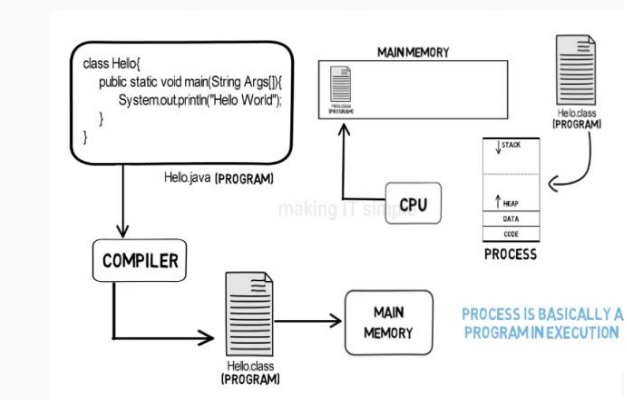
UNIT-2

**Process Concept:** Process scheduling, Operations on processes, Inter-process communication,
**Multithreaded Programming:** Multithreading models, Thread libraries, Threading issues.
**Process Scheduling:** Basic concepts, Scheduling criteria, Scheduling algorithms, Multiple processor scheduling

## Process

- A process is an instance of a program in execution.
- Batch systems work in terms of **"jobs".**
    - Many modern process concepts are still expressed in terms of jobs, (ex. job scheduling), and the two terms are often used interchangeably.




## Relation between process and Program

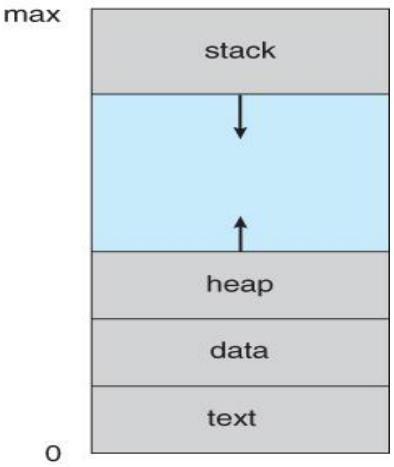
- When the code is not in execution then it is called as **Program**
- When the program is in execution then it called as **Process**

## Difference between the Process and Program

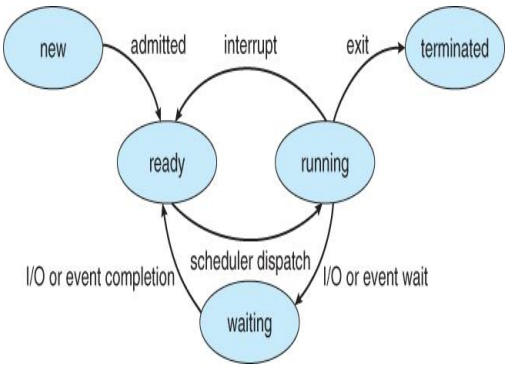
- Process is more than a program code. Process is an active entity as oppose to program which consider to be a passive entity
- Program is an algorithm expressed in some suitable notations
- Ex: Programming Language
- **Note: Process is the unit of work in a system**

## The Process

- Process memory is divided into four sections
    - The **text section** comprises the compiled program code, read in from non-volatile storage when the program is launched.
    - The **data section** stores global and static variables, allocated and initialized prior to executing main.
    - The **heap** is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
    - The **stack** is used for local variables. Space on the stack is reserved for local variables when they are declared and the space is freed up when the variables go out of scope.




**Note :** that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.




## Process State

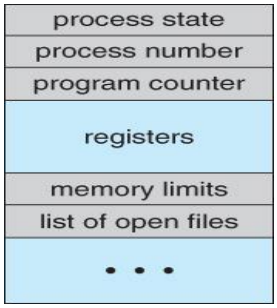
Processes may be in one of 5 states,

- **New** - The process is in the stage of being created.
- **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
- **Running** - The CPU is working on this process's instructions.
- **Waiting** - The process cannot run at the moment,because it is waiting for some resource to become available or for some event to occur.

- For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
- **Terminated -** The process has completed.

**Process Control Block**

- Process is an operating system is represented by a data structure called **Process Contol Block** or **Process Descriptor**
- For each process there is a Process Control Block, PCB, which stores the following (types of) **process-specific information**, (Specific details may vary from system to system.)




Process control block (PCB)

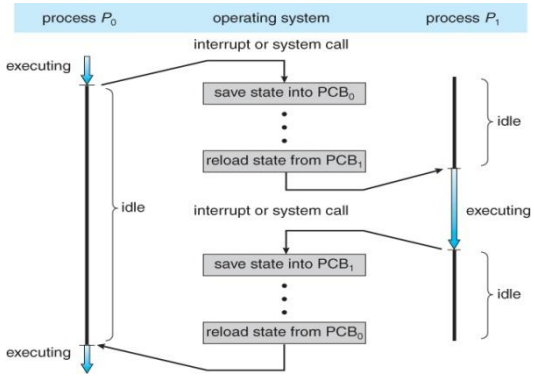
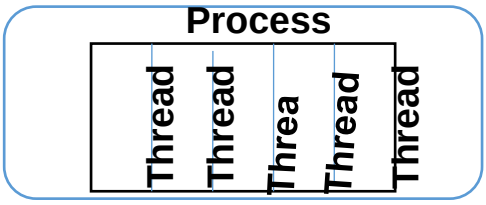



Diagram showing CPU switch from process to process

- **Process State** - Running, waiting, etc., as discussed above.
- **Process ID**, and parent process ID.
- **CPU registers and Program Counter** - These need to be saved and restored when swapping processes in and out of the CPU.
- **CPU-Scheduling information** - Such as priority information and pointers to scheduling queues.
- **Memory-Management information** - E.g. page tables or segment tables.
- **Accounting information** - user and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information** - Devices allocated, open file tables, etc.




**Thread:** A thread is the unit of execution within a process. A process can have anywhere from just one thread to many threads

**Process Scheduling:**

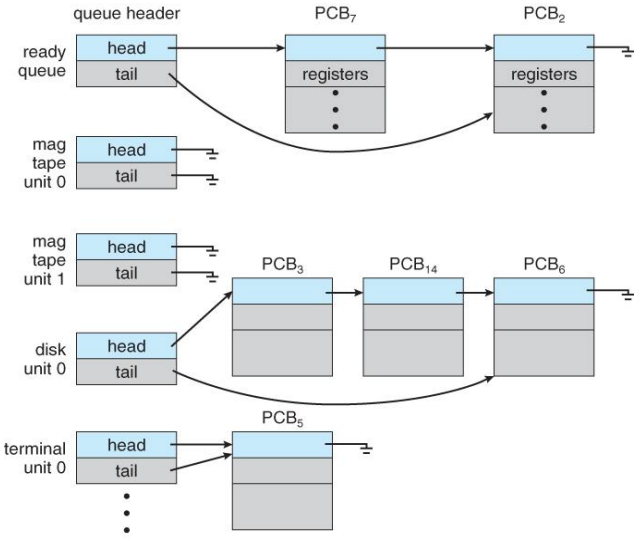
The act of determining which process is in the **ready** state, and should be moved to the **running** state is known as **Process Scheduling**.

**Process Scheduling Objectives**

- The two main objectives of the process scheduling system are
  - To keep the CPU busy at all times
  - To deliver "acceptable" response times for all programs, particularly for interactive ones.
- The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU.

**Scheduling Queues**

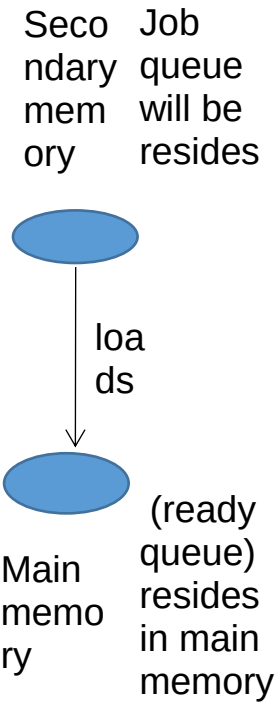
- All processes, upon entering into the system, are stored in the **Job Queue**.
- Processes in the Ready state are placed in the **Ready Queue**.
- Processes waiting for a device to become available are placed in **Device Queues**. There are unique device queues available for each I/O device.
- A new process is initially put in the **Ready queue**. It waits in the ready queue until it is selected for execution(or dispatched).



