor each other simultaneously. This is a more efficient architecture but it needs more than 1 app to be run at a time.
- For an application to take advantage of the **high-performance computing (HPC)** environment afforded by a cluster, it must be explicitly parallelized.
- This is done by splitting the program into independent tasks that function separately on individual nodes, and a final combination of all the individual results from each task. (eg: the MapReduce framework in Hadoop)
- Memory concurrency and the avoidance of conflicting operations being carried out by different nodes on the shared memory is managed by a hardware called the **Distributed Lock Manager (DLM)**

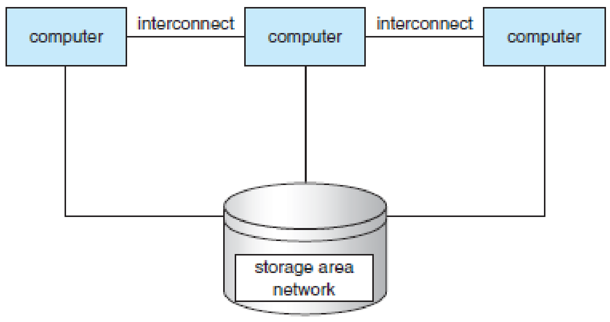


Figure 5: Clustered System Architecture

## 4 Structure of an Operating System

- **Multiprogramming** is a paradigm that allows the CPU to always have atleast one job to do (in terms of either I/O device handling or user program handling)
- The multiprogramming algorithm works as follows:
  - The OS maintains a job pool consisting of all processes residing on the disk that are awaiting access to the main memory.
  - The set of jobs that are actually in main memory is a subset of this job pool. The OS picks a job and begins to execute it.
  - While this job waits (for an I/O operation) a non-multiprogrammed OS might have made the CPU sit idle too.
  - But in a multiprogramming envt, the OS switches to another job during this wait time and executes it.
  - When this job needs to wait, the CPU switches to another job and starts running it, and so on.
  - Eventually the first job finishes waiting and it gets back its control of the CPU.
  - As long as at least one job needs to execute, the CPU is never idle.
- Multiprogramming systems only handle effective use of all the hardware resources, but they do not handle the user interaction with all these hardware resources.
- The **timesharing model** is a logical extension of multiprogramming, in which the CPU switches between various jobs at a very fast rate, and the switches occur so frequently that the users can interact with each program while it is running.
  - Time sharing OSes need an interactive system, where the user can directly communicate with the system hardware using I/O devices like a mouse, keyboard and display.
  - This necessitates a short response time - less than 1 second
  - A time-shared OS allows multiple users to access a single computer, where each user submits a job and the computer executes between all the jobs for some time interval each.
  - Choosing which job to load into memory when all the possible jobs cannot fit is the job of a set of algorithms called **job scheduling** algorithms.
- In a time-sharing system, the operating system must ensure reasonable response time. This goal is sometimes accomplished through **swapping**, whereby processes are swapped in and out of main memory to the disk.
- A more common method for ensuring reasonable response time is **virtual memory**, a technique that allows the execution of a process that is not completely in memory.

- The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual physical memory. Further, it abstracts main memory into a large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.
- Time Sharing OSes must also provide a file system, a disk management to manage the physical aspects of the file system, a resource protection mechanism from inappropriate use, as well as mechanisms for job synchronization and communication, to prevent jobs from getting stuck in a deadlock waiting for each other.

## 5 Operating System Operations

The OS must ensure that problems/errors occurring in a user program are localized only to that program, and do not affect any of the other processes running on the CPU.
Not doing so may lead to explicit errors (like the interruption of other processes) or more subtle errors (like one program erroneously overwriting the data of another).
Multi-mode operation of OSes, with the requisite hardware support is one approach to solve this problem.

### 5.1 Dual Mode Operation

- The two modes of operation are the **user mode** and the **kernel mode** (also known as the **supervisor mode**, **system mode**, or the **privileged mode**).
- A **mode bit** in hardware indicates the current mode of operation (1-user mode, 0-kernel mode)
- The mode bit distinguishes between applications run on behalf of the user (user mode) and on behalf of the system (kernel mode).
- The system boots up into kernel mode by default. After the OS is loaded, it switches into user mode and executes user jobs. Traps/interrupts cause the OS to switch into kernel mode from user mode.
- The system transitions from user mode to kernel mode whenever the user makes a system call (ie. a request for a system functionality). Before passing the control back to the user program, the mode bit is switched to 1 (user mode) again.
- The Dual mode operation allows for protecting the system hardware from errant users, and for protecting errant users from each other.
- Certain machine instructions, ie. those with the potential to harm the system, are designated as **privileged instructions** and can be executed only in kernel mode. These include the instruction to switch to kernel mode, as well as I/O control instructions, timer management and interrupt management.

### 5.2 Multi Mode Operation

- Multi mode operation allows other functions such as **Virtual Machine Management (VMM)** to be implemented in the OS. This is indicated by a separate mode in CPUs that support virtualization.
- In such a case, the mode is indicated by $> 1$ bit.
- The VMM is at a privilege level above the user but below the kernel, which is needed to change the CPU state while creating virtual machines.

We can now see the life cycle of instruction execution in a computer system. Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call.

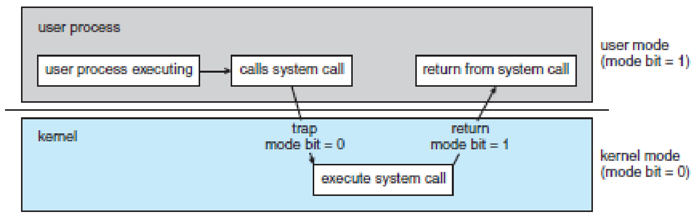


Figure 6: Switching between User and Kernel modes

### 5.3 System Calls

- System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf.
- A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic trap instruction or a specific instruction (as in the MIPS ISA)
- When a system call is executed, it is typically treated by the hardware as a software interrupt.
- Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode.
- The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting.
- Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers)
- The kernel verifies the validity of the request format, executes the request, and returns control to the next instruction after the system call.

Without the protection of dual mode operation, catastrophic results may arise where a user program overwrites the system's data (eg: MS-DOS running on early Intel x86 chips like the 8086 and 8088, modern x86 chips provide dual mode op.)

### 5.4 System Timer

- The job of the timer is to make sure that the user program returns control to the OS after a specific time period, so that the system can return to kernel mode.
- The timer is set to interrupt the computer after either a fixed delay or a variable delay.
- **Variable timers** are implemented using the system clock in combination with a down counter.
- Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

## 6 Computing Environments

### 6.1 Traditional Computing

- Traditional computing model consists of multiple individual workstation PCs connected to a single network with servers providing file storage and printing services.
- In the modern era, this has been replaced by an infrastructure where mobile computers as well as Desktops can connect to the company's internal network using a **portal**.

- Mobile computers can also use cellular or wireless networks using the company portal for further portability of information.
- Even in the home environment, now there is a prevalence of fast internet speeds, with more access to data. Many homes use personal **firewalls** to protect their networks from security breaches.
- In contrast to the time sharing approach between users, modern desktops/laptops/servers/mobile computers provide time sharing between multiple processes owned by the same owner (eg: multiple windows open by one user)

## 6.2 Mobile Computing

- Traditional mobile systems sacrificed processing power, screen size and memory capacity in exchange for portability.
- In modern mobile systems, this is no longer the case, with mobiles now being able to provide functions that are not possible or impractical on a desktop (eg: GPS navigation, gyroscope related functions, etc.)
- The wide range of apps that run on mobile devices are made possible by a large number of specialized **sensors** that are a part of mobile systems (eg: GPS, gyroscope, ambient light sensors, advanced camera systems, optical fingerprint sensors, AR sensors).
- Mobile systems use the IEEE 802.11 protocol (commonly called WiFi) or cellular data (3G/4G/5G protocol) to access the internet.
- Google's **Android** and Apple's **iOS** (formerly called iPhone OS) are the major players in this segment.

## 6.3 Distributed Computing

- A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked together to provide users with access to shared resources.
- Network access may be generalized as a form of file access, or as a separate protocol with network functions.
- There are various different of networks based on their range. They are **LAN** (room/building/campus range), **WAN** (building/city/country range), **MAN** (intra city), **PAN** (personal device range).
- Network transmission media also vary, such as copper cables, fiber optic cables, and wireless methods (microwave/radio wave/IR communication).
- A **network operating system** is an operating system that provides features such as file sharing across the network, along with a communication scheme that allows different processes on different computers to exchange messages.
- A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers.
- A distributed operating system provides a less autonomous environment. The different computers communicate closely enough to provide the illusion that only a single operating system controls the network.

## 6.4 Client-Server Computing

- Instead of a centralized architecture (where multiple terminals access the compute resources of a single mainframe), the functionalities handled by the mainframe systems are now being handled by PCs themselves, through a web interface.
- This necessitates that many of the systems be **servers**, which satisfy the requests made by **clients** over the internet.

- Servers can be classified into:
  - **Compute-Server systems:** The server executes the action requested by the server and returns the result to the client. (eg: DB servers)
  - **File-Server systems:** The server provides a file system interface where the client can create, update, delete and read files. (eg: web server using HTTP)

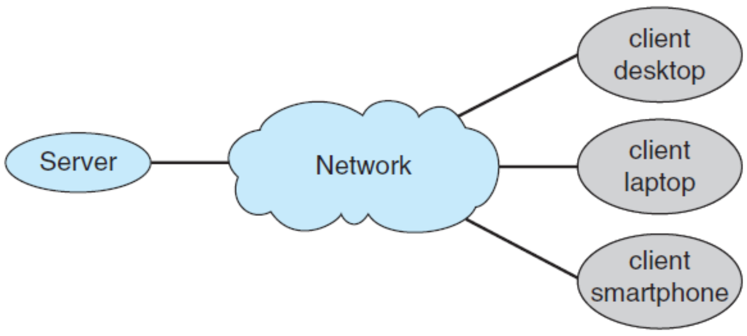


Figure 7: Client-Server architecture

## 6.5 Peer-to-Peer Computing (P2P)

- All nodes in a P2P system are considered to be peers, and each may act as a client or server depending on whether it requests or provides some service.
- In a client-server system, the total system performance depends on the server (ie. the server is a bottleneck) but in a P2P system services can be provided by ¿1 distributed nodes.
- Once a new node joins the peer network, it determines what services are available in the following ways:
  - The new node registers itself on a specialized *lookup service* on the network. A node that desires a service must find it on this lookup service and then make the request to the appropriate peer node.
  - The node that desires a service can broadcast its request to all the nodes in the network, and the appropriate node sends a response to this request (the rest of the nodes may ignore it). This approach needs a *discovery protocol* to allow a peer to discover the services offered by its fellow peers on the network.
- File sharing services such as Napster (centralized lookup table), Gnutella (broadcast file request to all nodes) and the BitTorrent client are examples of P2P systems. Skype is another example of a P2P client sending voice messages over the internet using a protocol called VoIP.

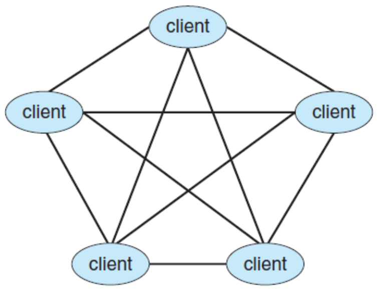


Figure 8: A peer-to-peer system with no centralized service

## 6.6 Virtualization

- Virtualization is a technology that allows operating systems to run as applications within other operating systems.
- Broadly speaking, virtualization is one member of a class of software that also includes **emulation**.
- **Emulation** is used when the source and target OSes are written for different CPU architectures. (eg: Apple's Rosetta for applications compiled on PowerPC architecture to run on Intel x86 CPUs, and Rosetta 2 for applications compiled on Intel x86 to run on ARM CPUs)
- That same concept can be extended to allow an entire operating system written for one platform to run on another. Emulation comes at a heavy price, however. Every machine-level instruction that runs natively on the source system must be translated to the equivalent function on the target system, frequently resulting in several target instructions.
- If the source and target CPUs have similar performance levels, the emulated code can run much slower than the native code.
- In **Virtualization**, an OS natively compiled for one CPU architecture runs within another OS compiled for the same CPU architecture.
- On laptops and desktops, a VMM allows the user to install multiple operating systems for exploration or to run applications written for operating systems other than the native host.

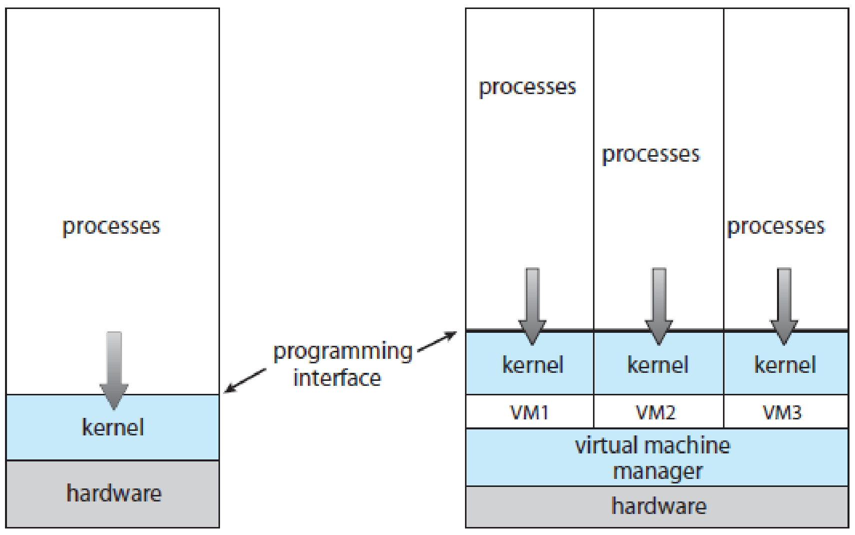


Figure 9: Architecture of the VMWare Virtual Machine

## 6.7 Cloud Computing

- Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network.
- In some ways, it is a logical extension of virtualization, because it uses virtualization as a base for its functionality.
- For example, the Amazon Elastic Compute Cloud (EC2) facility has thousands of servers, millions of virtual machines, and petabytes of storage available for use by anyone on the Internet. Users pay as per their usage of these services.
- The various types of cloud computing are as follows:
  - **Public cloud:** A cloud available via the internet for anyone willing to pay.
  - **Private Cloud:** A cloud run by a company for its internal use

- **Hybrid Cloud:** A cloud that includes both public and private components.
- **SaaS:** Applications available over the internet (eg: Google Office Suite)
- **PaaS:** A software stack ready for application use over the internet (eg: a DB server)
- **IaaS:** Servers/storage hardware available over the internet (eg: Google Cloud Storage solutions like BigTable, FireStore etc.)

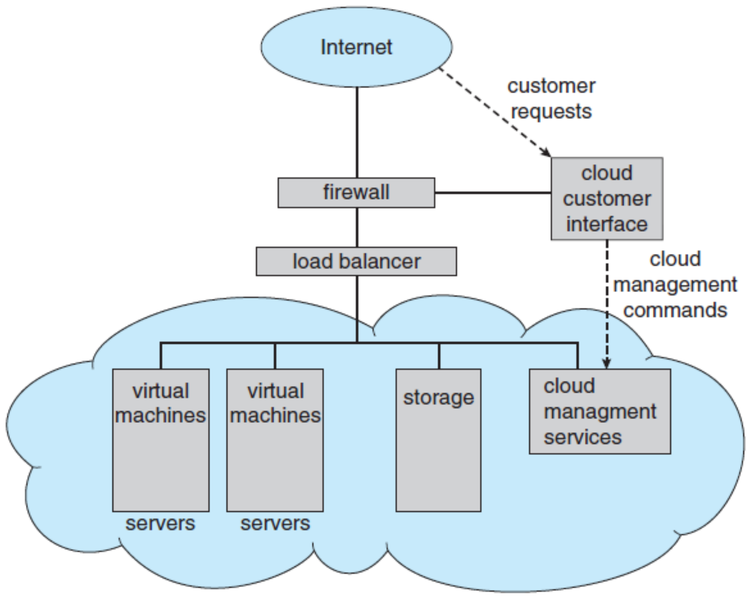


Figure 10: Cloud Computing (IaaS)

### 6.8 Real-Time Operating Systems

- Embedded systems are hardware devices with very specific purposes, and have little or no user interface for humans.
- They may be general purpose computers running standard OSes like Linux, special purpose devices running device specific OSes (network infrastructure like link-layer switches and routers), or even Application Specific Integrated Circuits (ASICs) which do not need an OS.
- A real-time OS is used when strict time constraints are placed on the processing/data flow. Data brought in from sensors is used to modify the surrounding environment in real time.
- A real time OS is said to function correctly only if the result is returned within the given time constraints.