Once the process is assigned to the CPU and is executing, one of the following several events can occur:

- The process could issue an I/O request, and then be placed in the I/O queue.
- The process could create a new sub process and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

**The ready queue and various I/O device queues**




There are three types of schedulers available:

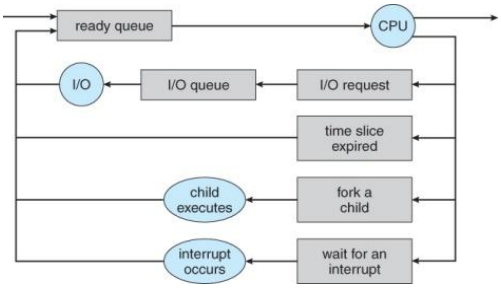
- Long Term Scheduler
- Short Term Scheduler
- Medium Term Scheduler

**Long Term or job scheduler:**

- It brings the new process to the 'Ready State'.
- It controls ***Degree of Multi-programming***, i.e., number of process present in ready state at any point of time.
- It is important that the long-term scheduler make a careful selection of both IO and CPU bound process.
- IO bound tasks are which use much of their time in input and output operations while CPU bound processes are which spend their time on CPU.
- The job scheduler increases efficiency by maintaining a balance between the two.

**Short term or CPU scheduler:**

- It is responsible for selecting one process from ready state for scheduling it on the running state.
  - **Note:** Short-term scheduler only selects the process to schedule it doesn't load the process on running. Here is when all the scheduling algorithms are used. The CPU scheduler is responsible for ensuring there is no starvation owing to high burst time processes.



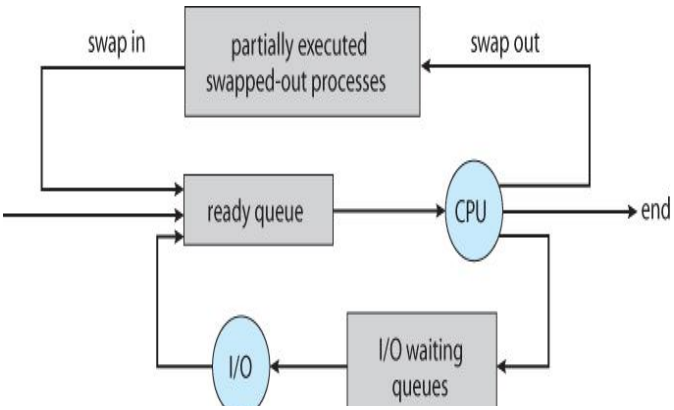

**Queueing-diagram representation of process scheduling**

**Dispatcher** is responsible for loading the process selected by Short-term scheduler on the CPU (Ready to Running State) Context switching is done by dispatcher only.

- A dispatcher does the following:
  - Switching context.
  - Switching to user mode.
  - Jumping to the proper location in the newly loaded program.

**Medium-term scheduler:**

- It is responsible for suspending and resuming the process.
- It mainly does swapping (moving processes from main memory to disk and vice versa).
- 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.
- It is helpful in maintaining a perfect balance between the I/O bound and the CPU bound. It reduces the degree of multiprogramming.




**Addition of a medium-term scheduling to the queueing diagram**

- When system loads get high, this scheduler will swap one or more processes out of the ready queue system for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system.

Context Switch

- Whenever an interrupt arrives, the CPU must do a **state-save** of the currently running process, then switch into kernel mode to handle the interrupt, and then do a **state-restore** of the interrupted process.
- Similarly, a **context switch** occurs when the time slice for one process has expired and a new process is to be loaded from the ready queue. This will be instigated by a timer interrupt, which will then cause the current process's state to be saved and the new process's state to be restored.
- Saving and restoring states involves saving and restoring all of the registers and program counter(s), as well as the **process control blocks** described above.
- Context switching happens VERY VERY frequently, and the overhead of doing the switching is just lost CPU time, so context switches ( state saves & restores ) need to be as fast as possible. Some hardware has special provisions for speeding this up, such as a single machine instruction for saving or restoring all registers at once.

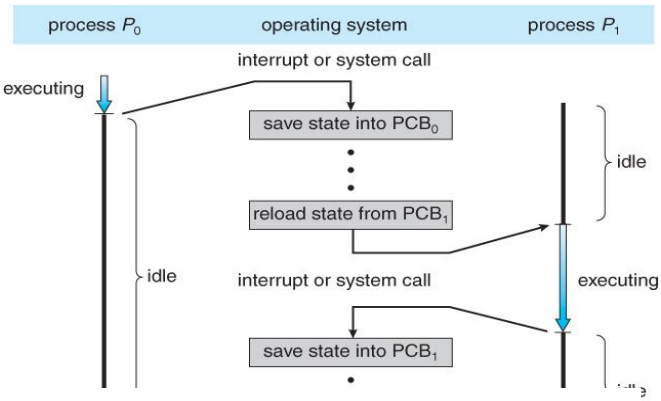


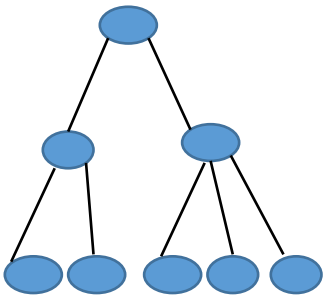

**Diagram showing CPU switch from process to process**

Operations on Processes

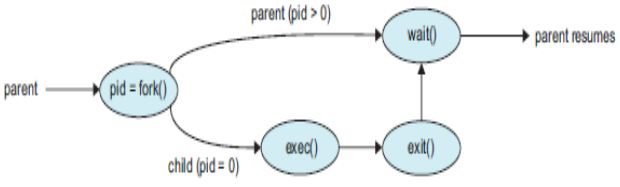
- Process Creation
- Process Termination

Process Creation

- A process may create several new processes, via a create-process system call, during the course of execution.
- The **creating process** is called a **parent process** and the **new processes** are called the **children of that process**
- Each of these new processes may in turn creates other processes forming a tree of processes



- **How Resource sharing will be done between parent and children**
  - Parent process can share all resource with children
  - Parent can share some resource with children
  - Parent never share any resource with children
- **When a process creates a new process, two possibilities exist in term of execution:**
  - 1.The parent continues to execute concurrently with its children.
  - 2.The parent waits until some or all of its children terminated
- **There are also two possibilities in terms of the address space of the new process**
  - 1.The child process is a duplicate of the parent process(it has the same program and data as the parent)
  - 2.The child process had a new program loaded into it.
- **Unix examples:**
  - fork() system call creates a new process
  - exec system call replaces newly created process with new process




**Process creation using the fork() system call.**

Process Termination

- A process terminates when its finishes executing its final statement and asks the operating system to delete it by the exit() system call.
- At that point, the process may return a status value(typically an integer) to its parent process(via the wait() system call).
- All the resources of the process-including physical and virtual memory open files, and I/O buffers are deallocated by the operating system.

Termination can occur in other circumstances as well:

1. A process can cause the termination of another process via an appropriate system call
2. Usually such a system call can be invoked only by the parent of the process that is to be terminated
3. Otherwise user could arbitrarily kill each other's jobs

Interprocess Communication

1. Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
2. A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.
3. A process is ***cooperating*** *if it can affect or be affected by the other processes* executing in the system.