## 7 Operating System Services

### 7.1 User Services

- **User Interface:** This may take three forms, a Command Line Interface (CLI) which relies on text-based individual commands that are run (eg: Linux), a batch OS where commands are entered into files which are executed (eg: bat files in Windows, Shell scripts in Linux OSes), or a Graphical User Interface (GUI) which uses familiar visual controls to enable the user to access OS functions (eg: Windows, Linux distros having GNOME/KDE desktop)
- **Program Execution:** The ability to load an user program from memory, execute it and terminate it (normally or abnormally).
- **I/O Management:** Manage the functionality of various I/O devices which the user is not allowed to do themselves (because of security reasons).

- **File system manipulation:** Read and write files, Create/Update/Delete/Search for files in the file system. Permission and access management for certain files.
- **Communication:** Inter Process Communication, which may be implemented via shared memory between processes, or using a message passing system between 2 or more processes, where communication takes place by means of packet transmission between processes.
- **Error Detection:** CPU/Memory errors, I/O device errors, user errors. Take appropriate action as per error type, and ensure correct and consistent computing.

### 7.2 System Services:

- **Resource Allocation:** Resource allocation to all users/jobs running at a given instant in time. Some resources (CPU cycles, main memory, file storage) have a specific allocation code, while others (such as I/O devices) have a more general request and release code.
- **Accounting:** Keeping track of resource usage type, grouped by user. Such statistics are used for billing in cloud instances and for system optimization by system admins.
- **Protection and Security:** Protection means ensuring that all access to system resources is controlled (processes don't overwrite each other's memory, interfere with each other's execution). Security means protection of system resources and data from outside intruders, by requiring user login with a password, and also the defense of I/O and network devices from illegal access attempts.

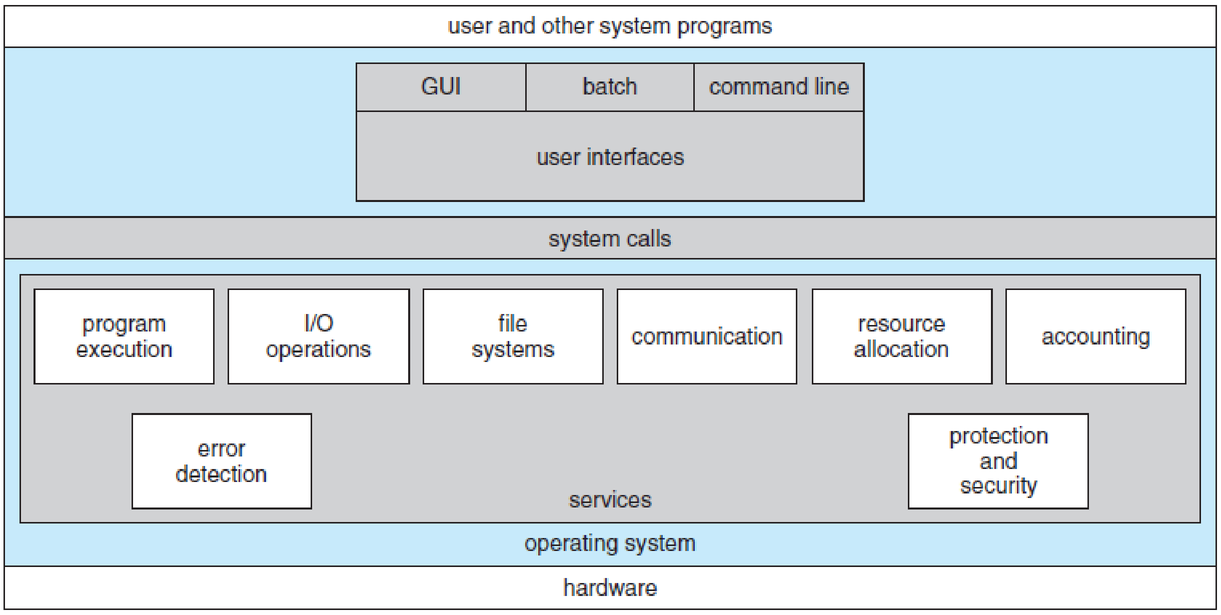


Figure 11: OS Services Architecture

## 8 Operating System Design and Implementation

The user and system goals defined above are the main driving forces behind design of a new operating system. These, along with any application specific requirements (like in the field of embedded systems) guide the design process of an OS.

### 8.1 Policy and Mechanism

- The **separation of policy and mechanism** is an important paradigm of software development.
- The policy is an idea of *what* is to be done, while the mechanism is *how* that policy will be done. (eg: the design of the system timer is a mechanism decision, but the timer delay is a policy decision)

- The two must be separated as policies may change, and without a clear separation while developing the OS, a change in policy will result in needing to overhaul large parts of the source code.
- On the other hand, developing a policy independent mechanism means that if policies do change later, only a certain number of parameters will have to be changed in order for the OS to function according to the new policy.
- Microkernel based OSes (like QNX, developed by BlackBerry for embedded systems) take this philosophy to its logical extreme, by implementing only a basic set of building blocks, which the user can combine to give more advanced functionalities to the OS.
- Consumer OSes like Windows and Mac OS X have a very tight integration between mechanism and policy to enforce a global look and feel to the OS.

### 8.2 Implementation

- Operating Systems consist of various modules written by many different teams of people over a long time, hence there is little scope of generalizing the implementation of an OS.
- Most early OSes were written exclusively in assembly instructions.
- Nowadays, high-level languages like C or even C++ are used. The choice of these 2 languages is their closeness to the system hardware.
- Modern OSes are more often than not in many different languages. Low level kernel modules are written in assembly, while higher level routines may be written in C, system software in C/C++ and application programs in ultra high level scripting or interpreted languages like Python, Java or PERL.
- The first system that was not written in assembly language was probably the Master Control Program (MCP) for Burroughs computers. MCP was written in a variant of ALGOL. MULTICS, developed at MIT, was written mainly in the system programming language PL/1.

## 9 Processes

### 9.1 The concept of a Process

**Definition 2.** A process is defined as a program under execution.

- A process is more than the source code of the program that it was created from. The following are the components of a process.
    1. **Text/Program Code:** Source code of the program that generated the process
    2. **Program Counter, Process Registers:** These identify the current activity of the process.
    3. **Stack:** A temporary data store for function parameters, local variables and function return addresses.
    4. **Global Data:** For all global and static variables
    5. **Heap:** Space for all memory allocated by the process at runtime
- Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.
- For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program.
- Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.
- It is also common to have a process that spawns many processes as it runs.

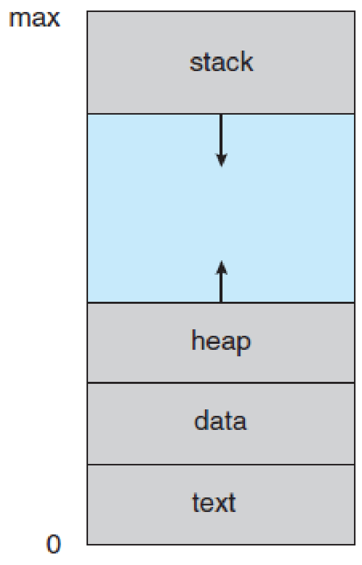


Figure 12: A process in Memory

## 9.2 Process States

- **New:** Process is being created
- **Running:** Process instructions are being executed
- **Waiting:** Process is waiting for an I/O operation or memory access.
- **Ready:** The process is waiting to be assigned to a processor.
- **Terminated:** The process has finished execution.

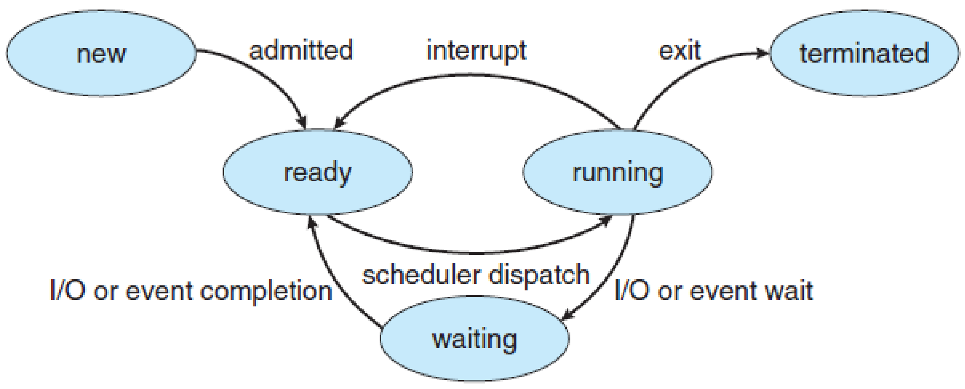


Figure 13: State Transition Diagram for a Process

## 9.3 Process Control Block (PCB)

- A PCB (also called a **task control block**) holds all the information that the CPU needs to identify and manage a process. The information held in a PCB is:
  1. **Process State:** One of the 5 states described above
  2. **Program Counter:** Address of the next instruction to execute.

3. **CPU Registers:** These vary by platform architecture, but in general they include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
   Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward
4. **Scheduling Info.:** Pointers to process queues, process priority
5. **Memory Management Info.:** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system
6. **Accounting Info.:** CPU hours and real time hours used, time limits, account numbers, job or process numbers
7. **I/O Device Info.:** List of all I/O devices allocated to the process, list of all files accessed by the process.

## 9.4 Threads

- Threads allow processes to perform multiple tasks at the same time
- All the threads belonging to a particular process share the same address space but have different program counters of their own.
- On multi core machines, threads can run in parallel, improving process performance.

## 9.5 Process Scheduling

To meet the objectives of multiprogramming and time sharing, the process scheduler selects tasks from a pool of processes and loads them into the main memory for exxecution.

### 9.5.1 Scheduling Queues

- As processes enter the system, they enter the **job queue**, which is a queue of all processes in the system.
- The scheduler selects processes from the job queue to enter the **ready queue**, which signifies loading the process into the main memory and preparing it for execution.
- A separate queue, called the **device queue** is maintained to list all processes that are waiting for I/O access or disk access.

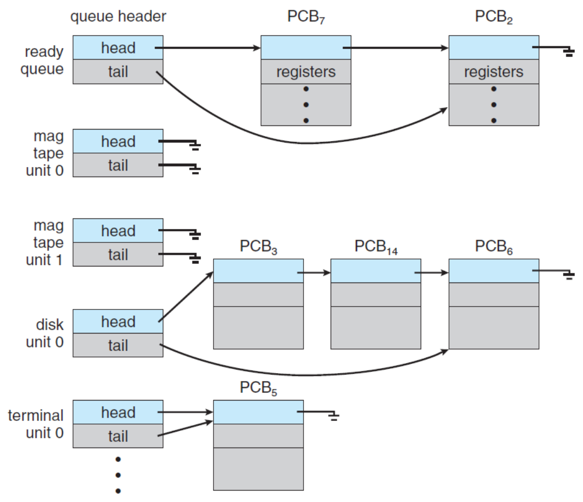


Figure 14: Ready Queue and various device queues

### 9.5.2 Process Schedulers

- There are three main types of schedulers that operate, the **long-term**, **short-term** and **medium term** schedulers.
- The **long-term scheduler**, or the job scheduler, selects processes from the job pool on the disk, and loads them into memory to make them ready for execution.
- The **short-term scheduler**, or CPU scheduler, selects one process from the ready processes and allocates the CPU to it.
- The short-term scheduler executes more frequently than the long term scheduler, at around once every 100ms.
- Speed is of the essence in a short-term scheduler. If 10ms are needed to schedule a process that executes for 100ms, then $\frac{10}{100+10} = 9.09\%$ of CPU time is wasted in scheduling.
- The long term scheduler operates much less frequently, around once every few minutes. The long term scheduler controls the **degree of multiprogramming** (the number of processes in memory).
- For a stable degree of multiprogramming, the rate of process creation and the rate of process departure from the system must be equal. Thus, the long-term scheduler may need to be invoked only when a process leaves the system.
- The long-term scheduler must select a good *process mix* of **CPU-bound** processes (that spend more time on computation than on I/O operations) and **I/O bound** processes (that spend more time on I/O access than on computation).
- If all the processes are I/O bound, the ready queue will always be empty and the short term scheduler will have nothing to do.
- If all the processes are CPU bound, then the I/O waiting queue will be always empty and the I/O devices will be unused.
- The **medium-term scheduler** is used to *reduce* the degree of multiprogramming by swapping out a waiting process from memory to the disk, and reloading it to the memory once it's wait is over. This process is called *swapping*.
- Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

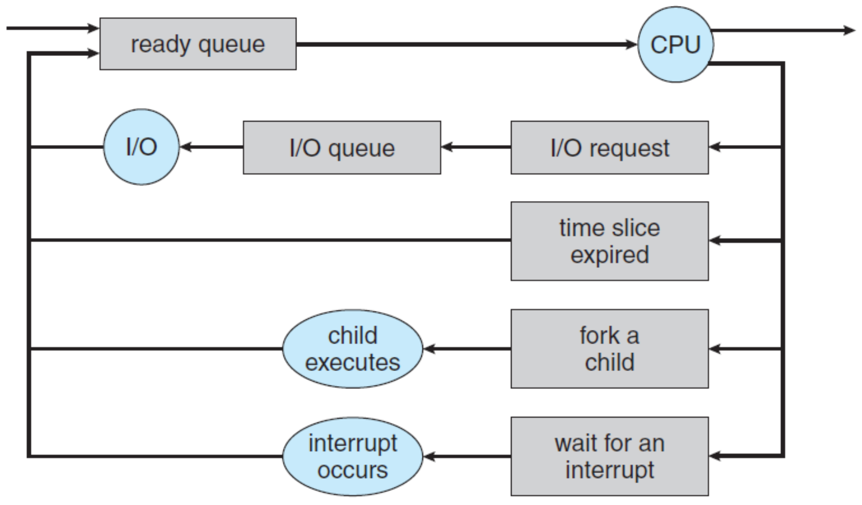


Figure 15: Queues in Process Scheduling

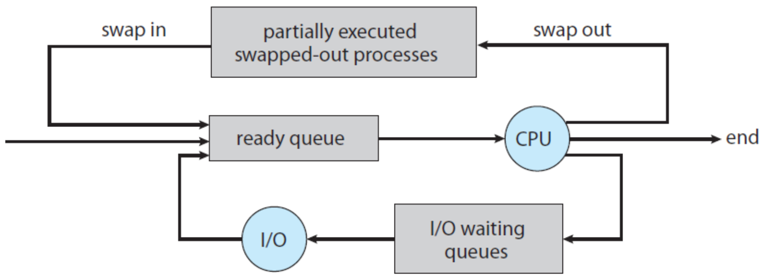


Figure 16: Medium-term Scheduling

## 9.6 Context Switching

- An interrupt may cause the CPU to pause the execution of the current process and move to kernle model to execute a system call.
- In such a case, the **context** of the running process (represented by its Process Control Block or PCB) must be saved so that it can be later retrieved.
- Switching the current context of the CPU involves a **state store** of the current running process, and a **state restore** of another process.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- The address space of the process also needs to be saved, to prevent another process from overwriting the data in the old process that was stored away.
- The time required for Context switching is purely an overhead, as no useful work is done at this time. It depends on hardware support, and greater OS complexity leads to larger context switching time.