1. There are several reasons for providing an environment that allows process cooperation:
    1. Information sharing
    2. Computation speedup.
    3. Modularity
    4. Convenience

2. Information sharing. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

3. Computation speedup. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.

4. Modularity. We may want to construct the system in a modular fashion dividing the system functions into separate processes or threads,

5. Convenience. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

6. Interprocess Communication (IPC) Mechanism
7. Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.
8. There are two fundamental models of interprocess communication:
    1. Shared memory
    2. Message passing.
9. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
10. 
11. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

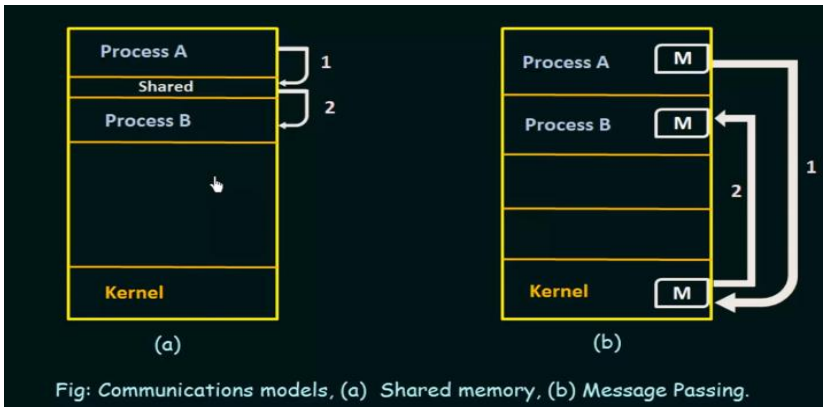
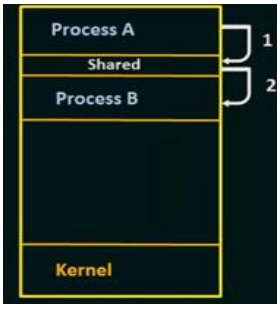


Fig: Communications models, (a) Shared memory, (b) Message Passing.

Shared-Memory Systems




- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
  - Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
  - Normally, the operating system tries to prevent one process from accessing another process's memory.

Producer-Consumer Example Using Shared Memory

- A producer process produces information that is consumed by a consumer process.

For example,

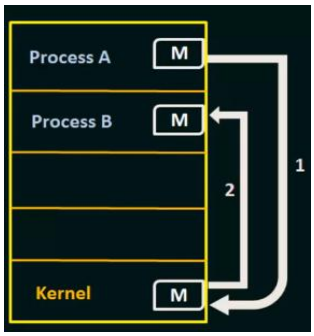
- A compiler may produce assembly code that is consumed by an assembler.
- The assembler, in turn, may produce object modules that are consumed by the loader.

Two types of buffers can be used.

- The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
- The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

## Message-Passing Systems

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.




- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- A message-passing facility provides at least two operations:
  send(message) receive(message)
- Messages sent by a process can be either fixed or variable in size.
  - Fixed-sized messages can be sent, the system-level implementation is straightforward. (But makes the task of programming more difficult)
  - Variable-sized messages require a more complex system level implementation, ( but the programming task becomes simpler)
- If processes *P and Q want to communicate, they must send messages to and* receive messages from each other:
- A *communication link must exist between* them. This link can be implemented in a variety of ways. Here are several methods for logically implementing a link and the send()/receive() operations

Issues related to each of these

Naming

- Direct communication
- indirect communication

Synchronization

- Blocked Sender / Blocked Receiver
- Non Blocked Sender /Non Blocked Receiver

**Buffering**

- The size of the buffer may **0 (ZERO)** capacity
- The size of the buffer may be **finite** capacity
- The size of the buffer is **infinite** capacity

**Direct Communication**

Direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication.


- **A communication link in this scheme has the following properties:**
- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

**Indirect Communication**

The messages are sent to and received from ***mailboxes, or ports***

- send(A, message)—Send a message to mailbox A.
- receive(A, message)—Receive a message from mailbox A.

**In this scheme, a communication link has the following properties:**

- *A link is established between a pair of processes only if both members of the pair have a shared mailbox.*
- *A link may be associated with more than two processes.*
- *Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.*.

**Synchronization**

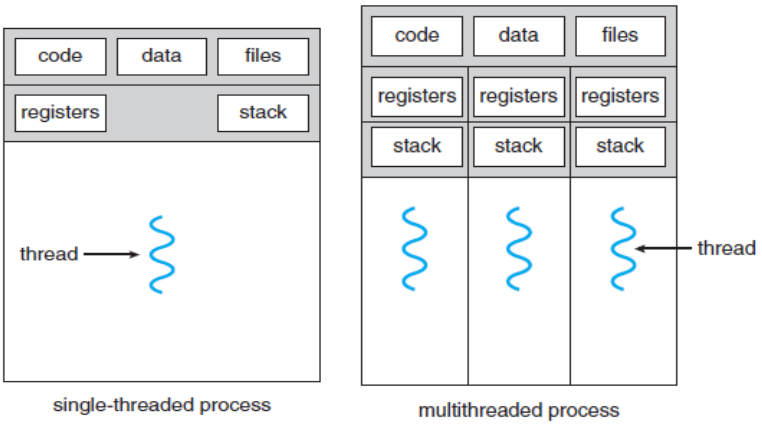
- Message passing may be either **blocking or nonblocking—** also known as **synchronous and asynchronous**
  1. **Blocking send.** The sending process is blocked until the message is received by the receiving process or by the mailbox.
  2. **Blocking receive.** The receiver blocks until a message is available.
  3. **Nonblocking send.** The sending process sends the message and resumes operation.
  4. **Nonblocking receive.** The receiver retrieves either a valid message or a null.

**Buffering**

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:
  - **Zero capacity. :**The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
  - **Bounded capacity.** The queue has finite length *n; thus, at most n messages* can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
  - **Unbounded capacity.** The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

- Definition of Thread

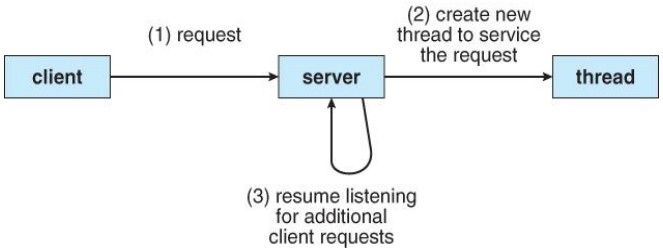
1. A thread is the unit of execution within a process. (Or)
2. *A **thread** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID. )*
3. Threads are also known as Lightweight processes.
4. A traditional (or ***heavyweight) process*** has a single thread of control.
5. If a process has multiple threads of control, it can perform more than one task at a time.
6. Most software applications that run on modern computers are multithreaded.
7. An application typically is implemented as a separate process with several threads of control




*Example:* word processor (MS Word)

The MS-Word process could involve many threads:

1. Interaction with the keyboard § Display of characters on the display page
2. Regularly saving file to disk
3. Controlling spelling and grammar Etc.
4. All these threads would share the same document




**Multithreaded server architecture**

Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request.

Benefits of Multi-threading:

There are four major categories of benefits of 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 – Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switches threads.
- Scalability: 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 processor

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the processes is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

Types of Thread

- There are two types of threads:

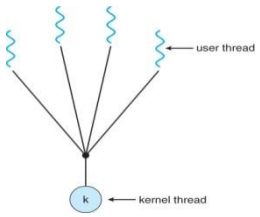
- User Threads
- Kernel Threads

- **User threads** are above the kernel and without kernel support. These are the threads that application programmers use in their programs.
  - Implementation of User Level thread is done by a thread library and is easy.
  - Example of User Level threads: Java thread, POSIX threads.
- **Kernel threads** are supported within the kernel of the OS itself. All modern OSs support kernel-level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.
  - While the Implementation of the kernel-level thread is done by the operating system and is complex.
  - Example of Kernel level threads: Window Solaris.