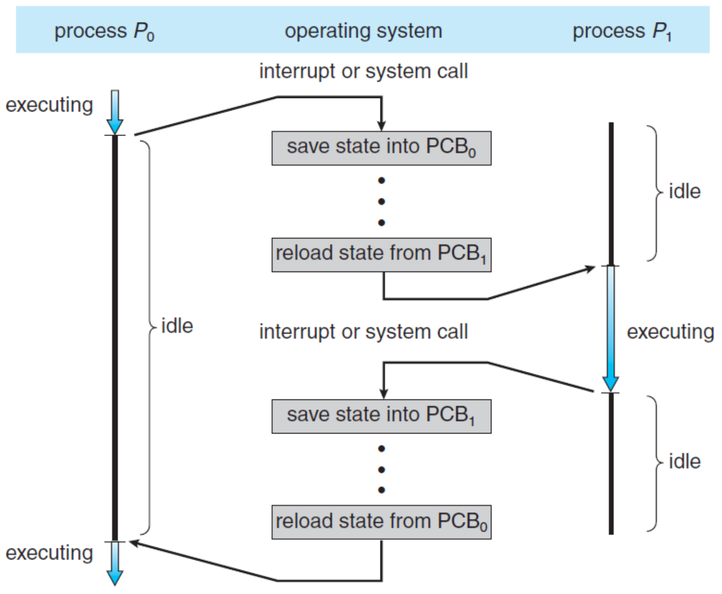


Figure 17: Context Switching from Process to Process

# 10 Process Operations

## 10.1 Process Creation

- A process, while executing, may create processes of its own. Such a situation leads to the creation of a **process tree**.
- In Linux, processes are identified by their **process identifier** (pid), which is an integer number.
- The `init` process is the root process for all user processes. It's children are the `sshd` (used to manage users logging in via SSH) and the `kthreadd` (used to create additional processes that perform tasks on behalf of the kernel) processes.
- A child process of a parent may be constrained to use only a subset of the parent's resources (CPU time, I/O devices, memory files), or it can directly obtain these from the OS.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.
- There are two possibilites for execution when a child process is created:
  1. Parent and child continue to execute concurrently
  2. Parent waits for some/all of its children to terminate.
- In terms of address space, the following possibilities exist:
  1. Child is duplicate of parent (same data and program as parent)
  2. Child process has a new program loaded into it.

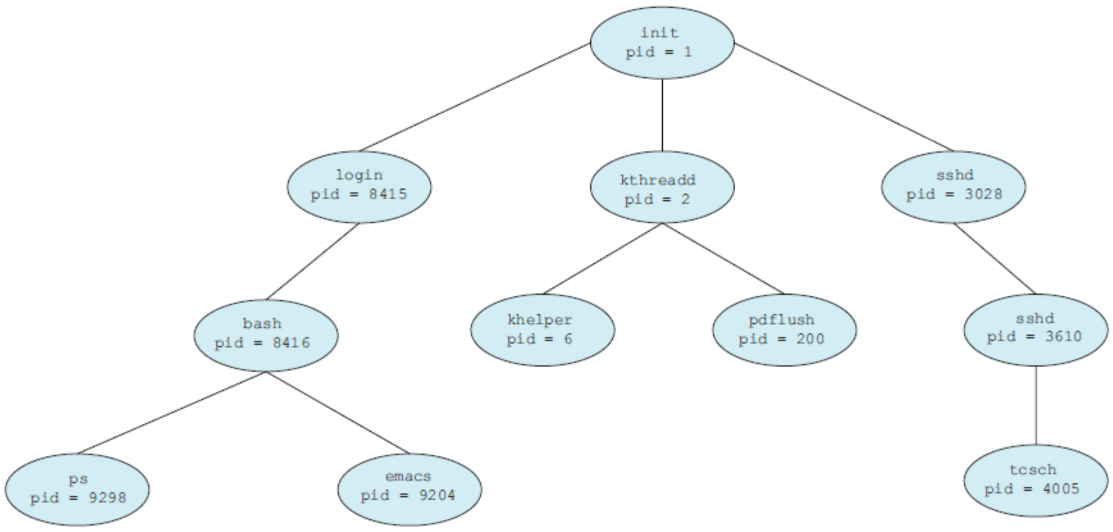


Figure 18: A Linux Process tree

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
    pid_t process_code;
    process_code = fork();

    if(process_code < 0) /* An error occurred */
    {
        fprintf(stderr, "Error occurred while forking\n");
        return 1;
    }
    else if(process_code == 0) /* child process */
    {
        execlp("/bin/ls", "ls", NULL);
    }
    else  /* Parent process waits for child process to finish */
    {
        wait(NULL);
        printf("Child process completed!\n");
    }
    return 0;
}
```

Process creation using `fork` system call in UNIX

- The `fork` call is used to create a copy of the current (orginal)process address space. This allows easy communication between parent and child processes.
- The return code of the `fork` call is 0 for the child process, and the non-zero pid of the child process is returned to the parent process.
- After `fork()` creates a new process, the `exec()` system call is used to replace the process' memory space with a new program, and execute it.

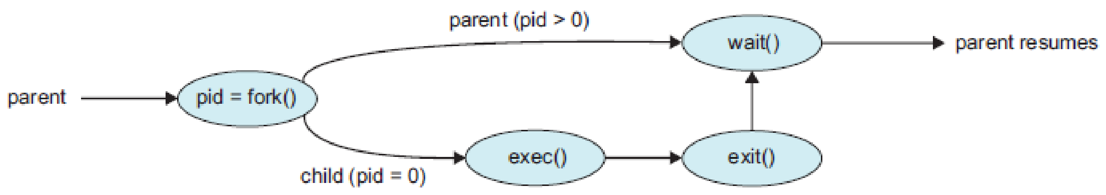


Figure 19: Process Creation using the `fork()` call

## 10.2 Process Termination

- A process terminates when the last instruction is executed and it asks the OS to terminate it using the `exit()` call.
- At this point, the process returns a status code to its parent using the `wait()` call, and the resources allocated to the process are deallocated.
- Only parents can kill their child processes, using the appropriate identifier. This can be for the following reasons:

  - Child using excess resources
  - Child process task is no longer needed
  - Parent is exiting, and OS doesn't allow child to run without its parent.

- **Cascading termination** is the OS' mechanism to terminate a process, by recursively killing off all its child processes first.
- Parent processes wait for the termination of their children by issuing a `wait()` system call. The `wait()` call allows the parent to know the exit status of its children and to know the pid of the child process that terminated.
- Until the parent of a process calls `wait()`, the terminated process must have an entry in the process table (as the exit status is contained in this entry). A process that has terminated but whose parent has not yet called `wait()` is called a **zombie** process.
- All processes briefly enter a zombie state when they state, as there is some delay between termination and the parent calling `wait()`.
- If the parent terminates instead of issuing a `wait()` call, the child processes become **orphan** processes. In UNIX and UNIX-like systems, this is handled by assigning the `init` process to be the parent of all the orphans.
- The `init` process periodically invokes `wait()`, thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's pid.

## 11 Process Scheduling

- While one process waits for some device or I/O response, the scheduler loads one more process to keep the CPU busy in the meantime.
- Such an arrangement increases the level of multiprogramming, and keeps the CPU busy at all times, with no idle periods of inefficiency.
- The success of this arrangement is due to the fact that processes commonly alternate between periods of heavy CPU activity (a **CPU burst**) and periods of high I/O activity (an **I/O burst**).
- The frequency distribution of CPU and I/O bursts (CPU bound progs have few long CPU bursts, I/O bound progs have many short CPU bursts, and vice versa for I/O bursts) is important when choosing an appropriate scheduling algorithm.

### 11.1 Preemptive and Non Preemptive Scheduling

- CPU scheduling takes place in the following four situations:
  1. Process switching from **running** to the **waiting** state, for an I/O or memory access, or invocation of wait() for child process termination.
  2. Process switching from **running** to **ready**, when an interrupt occurs.
  3. Process switching from **waiting** to **ready** state, for example when an I/O op. completes.
  4. Process **terminates**
- The situations 1 and 4 come under the purview of **non-preemptive** or **cooperative** scheduling.
- Situations 2 and 3 come under **preemptive** scheduling.
- Under non-preemptive scheduling, once a process is allocated the CPU, it takes control of the CPU until it has to wait or it terminates.
- Preemptive scheduling can lead to race conditions. If one process is preempted while it is writing to some data location, and the new process reads from that location, then that data is an inconsistent state.

- Preemptive scheduling can also affect kernel design. If a kernel process is preempted while it is changing some kernel data (eg: an I/O queue), then chaos ensues. UNIX family OSes solve this problem by waiting for the entire system call or I/O op to complete before performing the context switching. This is a working solution but is not feasible in real time OSes.
- The **dispatcher** gives control of the CPU to the process that is selected by the short term scheduler. This involves a context switch, a switch to user mode, and jumping to the appropriate location in the user program to start execution.
- Every process switch invokes the dispatcher, and the time taken for the dispatcher to stop one process and start another is called the **dispatch latency**.

## 11.2 Scheduling Criteria

- **CPU Utilization:** The fraction of time the CPU is kept busy. Theoretically can be anywhere from 0 to 100%, but real world values range from 40% to 90%.
- **Throughput:** Number of processes completed per unit time.
- **Turnaround Time:** Interval between process submission and process completion. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting Time:** Sum of all time periods spent by process in the wait queue.
- **Response Time:** Time between process submission and first response.

**maximize:** CPU Util, Throughput
**minimize:** Turnaround time, wait time, response time

## 11.3 Scheduling Algorithms

### 11.3.1 First Come First Serve (FCFS) Scheduling

- Processes are added to the ready queue in the order in which they arrive.
- The average wait times for FCFS depend very heavily on the process mix, and the burst times for each process. A long process arriving behind many short ones can lead to short wait time, but a long process in front of short ones may lead to longer wait times.
- The effect of a long process in front of many short processes, making the short processes wait for CPU time, is called the **convoy effect**.
- The FCFS is non preemptive because it allocates CPU time entirely to the new process, and it causes problems in time sharing systems where each user must get CPU time at regular intervals.

### 11.3.2 Shortest Job First (SJF) Scheduling

- The process with the shortest CPU burst time is selected for CPU time.
- To predict the length of the next CPU burst, in a long term system the process time limit per user can be used, which is supplied by the user.
- For a short term scheduling situation, the next CPU burst time is predicted using an exponential average of the previous CPU burst times.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n \tag{1}$$

  Where $\tau_i$ refers to the *predicted* CPU burst value at the time index $i$, and $t_i$ refers to the *actual* CPU burst time at time index $i$.

- The parameter $\alpha$ is chosen to be a constant with value $< 1$, most often a value of 0.5
- The preemptive version of SJF, also called shortest waiting-time first, where the process with the shortest CPU burst time is chosen, and the current process is pre-empted in its place.

### 11.3.3 Priority Scheduling

- Each process is associated with a priority (a non-negative integer) and the CPU is allocated to the process with the highest priority.
- Allocating priorities to processes can be due to external or internal reasons.
- Internal factors could be user time limits, memory requirements, the number of open files, and even the ratio of average I/O and CPU burst times.
- External factors are primarily influenced by human considerations of process importance.
- Priority scheduling may be preemptive (if the priority of the incoming process is higher than that of the currently running process then the currently running process is preempted), or non-preemptive (if the incoming process has a higher priority than the currently running one, it is placed at the head of the ready queue).
- Low priority processes may be blocked from CPU access by a steady stream of high priority processes arriving constantly. This problem is called **starvation**.
- The problem of starvation is resolved by a process called**aging**, wherein the priority of a waiting process is increased after fixed intervals of time.

### 11.3.4 Round Robin Scheduling

- The fundamental unit of time in an RRS algorithm is called the **time slice** or **time quantum** (denoted as $q$).
- Each process is assigned a time equal to $q$ on the CPU, and the ready queue is treated as a circular queue. Once all processes in the queue have got one $q$ unit of CPU time, the allocation circles back to the first process in the ready queue.
- No process has a wait time of more than $(n-1)q$ where $n$ is the number of processes on the ready queue. Every process gets an allocation of $1/n$ of the CPU time.
- The system timer is rigged to throw an interrupt and cause a context switch after every quantum of time.
- A very small value of $q$ can lead to large overheads due to frequent context switching, while a large value of $q$ can lead to the RRS algorithm becoming a FCFS system instead.

### 11.3.5 Multilevel Queue Scheduling

- When processes can be classified into groups (eg: **foreground** (interactive) and **background** (batch) processes), the MQS scheduling is used.
- The groups would each have different scheduling requirements, and one group (foreground) would have an externally defined priority over the other.
- The MQS algorithm partitions the ready queue into several smaller queues, each with its own scheduling algorithm and associated properties.
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- Between queues, there could be an ordering of queues such that processes in one queue have higher priority over processes in another queue.
- An alternative approach to inter-queue scheduling is to have a time slice of the CPU for each queue to submit processes to the CPU. (eg: foreground queue gets 80% of CPU time, and background queue gets 20%)

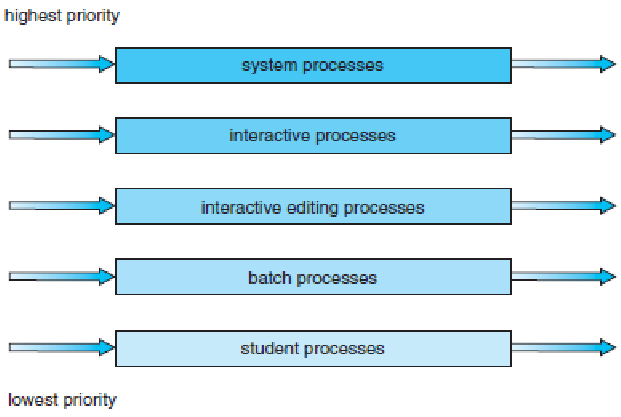


Figure 20: Multilevel Queue Scheduling

#### 11.3.6 Multilevel Feedback Queue Scheduling

- This allows processes to move between all the various process queues, with the intent of separating processes according to their CPU burst times.
- If a process uses too much of CPU time, it may be moved to a lower-priority queue, with I/O bound and interactive processes in the highest priority queues.
- To prevent the problem of starvation, aging is implemented in this algorithm by moving a process to a higher priority queue.
- A Multilevel Feedback Scheduling algorithm is characterized by:
  1. The number of queues, and their individual scheduling algorithms
  2. The method used to promote or demote a process from one queue to another
  3. The method used to determine which queue an incoming process will enter when it needs service.

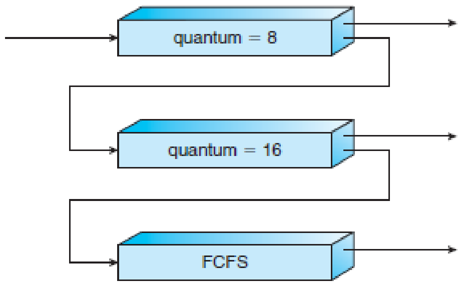


Figure 21: Multilevel Feedback Queue Scheduling

## 12 Multiprocessor Scheduling

### 12.1 Approaches

- The **asymmetric** multiprocessing approach involves one master CPU that manages all scheduling, I/O and system activites while the rest execute only user code.
- Only one processor needs to access the data structures for scheduling, hence data sharing and concurrency problems are fewer here.
- In a **symmetric** multiprocessor, each processor is self scheduling, with each having either its own ready queue or a shared ready queue, but each having their own scheduling algorithms.
- SMP systems having common data structures must ensure concurrency and