## Multithreading Models

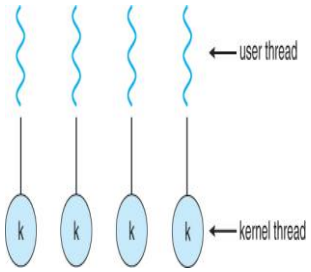
- The user threads must be mapped to kernel threads, by one of the following strategies:
  - Many to One Model
  - One to One Model
  - Many to Many Model

## Many-To-One Model



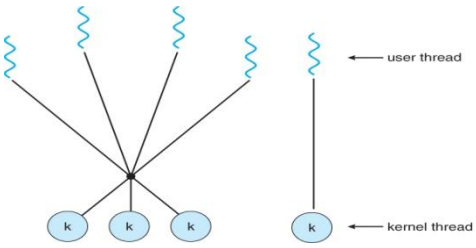

- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is very efficient.
- However, the entire process will block if a thread makes a blocking system call.
- Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs.
- The disadvantage is when we considers multiprocessor system so this cannot be considered because only one kernel is present so we can't achieve the parallelism

## One-To-One Model




- The one-to-one model creates a separate kernel thread to handle each user thread.
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- Most implementations of this model place a limit on how many threads can be created.

## Many-To-Many Model




- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.
- The following diagram shows the many-to-many threading model where 4 user level threads are multiplexing with 3 kernel level threads.
- In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine.
- This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

One popular variation of the many-to-many model is the two-tier model, which allows either many-to-many or one-to-one operation.

## Thread libraries

1. Thread libraries provide programmers with an API for creating and managing threads.

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

Threading Issues

1. There are a variety of issues to consider with multithreaded programming
    1. Semantics of fork() and exec() system calls
    2. Thread cancellation
    3. Signal handling
    4. Thread pooling
    5. Thread-specific data

Semantics of fork() and exec()

1. The fork() and exec() are the system calls. The fork() call creates a duplicate process of the process that invokes fork(). The new duplicate process is called child process and process invoking the fork() is called the parent process
2. **Let us now discuss the issue with the fork() system call.**
   Consider that a thread of the multithreaded program has invoked the fork().
    1. **ISSUE:**
        1. Here the issue is whether the new duplicate process created by fork() will duplicate all the threads of the parent process or
        2. The duplicate process would be single-threaded.
    2. **Solution:**
        1. There are two versions of fork() in some of the UNIX systems. Either the fork() can duplicate all the threads of the parent process in the child process or the fork() would only duplicate that thread from parent process that has invoked it.

**Which version of fork() must be used totally depends upon the application.**

1. Next system call i.e. **exec()** system call when invoked replaces the program along with all its threads with the program that is specified in the parameter to exec(). Typically the exec() system call is lined up after the fork() system call.
2. Here the **issue** is if the exec() system call is lined up just after the fork() system call then duplicating all the threads of parent process in the child process by fork() is useless.
3. As the exec() system call will replace the entire process with the process provided to exec() in the parameter.

**Thread Cancellation**

1. Thread cancellation involves terminating a thread before it has completed.
2. A thread that is to be cancelled is often referred to as the target thread.

Cancellation of a target thread may occur in two different scenarios:

1. 1. Asynchronous cancellation:
    1. One thread immediately terminates the target thread.
2. 2. Deferred cancellation.
    1. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.
3. The issue related to the target threads are listed below:

1. What if the resources had been allotted to the cancel target thread?
2. What if the target thread is terminated when it was updating the data, it was sharing with some other thread.

4. Here the **asynchronous cancellation** of the thread where a thread immediately cancels the target thread without checking whether it is holding any resources or not creates troublesome.
5. However, in **deferred cancellation**, the thread that indicates the target thread about the cancellation, the target thread crosschecks its flag in order to confirm that it should it be cancelled immediately or not. The thread cancellation takes place where they can be cancelled safely such points are termed as **cancellation points** by Pthreads.

## Signal Handling

1. A **signal** is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously,
2. **Synchronous signals** are delivered to the same process that performed the operation that caused the signal
   1. Examples of synchronous signal include illegal memory access and division by 0.
3. **Asynchronous Signal** are generated by an event external to a running process, that process receives the signal asynchronously.
   1. Examples of such signals include terminating a process with specific keystrokes (such as *<control><C>) and* having a timer expire.
4. Handling signals in **single-threaded programs** is straightforward: Signals are always delivered to a process.
5. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?
6. In general, the following options exist:
   1. **1. Deliver the signal to the thread to which the signal applies.**
   2. **2. Deliver the signal to every thread in the process.**
   3. **3. Deliver the signal to certain threads in the process.**
   4. **4. Assign a specific thread to receive all signals for the process.**
7. So if the signal is **synchronous** it would be delivered to the specific thread causing the generation of the signal.
8. If the signal is **asynchronous** it cannot be specified to which thread of the multithreaded program it would be delivered.
   1. If the asynchronous signal is notifying to terminate the process the signal would be delivered to all the thread of the process.
9. The **issue of an asynchronous signal** is resolved up to some extent in most of the multithreaded UNIX system.
10. Here the thread is allowed to specify which signal it can accept and which it cannot.
11. However, the Window operating system does not support the concept of the signal instead it uses asynchronous procedure call (ACP) which is similar to the asynchronous signal of the UNIX system.

## Thread Pool

1. When a user requests for a webpage to the server, the server creates a separate thread to service the request. Although the server also has some potential issues. Consider if we do not have a bound on the number of actives thread in a system and would create a new thread for every new request then it would finally result in exhaustion of system resources.
2. The idea is to create a finite amount of threads when the process starts. This collection of threads is referred to as the **thread pool.**
3. **A thread pool is a group of threads that have been pre-created and are available to do work as needed**
   1. Threads may be created when the process starts
   2. A thread may be kept in a queue until it is needed
   3. After a thread finishes, it is placed back into a queue until it is needed again
   4. Avoids the extra time needed to spawn new threads when they're needed
4. **In applications where threads are repeatedly being created/destroyed thread pools might provide a performance benefit**

**Example:** A server that spawns a new thread each time a client connects to the system and discards that thread when the client disconnects

**Advantages of thread pools:**

1. Typically faster to service a request with an existing thread than create a new thread (performance benefit)
2. Bounds the number of threads in a process

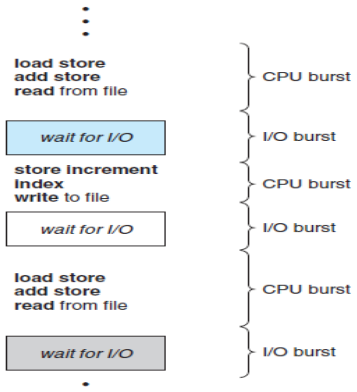
- The only threads available are those in the thread pool
- If the thread pool is empty, then the process must wait for a thread to re-enter the pool before it can assign work to a thread
- Without a bound on the number of threads in a process, it is possible for a process to create so many threads that all of the system resources are exhausted

**Thread Specific data**

- We all are aware of the fact that the threads belonging to the same process share the data of that process. Here the issue is what if each particular thread of the process needs its own copy of data. So the specific data associated with the specific thread is referred to as **thread-specific data.**
  1. Consider a transaction processing system, here we can process each transaction in a different thread. To determine each transaction uniquely we will associate a unique identifier with it. Which will help the system to identify each transaction uniquely.
  2. As we are servicing each transaction in a separate thread.
- So we can use thread-specific data to associate each thread to a specific transaction and its unique id. Thread libraries such as Win32, Pthreads and Java support to thread-specific data.
- So these are threading issues that occur in the multithreaded programming environment. We have also seen how these issues can be resolved.

**Scheduling**

1. **Scheduling** of this kind is a fundamental operating-system function.
2. Almost all computer resources are scheduled before use.
3. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.
4. **Scheduling of processes/work is done to finish the work on time.**
5. **CPU–I/O Burst Cycle**
6. The success of CPU scheduling depends on an observed property of processes:
7. Process execution consists of a **cycle** of **CPU execution and I/O wait.**
8. **Processes** alternate between these two states.
   1. Process execution begins with a **CPU burst.**
   2. That is followed by an **I/O burst.**
9. Another CPU burst, then another I/O burst, and so on.
10. Eventually, the final CPU burst ends with a system request to terminate execution



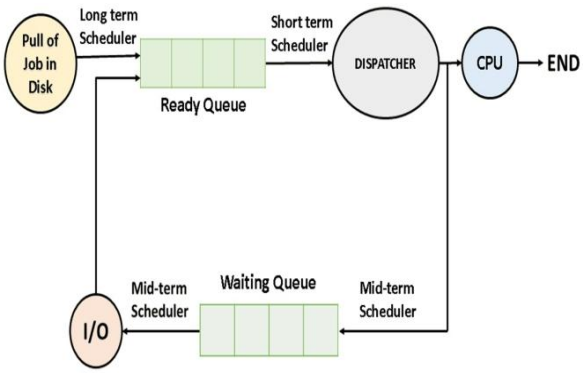

**Alternating sequence of CPU and I/O bursts.**

**Process scheduling**

The act of determining which process is in the **ready** state, and should be moved to the **running** state is known as **Process Scheduling**.

**CPU Scheduler**

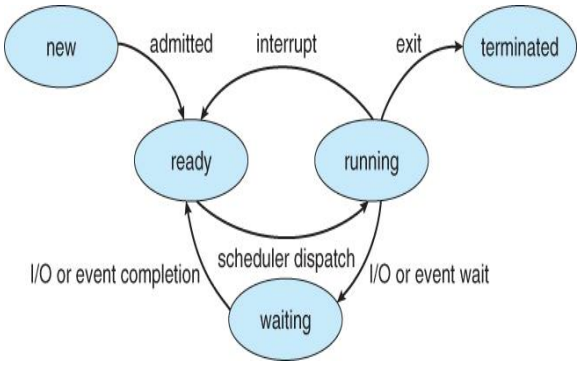
1. **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them



Queue may be ordered in various ways

CPU scheduling decisions may take place when a process:

1. When a process switches from the **running state** to the **waiting state** (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)
2. When a process switches from the **running state** to the **ready state** (for example, when an interrupt occurs)
3. When a process switches from the **waiting state** to the **ready state** (for example, at completion of I/O)
4. When a process terminates




For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **Non-pre-emptive or Cooperative Otherwise,** it is **Pre-emptive.**

**Preemptive**

Preemptive scheduling is used when a process switches from **running state** to **ready state** or from **waiting state** to **ready state**.

The resources (mainly CPU cycles)are allocated to the process for the limited amount of time and then is taken away ,and the process is again placed back in the ready queue if that process still has CPU burst time remaining. The process stays in ready queue till it gets next chance to execute.

**Algorithms based on preemptive scheduling are**:

1. Round Robin(RR)
2. Shortest Remaining First(SRTF),
3. Priority(Preemptive version).etc

**Non-Preemptive Scheduling**

1. Non-Preemptive scheduling is used when a process **terminates**, or a process switches from **running** to **waiting** state.
2. In this scheduling, once the resources(CPU Cycles) is allocated to a process, the process hold the CPU till it gets terminated or it reaches a waiting state.
3. In case of non-preemptive scheduling does not interrupts a process running CPU in middle of the executing .
4. Instead it waits still the process completes its CPU burst time and then it can allocates the CPU to another process

**Algorithms based on Non-Preemptive scheduling are:**

1. Shortest Job First(SJF basically non Preemptive)
2. Priority (non-preemptive version)etc
3. First Come First Serve

**Dispatcher**

1. Another component involved in the CPU-scheduling function is the **dispatcher.**
2. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
3. This function involves the following:
    1. Switching context
    2. Switching to user mode
    3. Jumping to the proper location in the user program to restart that program
4. The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency.**

**Scheduling Criteria**

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best.

**The criteria include the following:**

**CPU utilization:**

Keep the CPU as busy as possible

**Throughput**:

Number of processes that complete their execution per time unit

**Turnaround time:**

Amount of time to execute a particular process

**Waiting time:**

Amount of time a process has been waiting in the ready queue

**Response time**

Amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

**Objectives of Process Scheduling Algorithms**

- Maximum CPU utilization
- Fair allocation of CPU
- Maximum throughput
- Minimum turnaround time
- Minimum waiting time
- Minimum response time

**Below are different times with respect to a process.**

1. **Arrival Time:** Time at which the process arrives in the ready queue.
2. **Completion Time:** Time at which process completes its execution.
3. **Burst Time:** Time required by a process for CPU execution.
4. **Turn Around Time:** Time Difference between completion time and arrival time.

   **Turn Around Time = Completion Time – Arrival Time**
5. **Waiting Time(W.T):** Time Difference between turn around time and burst time.

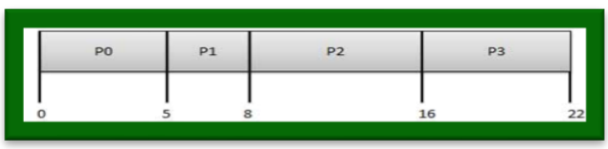
   **Waiting Time = Turn Around Time – Burst Time**

**Scheduling Algorithms**

- **Different Scheduling Algorithms**
- First Come First Serve(FCFS) Scheduling
- Shortest-Job-First(SJF) Scheduling
- Priority Scheduling
- Round Robin(RR) Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling
- Shortest Remaining Time First (SRTF)
- Longest Remaining Time First (LRTF)

**Gantt chart**

A Gantt chart is a bar chart that provides a visual view of tasks scheduled over time.




**First-Come, First-Served Scheduling**

1. By far the simplest CPU-scheduling algorithm.
2. In this scheduling, process that requests the CPU first is allocated the CPU first.
3. The implementation of the FCFS policy is managed with a FIFO queue.

4. When a process enters the ready queue, its PCB is linked onto the tail of the queue.
5. When the CPU is free, it is allocated to the process at the head of the queue.
6. The running process is then removed from the queue.


**Example:**

Consider the following set of **processes** that arrive at time 0, with the length of the **CPU burst** given in milliseconds

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |


If the processes arrive in the order *P1, P2, P3, and are served in FCFS order,* we get the result shown in the following **Gantt chart**

- *Waiting time for P1 = 0 milliseconds*
- *Waiting time for P2 = 24 milliseconds*
- *Waiting time for P3 = 27 milliseconds*

*Thus, the average waiting time is (0* + 24 + 27)/3 = 17 milliseconds.

**If the processes arrive in the order *P2, P3, P1,* however, the results will be as shown in the following Gantt chart:**


- *Waiting time for P1 = 6 milliseconds*
- *Waiting time for P2 = 0 milliseconds*
- *Waiting time for P3 = 3 milliseconds*

*Thus, the average waiting time is (6* + 0 + 3)/3 = 3 milliseconds.

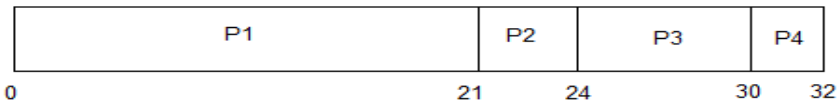
This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

**The FCFS Scheduling algorithm is Non-Preemptive**

**Example: find** the average waiting time using the FCFS scheduling algorithm.

| PROCESS | BURST TIME |
|---|---|
| P1 | 21 |
| P2 | 3 |
| P3 | 6 |
| P4 | 2 |

The average waiting time will be = ( 0 + 21 + 24 + 30 )/4 = 18.75 ms

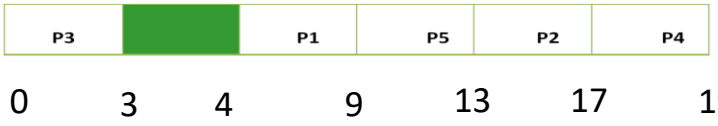



This is the GANTT chart for the above processes

**Example:2**

| Process ID | Arrival Time | Burst Time |
|---|---|---|
| P1 | 4 | 5 |
| P2 | 6 | 4 |
| P3 | 0 | 3 |
| P4 | 6 | 2 |
| P5 | 5 | 4 |