### 12.2 Processor Affinity

- Processor affinity is the phenomenon where a process perfers to execute on one particular processor.
- There are two types: **soft affinity** implies that the system tries to keep the process on that processor, but will migrate it if needed, while **hard affinity** allows the process to specify a subset of the processors on which it will execute.
- Affinity allows for performance improvement via caching, as keeping a process on one processor allows for its data to be present in cache, hence reducing memory accesses to main memory.
- In boards with NUMA memory, the scheduler and memory placement algos work together such that processes using one particular processor are allocated memory that is faster to access for that processor.

### 12.3 Load Balancing

- In order to ensure that most processors does not remain idle while only one works extra hard to handle all processes, load balancing is important. This is implemented by moving processes between queues.
- **Push migration** involves a specific task that checks all the CPUs and moves tasks from overloaded CPUs to idle ones. **Pull migration** occurs when idle CPUs pull processes from overloaded CPUs.
- Load balancing nullifies the performance gain derived from processor affinity via caching, as each migration involves invalidating the old CPUs cache and copying those values to the new CPU.

### 12.4 Multicore Scheduling

- Multicore systems have more complicated scheduling due to multithreaded architecture.
- Each core has multiple hardware threads that it can switch between when one thread is busy waiting for data to be available in main memory.
- In a **coarse-grained** multithreading system, the threads switch only when a high-latency event like memory access occurs. This is expensive as it involves flushing the instruction pipeline for the current thread and refilling it for the other thread.
- In **fine-grained** multithreading, threads switch more often, at the boundary of every instruction cycle. Thread switching is less expensive here as such systems have extra hardware logic to handle this.
- Multithreaded, multicore processors have 2 levels of scheduling. The first level is the OS choosing which software thread to run on each hardware thread, while the second level is the CPU choosing the hardware thread to run.

## 13 Real-Time CPU Scheduling

- **Soft real-time systems** provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes.
- **Hard real-time systems** have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all.
- Different events have different latency requirements in a real-time system, based on the evironmental factors. The two types of delays that are to be minimized are:
  1. **Interrupt Latency:** The interval between interrupt arrival at the CPU and the start of its ISR. This time includes the time taken to finish the current instruction, save the process state and to switch context to the ISR.
  2. **Dispatch Latency:** The amount of time needed for the scheduler to stop one process and schedule the next one in the ready queue. This is minimized by using preemptive scheduling.

## 13.1 Scheduling in Real Time systems: Priority Scheduling

- Real-time systems must support preemptive, priority based scheduling (for soft real time) and additionally some mechanism to ensure that processes will not exceed their deadline of execution (for hard real time).
- Processes in real-time OSes are said to be periodic, ie. it requires the CPU at regular intervals.
- Once a process gets the CPU, it has a processing time $t$, a deadline $d$ and a period $p$. These must follow $0 < t \leq d \leq p$.
- The rate of a task, a measure of its priority is measured as $1/p$.
- If the process has to announce its deadline needs to the scheduler, the **admission control program** can decide to add the process to the ready queue or not based on whether it can be serviced within the given deadline.

## 13.2 Rate-Monotonic Scheduling

- This uses a static priority policy with preemption
- Each process is assigned a priority that is $1/p$ where $p$ is the period of the process.
- It is assumed that every time a process acquires the CPU, the duration of its CPU burst is same.
- Rate monotonic scheduling is optimal, but CPU time is bounded and it is not possible to maximise CPU utilization always. The worst case CPU utilization for $N$ processes is $N(2^{1/N} - 1)$ which tends to $ln(2)$ as $n \longrightarrow \infty$.

## 13.3 Earliest Deadline First Scheduling (EDF)

- Priorities are dynamically assigned based on deadline (processes with earlier deadlines have higher deadlines).
- When a process becomes runnable, it announces its deadline requirement to the scheduler so that new priorities can be calculated.
- This is not a preemptive method, so higher priority processes cannot move running processes off the CPU.
- Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst.
- Theoretically, EDF is capable of acheiving 100% CPU utilization such that all processes meet their deadlines, but in practice this is affected by context switching overheads.
- At each step, **check for which process has the closer deadline**.

## 13.4 Proportional Share Scheduling

- $T$ shares of CPU time are allocated to all applications, each application receiving $N$ shares and hence owning the CPU for $N/T$ fraction of time
- The admission control program loads the program into the queue only if its requested number of shares are available at that moment.

# 14 Linux Scheduling

## 14.1 Pre version 2.5

- Before Linux 2.5, a variant of the UNIX scheduler was used.
- This algorithm was not built with SMP (symmetric multiprocessing) systems in mind, hence was discarded in version 2.5

### 14.2 Version 2.5-2.6.22

- Linux version 2.5 introduced the O(1) scheduler, named because of its constant time complexity.
- The algorithm maintains arrays of active and expired processes. Each process is given a fixed time quantum, after which it is preempted and moved to the expired array
- Once the active array is empty, the pointers are swapped and the expired array becomes the new active array, while the new expired array is empty (the old active array).
- This scheduler led to performance gains on SMP systems but did not suit the interactive processes that run on desktop systems.

### 14.3 Version 2.6.23 and beyond

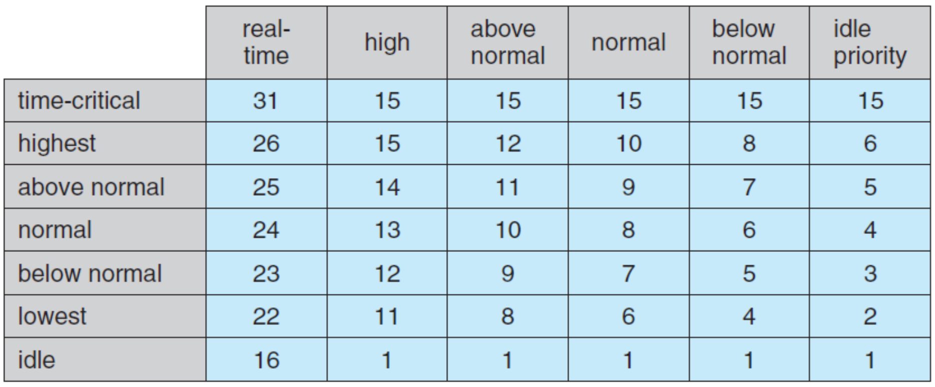
- The **Completely Fair Scheduler** (CFS) is the default Linux scheduling algorithm for versions 2.6.23 and beyond of Linux.
- Scheduling is based on **priority classes**, with each class assigned a certain priority. The scheduler selects the highest priority task from the highest scheduling class.
- The standard kernel implements two classes, one based on the CFS and the other one for real-time processes.
- The proportion of CPU time alloted to each process is based on the calculated **nice value** of that process. The nice value can be between -20 and +19, and is so called because an increase in the nice value suggests that the process is being *nice* to other processes by lowering its relative priority. Default nice value is 0
- CFS doesn't use discrete values of time slices and instead identifies a targeted latency, which is an interval of time during which every runnable task should run at least once. Proportions of CPU time are allocated from the value of targeted latency.
- The CFS uses the **vruntime** (virtual runtime) variable to store how much time the process has been running for.
- The vruntime of a process is associated with a decay factor, with low priority tasks having a higher decay factor than the higher priority tasks.
- High priority processes have vruntime < physical runtime, low priority processes have vruntime > physical runtime, and processes at normal priority (class 0) have vruntime = physical runtime.
- Real-time processes are assigned priorities between 0 and 99, while normal tasks are assigned priorities from 100 to 139. These two ranges map into a global priority scheme wherein numerically lower values indicate higher relative priorities.
- Normal tasks are assigned a priority based on their nice values, where a value of –20 maps to priority 100 and a nice value of +19 maps to 139.

## 15 Windows Scheduling

- Priority based preemptive scheduling algorithm.
- The **dispatcher** is the portion of the Windows kernel that schedules processes.
- 32 priority classes are used, divided into two. **Variable classes** are assigned priority of 1-15 while **real-time classes** are assigned priorities 16-31. Thread 0 is the memory management thread.
- If no ready threads are found, the dispatcher sends the idle thread.
- The Windows API relates these 32 classes to 6 classes and subclasses of its own. The 6 main subclasses according to the Windows API are:
    1. IDLE_PRIORITY_CLASS

2. BELOW_NORMAL_PRIORITY_CLASS
3. NORMAL_PRIORITY_CLASS
4. ABOVE_NORMAL_PRIORITY_CLASS
5. HIGH_PRIORITY_CLASS
6. REAL_TIME_PRIORITY_CLASS

- All threads in all classes except REAL_TIME_PRIORITY_CLASS have variable priority, meaning their priorities can change.
- Within these classes, there are relative priorities a thread can have, which are denoted as:
  1. IDLE
  2. LOWEST
  3. BELOW_NORMAL
  4. NORMAL
  5. ABOVE_NORMAL
  6. HIGHEST
  7. TIME_CRITICAL



| | real-time | high | above normal | normal | below normal | idle priority |
|---|---|---|---|---|---|---|
| time-critical | 31 | 15 | 15 | 15 | 15 | 15 |
| highest | 26 | 15 | 12 | 10 | 8 | 6 |
| above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| normal | 24 | 13 | 10 | 8 | 6 | 4 |
| below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| idle | 16 | 1 | 1 | 1 | 1 | 1 |

Figure 22: Windows Thread Priorities

- When a thread's time quantum runs out, that thread is interrupted. It's priority is lowered if it belongs to a variable priority class.
- When a variable-pty thread is released from a0`wait()` operation, its priority is boosted. This is useful for increasing the performance of I/O bound processes, while allowing CPU bound processes to use the spare CPU cycles.
- **User-mode scheduling** introduced in Windows 7 allows user applications to create and manage threads independently of the kernel. Microsoft's **ConcRT** runtime library for C++ allows user-mode scheduling, user apps to create and manage threads, as well as methods to divide applications into parallelizable tasks.

## 16 Interprocess Communication (IPC)

- Processes may be classified as **independent** (cannot affect or be affected by any other process) or **cooperative** (can affect/be affected by other processes).
- Cooperative processes require a method for IPC. The advantages of allowing cooperative processes are
  - Information sharing between users

  - Computation speedup from task-level parallelism
  - Modularity
  - Convenience due to multitasking

- The models of IPC are **shared memory** model and **message passing** model. **Pipes** are a model of IPC used in very early UNIX systems, which are not used widely today but are simple to study.

## 16.1 Shared Memory Model

- Processes exchange information by writing to and reading from a common memory area dedicated for this task.
- The OS normally prevents processes from writing into each other's dedicated memory, but for shared memory IPC this restriction is temporarily removed, upon agreement from both processes.
- The processes (not OS) control the format of data that is to be shared as well any coherency issues that arise.

### 16.1.1 Producer-Consumer Problem

- The producer process generates information that is to be read by the consumer process.
- The solution to this system involves a shared buffer that the producer fills and the consumer empties. The producer may produce an item while the consumer consumes another item in the buffer.
- The producer and consumer must be synchronized to prevent the consumer from consuming a not-yet generated process.
- The following variables are shared by the producer and consumer processes.