**The shaded box represents the idle time of CPU**




## Gantt chart

| Process ID | Arrival Time | Burst Time | Completion Time | Turnaround Time | Waiting Time |
|---|---|---|---|---|---|
| P1 | 4 | 5 | 9 | 9-4=5 | 5-5=0 |
| P2 | 6 | 4 | 17 | 17-6=11 | 11-4=7 |
| P3 | 0 | 3 | 3 | 3-0=3 | 3-3=0 |
| P4 | 6 | 2 | 19 | 19-6=13 | 13-2=11 |
| P5 | 5 | 4 | 13 | 13-5=8 | 8-4=4 |

Calculation of the average **waiting time** and **average Turnaround** time, if FCFS Scheduling Algorithm is followed

***Waiting Time = Turn Around Time – Burst Time***

***Turn Around Time = Completion Time – Arrival Time***

Average Turn Around time = (5+11+3+13+8)/5
= 40/5
= 8 ms

Average Waiting Time = (0+7+0+11+4)/5
= 22/5
= 4.4 ms

Convoy Effect

1. FCFS algorithm is non-pre-emptive in nature, that is, once CPU time has been allocated to a process, other processes can get CPU time only after the current process has finished. This property of FCFS scheduling leads to the situation called Convoy Effect.
2. Convoy Effect is phenomenon associated with the First Come First Serve (FCFS) algorithm, in which the whole Operating System slows down due to few slow processes.
3. Suppose there is one CPU intensive (large burst time) process in the ready queue, and several other processes with relatively less burst times but are Input/Output (I/O) bound (Need I/O operations frequently)

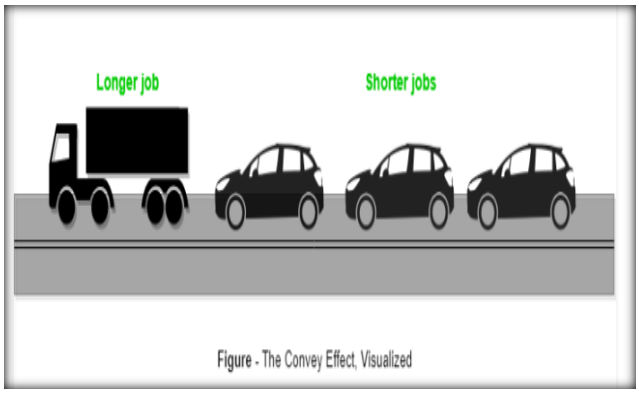


Figure - The Convey Effect, Visualized

Steps are as following below:

1. The I/O bound processes are first allocated CPU time. As they are less CPU intensive, they quickly get executed and goto I/O queues.
2. Now, the CPU intensive process is allocated CPU time. As its burst time is high, it takes time to complete.
3. While the CPU intensive process is being executed, the I/O bound processes complete their I/O operations and are moved back to ready queue.
4. However, the I/O bound processes are made to wait as the CPU intensive process still hasn't finished. This leads to I/O devices being idle.
5. When the CPU intensive process gets over, it is sent to the I/O queue so that it can access an I/O device.
6. Meanwhile, the I/O bound processes get their required CPU time and move back to I/O queue.
7. However, they are made to wait because the CPU intensive process is still accessing an I/O device. As a result, the CPU is sitting idle now.

Hence in Convoy Effect, one slow process slows down the performance of the entire set of processes, and leads to wastage of CPU time and other devices.

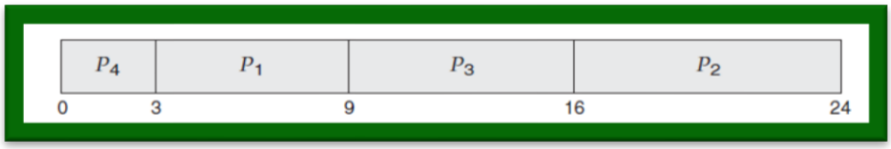
**Shortest-Job-First Scheduling**

1. A different approach to CPU scheduling is the **shortest-job-first (SJF) scheduling** algorithm.
2. This algorithm associates with each process the **length of the process's next CPU burst**.
3. When the CPU is available, it is assigned to the **process that has the smallest next CPU burst**.
4. This is the best approach to **minimize waiting time**.
5. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
6. A more appropriate term for this scheduling method would be the ***shortest-next- CPU-burst algorithm, because scheduling depends on the length of the next*** CPU burst of a process, rather than its total length.

**Example of SJF Scheduling (Non-Preemptive)**

The following set of processes, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |



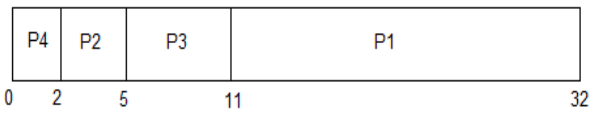

The waiting time for ***P1*** =3 ms
The waiting time for ***P2*** *=16 ms*
The waiting time for ***P3****= 9 ms*
The waiting time for ***P4*** *= 0 ms*
***The*** **average waiting time** = (3 + 16 + 9 + 0)/4 = 7ms

**By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.**

**Example 2:**

Consider the below processes available in the ready queue for execution, with **arrival time** as 0 for all and given **burst times**.

| PROCESS | BURST TIME |
|---|---|
| P1 | 21 |
| P2 | 3 |
| P3 | 6 |
| P4 | 2 |




**Gantt chart:**

Now, the average waiting time will be = ( 0 + 2 + 5 + 11)/4 = 4.5 ms

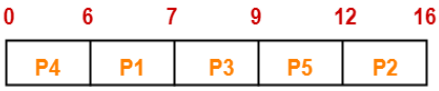
**Example: 3**

Consider the set of 5 processes whose arrival time and burst time are given below-

| Process Id | Arrival time | Burst time |
|---|---|---|
| P1 | 3 | 1 |
| P2 | 1 | 4 |
| P3 | 4 | 2 |
| P4 | 0 | 6 |
| P5 | 2 | 3 |

If the CPU scheduling policy is SJF non-preemptive, calculate the average waiting time and average turn around time.

Gantt Chart-




Gantt Chart

*Turn Around Time = Completion Time – Arrival Time*

*Waiting Time = Turn Around Time – Burst Time*

| Process Id | Completion time | Turn Around time | Waiting time |
|---|---|---|---|
| P1 | 7 | 7 – 3 = 4 | 4 – 1 = 3 |
| P2 | 16 | 16 – 1 = 15 | 15 – 4 = 11 |
| P3 | 9 | 9 – 4 = 5 | 5 – 2 = 3 |
| P4 | 6 | 6 – 0 = 6 | 6 – 6 = 0 |
| P5 | 12 | 12 – 2 = 10 | 10 – 3 = 7 |

Now,

- Average Turn Around time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit
- Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit

Problem with Non Pre-emptive SJF

- If the arrival time for processes are different, which means all the processes are not available in the ready queue at time 0, and some jobs arrive after some time, in such situation, sometimes process with short burst time have to wait for the current process's execution to finish, because in Non Pre-emptive SJF, on arrival of a process with short duration, the existing job/process's execution is not halted/stopped to execute the short job first.

Shortest Job First Scheduling (Preemptive)

1. In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with short burst time arrives, the existing process is preempted or removed from execution, and the shorter job is executed first.
2. The average waiting time for preemptive shortest job first scheduling is less than both,non preemptive SJF scheduling and FCFS scheduling.
3. The Pre-emptive SJF is also known as Shortest Remaining Time First, because at any given point of time, the job with the shortest remaining time is executed first.

Example:SJF (Preemptive)

Consider the set of 5 processes whose arrival time and burst time are given below-

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 8 |
| P2 | 1 | 1 |
| P3 | 2 | 3 |
| P4 | 3 | 2 |
| P5 | 4 | 6 |

If the CPU scheduling policy is SJF preemptive, calculate the average waiting time and average turnaround time.

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 8 |
| P2 | 1 | 1 |
| P3 | 2 | 3 |
| P4 | 3 | 2 |
| P5 | 4 | 6 |

**Gantt chart:**

| P1 | P2 | P3 | P4 | P5 | P1 |
|---|---|---|---|---|---|

0 1 2 5 7 13 20

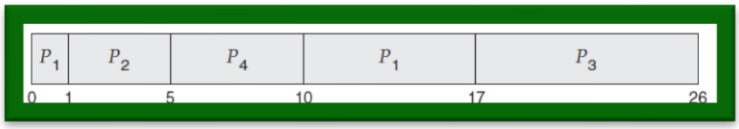
| Process | Arrival Time | Burst Time | Completion Time | Turn Around Time | Waiting Time |
|---|---|---|---|---|---|
| P1 | 0 | 8 | 20 | 20 | 12 |
| P2 | 1 | 1 | 2 | 1 | 0 |
| P3 | 2 | 3 | 5 | 3 | 0 |
| P4 | 3 | 2 | 7 | 4 | 2 |
| P5 | 4 | 6 | 13 | 9 | 3 |

**Average Turn Around Time =37/5 =7.4 ms**
**Average Waiting Time = 17/5=3.4 ms**

Example of SJF Scheduling (Preemptive)

consider the following four processes, with the length of the CPU burst given in milliseconds and processes arrive at the ready queue at the times shown

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |




| Process | Arrival | Burst Time | Completion Time | Turn Around Time | Waiting Time |
|---|---|---|---|---|---|
| P1 | 0 | 8 | 17 | 17 | 9 |
| P2 | 1 | 4 | 5 | 4 | 0 |
| P3 | 2 | 9 | 26 | 24 | 15 |
| P4 | 3 | 5 | 10 | 7 | 2 |

**Average Turnaround Time =52/4 =13 ms** **Average Waiting Time = 26/4=6.5 ms**

Priority Scheduling

1. A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
2. Equal-priority processes are scheduled in FCFS order.
3. An SJF algorithm is simply a priority algorithm where the priority (*p) is the* inverse of the (predicted) next CPU burst.
4. The larger the CPU burst, the lower the priority, and vice versa.

Note:Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion.

1. We assume that low numbers represent high priority.
2. Priority scheduling can be either preemptive or nonpreemptive.
3. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
4. A non preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

Example:

Consider the following set of processes, assumed to have arrived at time 0 in the order *P1, P2, · · ·, P5, with the length of the CPU burst* given in milliseconds:

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

**Using priority scheduling, we would schedule these processes according to the following**

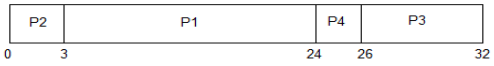

**Average Waiting Time =(6+0+16+18+1)/5**
**= 41/5**
**=8.2 ms**

Example: Priority Scheduling (Non-Preemptive)

Consider the below table for processes with their respective CPU burst times and the priorities.

| PROCESS | BURST TIME | PRIORITY |
|---|---|---|
| P1 | 21 | 2 |
| P2 | 3 | 1 |
| P3 | 6 | 4 |
| P4 | 2 | 3 |

The GANTT chart for following processes based on Priority scheduling will be,




The average waiting time will be, ( 0 + 3 + 24 + 26 )/4 = 13.25 ms

Example:
Consider the set of 5 processes whose arrival time and burst time are given below-

| Process Id | Arrival time | Burst time | Priority |
|---|---|---|---|
| P1 | 0 | 4 | 2 |
| P2 | 1 | 3 | 3 |
| P3 | 2 | 1 | 4 |
| P4 | 3 | 5 | 5 |
| P5 | 4 | 2 | 5 |

If the CPU scheduling policy is priority non-preemptive, calculate the average waiting time and average turnaround time. (Higher number represents higher priority)

**Gantt Chart-**


Gantt Chart

| Process Id | Exit time | Turn Around time | Waiting time |
|---|---|---|---|
| P1 | 4 | 4 – 0 = 4 | 4 – 4 = 0 |
| P2 | 15 | 15 – 1 = 14 | 14 – 3 = 11 |
| P3 | 12 | 12 – 2 = 10 | 10 – 1 = 9 |
| P4 | 9 | 9 – 3 = 6 | 6 – 5 = 1 |
| P5 | 11 | 11 – 4 = 7 | 7 – 2 = 5 |

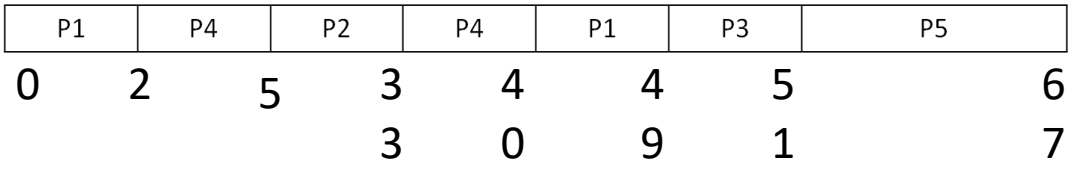
Now,

- Average Turn Around time = (4 + 14 + 10 + 6 + 7) / 5 = 41 / 5 = 8.2 unit
- Average waiting time = (0 + 11 + 9 + 1 + 5) / 5 = 26 / 5 = 5.2 unit

Example: Priority Scheduling Preemptive

Consider the set of processes with arrival time (in milliseconds), CPU burst time (in milliseconds), and priority (0 is the highest priority) shown below. None of the processes have I/O burst time.
Calculate the average waiting time (in milliseconds) of all the processes using pre-emptive priority scheduling algorithm

| Process | Arrival Time | Burst Time | Priority |
|---|---|---|---|
| $P_1$ | 0 | 11 | 2 |
| $P_2$ | 5 | 28 | 0 |
| $P_3$ | 12 | 2 | 3 |
| $P_4$ | 2 | 10 | 1 |
| $P_5$ | 9 | 16 | 4 |




Waiting Time for p1= 38 ms

Waiting Time for p2= 0 ms

Waiting Time for p3= 37 ms

Waiting Time for p4= 28 ms

Waiting Time for p5= 42 ms

Average Waiting Time =(38+0+37+28+42)/5
=145/5 =29 ms

Example:
Consider the set of 5 processes whose arrival time and burst time are given below-

| Process Id | Arrival time | Burst time | Priority |
|---|---|---|---|
| P1 | 0 | 4 | 2 |
| P2 | 1 | 3 | 3 |
| P3 | 2 | 1 | 4 |
| P4 | 3 | 5 | 5 |
| P5 | 4 | 2 | 5 |

If the CPU scheduling policy is priority preemptive, calculate the average waiting time and average turn around time. (Higher number represents higher priority)

**Gantt Chart-**


Gantt Chart

| Process Id | Exit time | Turn Around time | Waiting time |
|---|---|---|---|
| P1 | 15 | 15 – 0 = 15 | 15 – 4 = 11 |
| P2 | 12 | 12 – 1 = 11 | 11 – 3 = 8 |
| P3 | 3 | 3 – 2 = 1 | 1 – 1 = 0 |
| P4 | 8 | 8 – 3 = 5 | 5 – 5 = 0 |
| P5 | 10 | 10 – 4 = 6 | 6 – 2 = 4 |

Now,

- Average Turn Around time = (15 + 11 + 1 + 5 + 6) / 5 = 38 / 5 = 7.6 unit
- Average waiting time = (11 + 8 + 0 + 0 + 4) / 5 = 23 / 5 = 4.6 unit

### Problem with the Priority Scheduling

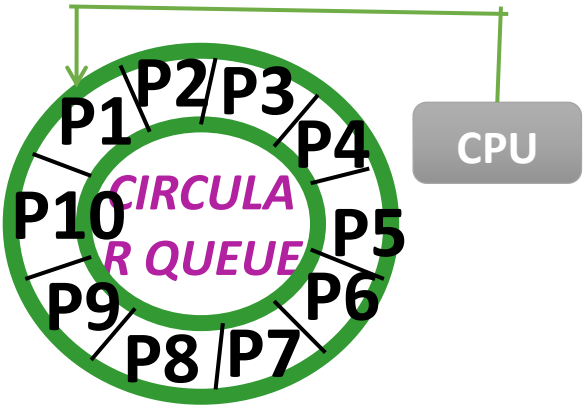
1. A major problem with priority scheduling algorithms is **indefinite blocking,** or **starvation.**
2. A process that is ready to run but waiting for the CPU can be considered blocked.
3. A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
4. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

### Solution to Problem

1. A solution to the problem of indefinite blockage of low-priority processes is **aging.** Aging involves gradually increasing the priority of processes that wait in the system for a long time.
    1. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

### Round-Robin Scheduling

1. The **round-robin (RR) scheduling algorithm is designed especially for time sharing systems**.
2. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
3. A small unit of time, called a **time quantum or time slice,** is defined**.** A **time quantum** is generally from **10 to 100 milliseconds** in length.
4. The ready queue is treated as a circular queue.
5. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time



quantum.