```c
#define BUFFER_SIZE 10

typedef struct{
    ...
}item;
item buffer[BUFFER_SIZE]; //Buffer array
int in = 0, out = 0;       //Buffer Pointers
```

- An **unbounded buffer** has no size restrictions, thus the producer can produce items indefinitely, but the consumer must wait if the buffer becomes empty at any point.
- A **bounded buffer** has a fixed and finite size, hence the consumer must wait if the buffer becomes empty and the producer must wait if the buffer is full.
- The buffer is implemented as a circular queue with two pointers `in` and `out` at the head and tail respectively. If `in == out` the queue is empty, and if `(in + 1)%BUFFER_SIZE == out` the queue is full.

```c
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;
}
```

Producer Process

```c
item next_consumed;

while(true)
 {
      while(in == out)
          ; //Do nothing
      next_consumed = buffer[out];
      out = (out+1)%BUFFER_SIZE;
 }
```

Consumer Process

## 16.2 Message Passing

- Allows processes to communicate without sharing a common address space, useful on distributed systems where processes may be connected only by some network.
- Every Message Passing interface must provide a `send(message)` and a `receive(message)` function.
- Messages may have a constant size (easy to implement, hard to write programs using), or a variable size (hard to implement, easy to write programs for).

### 16.2.1 Communication

- **Direct communication** means that processes refer to each other explicitly when sending messages. The processes must know each other's identity if they want to communicate
- In this scheme, a link is established between every pair of processes that want to communicate. A link is associated withe exactly two processes, and there is only one link between each pair of processes.
- The direct message passing may be *symmetric* (both sender and receiver name each other explicitly while sending/receiving messages) or *asymmetric* (only sender needs to name the receiver, receiver need not name the sender).
- Direct communication is not desirable as if a process ID changes then all the message passing calls associated with that process must be changed, which is tedious.
- In **indirect communication**, messages are sent to and received from *mailboxes* or *ports*.
- A mailbox may be viewed as an abstract object into which messages can be deposited and from which messages can be retrieved. Mailboxes have their identification numbers (POSIX defines a mailbox ID), and two processes can communicate only if they have a shared mailbox.
- If process $P_1$ sends a message to mailbox $M$, and processes $P_2$ and $P_3$ execute a `receive()` call from mailbox $M$ at the same time, this leads to concurrency issues. These are resolved using any one of the following
  - Allow one link between 2 fixed processes only
  - Allow only one process to execute a `receive()` call at a time
  - Allow the system to use an algorithm to identify the next receiver (such as a round robin system)
- A mailbox may be owned by a process or the OS. If the process owns the mailbox, then it can only receive from that box, while other processes can only send to that mailbox.
- If the OS owns the mailbox, then it must provide a means to create, delete mailboxes and a message passing interface.

### 16.2.2 Synchronization

- **Blocking send**: The process that sends is blocked until the message currently being sent is received and read by the receiver or the receiving mailbox.
- **Non-blocking Send**: Sending process can send a message and resume operation.
- **Blocking receive**: The receiver process blocks itself until a message arrives.
- **Non-blocking receive**: The receiver retrieves either a valid message or a NULL message.
- When both send and receive are blocking, there is a **rendezvous** between the two processes. This makes the solution to the producer-consumer problem trivial.

### 16.2.3 Buffering

- All messages are stored in a temporary queue as they travel from sender to receiver. This queue can be implemented as:
  - **Zero-capacity** buffer: The link cannot have any messages waiting in it, hence the sender must block itself until the current message is received.
  - **Bounded** buffer: The sender can continue to work after it sends the message, but if the queue is full then the sender must block itself until the queue has space again.
  - **Unbounded** buffer: The sender is never required to block itself, it can send an infinite amount of messages and resume operation.

```
message next_produced;
while(true)
{
     //Produce a message
     next_produced = produce_message();
     send(next_produced);
}
```

Sender Process

```
message next_consumed;
while(true)
 {
      receive(next_consumed);
      /* Consume this message */
      consume(next_consumed);
 }
```

Receiver Process

## 16.3 Pipes

- Pipes are the simplest form of IPC, found in the earlier UNIX systems.
- The considerations while implementing pipes are whether
  - The pipe allows bidirectional communication?
  - If it does, is it half duplex or full duplex?
  - Must a relationship (eg: parent-child) exist between the processes?
  - Can the processes be on different machines (comm. over a network) or must they be on the same machine?

### 16.3.1 Ordinary Pipes

- Ordinary pipes are unidirectional, allowing only one-way communication.
- The producer writes to the **write-end** of the pipe, while the consumer reads from the **read-end** of the pipe.
- Ordinary pipes in UNIX are constructed using the `pipe(int fd[])` function, where the array `fd` holds the file descriptors of the read end (index 0) and write end (index 1).
- UNIX treats pipes as a special type of file, hence pipes can be accessed using the `read()` and `write()` system calls.
- Ordinary pipes require a parent-child relationship between the processes, hence the processes must be on the same machine
- Once the processes finish communicating and terminate, the named pipe ceases to exist.

### 16.3.2 Named Pipes

- These allow bidirectional communication, and no parent-child hierarchy is needed between the communicating processes.
- Several processes can use a single named pipe, and a named pipe typically has several writers.
- The named pipe is retained in the memory even after all the communicating processes terminate, and it stays that way until it is explicitly deleted.

- UNIX named pipes allow only half duplex communication, and all processes must be on the same machine. They are referred to as FIFOs, they are created by the `mkfifo()` system call, and accessed using the `open()`, `read()`, `write()` and `close()` system calls.
- Windows named pipes allow full duplex communication, inter machine communication and transfer of both byte- and message-oriented data.