**Implementation RR scheduling**

1. We keep ready queue as a FIFO queue of processes.
2. New processes are added to the tail of the ready queue.
3. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

**One of two things will then happen.**


**1.** The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.

**2.** If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

**Example:**

**Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds: time quantum of 4 milliseconds**

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

Average Turn Around time= (30+7+10)/3

=47/3

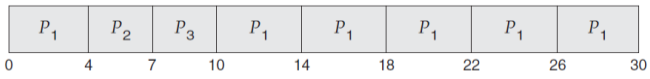
=15.66ms

Average Waiting Time = (6+4+7)/3

=5.66 ms

Solution:

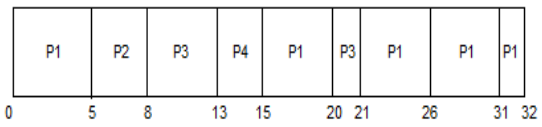
**Gantt Chart:**




| Process | Completion Time | Turnaround Time | Waiting Time |
|---|---|---|---|
| P1 | 30 | 30-0=30 | 30-24=6 |
| P2 | 7 | 7-0=7 | 7-3=4 |
| P3 | 10 | 10-0=10 | 10-3=7 |

**Example: Consider 5ms**

The GANTT chart for round robin scheduling will be,




The average waiting time will be, 11 ms.

| PROCESS | BURST TIME |
|---|---|
| P1 | 21 |
| P2 | 3 |
| P3 | 6 |
| P4 | 2 |

**Example: Consider the set of 4 processes whose arrival time and burst time are given below-**

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 5 |
| P2 | 1 | 4 |
| P3 | 2 | 2 |
| P4 | 4 | 1 |

If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time and average turnaround time.

**Ready Queue**

| P1 | P2 | P3 | P1 | P4 | P2 | P1 |
|---|---|---|---|---|---|---|

**Gantt chart**

| P1 | P2 | P3 | P1 | P4 | P2 | P1 |
|---|---|---|---|---|---|---|

0 2 4 6 8 9 11 12

| Process | Arrival Time | Burst Time | Completion Time | Turn-Around Time | Waiting Time |
|---|---|---|---|---|---|
| P1 | 0 | 5 | 12 | 12 | 7 |
| P2 | 1 | 4 | 11 | 10 | 6 |
| P3 | 2 | 2 | 6 | 4 | 2 |
| P4 | 4 | 1 | 9 | 5 | 4 |

Consider the set of 5 processes whose arrival time and burst time are given below-

| Process Id | Arrival time | Burst time |
|---|---|---|
| P1 | 0 | 5 |
| P2 | 1 | 3 |
| P3 | 2 | 1 |
| P4 | 3 | 2 |
| P5 | 4 | 3 |

If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time and average turn around time.


Gantt Chart

| Process Id | Exit time | Turn Around time | Waiting time |
|---|---|---|---|
| P1 | 13 | 13 – 0 = 13 | 13 – 5 = 8 |
| P2 | 12 | 12 – 1 = 11 | 11 – 3 = 8 |
| P3 | 5 | 5 – 2 = 3 | 3 – 1 = 2 |
| P4 | 9 | 9 – 3 = 6 | 6 – 2 = 4 |
| P5 | 14 | 14 – 4 = 10 | 10 – 3 = 7 |

Now,

- Average Turn Around time = (13 + 11 + 3 + 6 + 10) / 5 = 43 / 5 = 8.6 unit
- Average waiting time = (8 + 8 + 2 + 4 + 7) / 5 = 29 / 5 = 5.8 unit

1. ***Longest Job First (LJF):*** It is similar to SJF scheduling algorithm. But, in this scheduling algorithm, we give priority to the process having the longest burst time. This is non-preemptive in nature i.e., when any process starts executing, can't be interrupted before complete execution.
2. ***Longest Remaining Time First (LRTF):*** It is preemptive mode of LJF algorithm in which we give priority to the process having largest burst time remaining.

## Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.

For example:

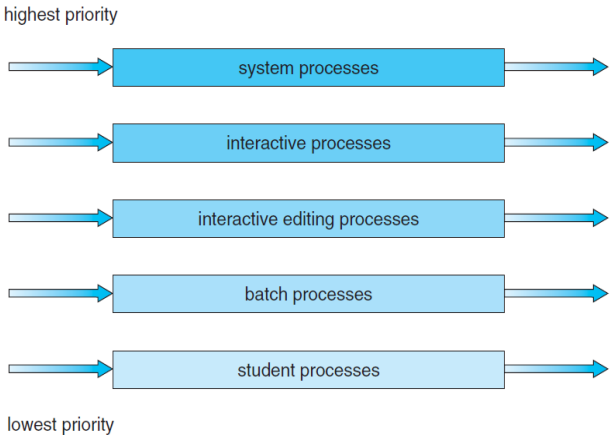
1. Processes that require a user to start them or to interact with them are called ***foreground processes***. Processes that are run independently of a user are referred to as ***background processes***.
    1. Programs and commands run as foreground processes by default. To run a process in the background, place an ampersand (&) at the end of the command name that you use to start the process.
2. Common division is made between **foreground (interactive) processes and background (batch) processes.**
3. These two types of processes have different response-time requirements different scheduling needs.
4. In addition, foreground processes may have priority (externally defined) over background processes.
5. According to the priority of process, processes are placed in the different queues. Generally high priority process are placed in the top level queue. Only after completion of processes from top level queue, lower level queued processes are scheduled. It can suffer from starvation.

6. An example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

    1. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as **fixed-priority pre-emptive scheduling**

. For example, the foreground queue may have absolute priority over the background queue. Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.




Another possibility is to **time-slice among the queues**.

Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

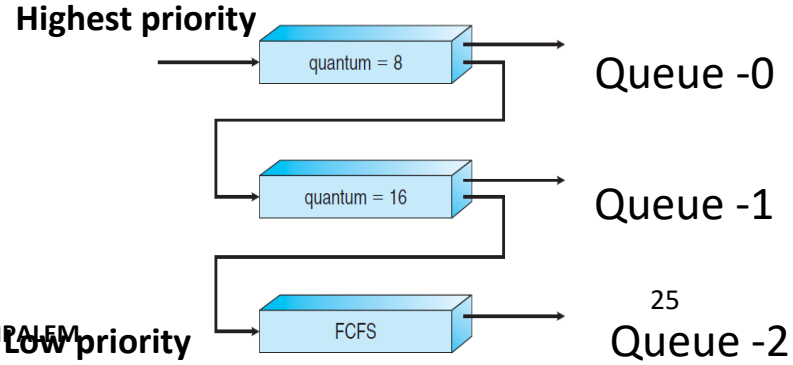
**Multilevel Feedback Queue Scheduling**

1. The **multilevel feedback queue scheduling algorithm, allows** a process to move between queues.
2. The idea is to separate processes according to the characteristics of their CPU bursts.
3. If a process **uses too much CPU time**, it will be **moved to a lower-priority queue.**
4. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
5. In addition, **a process that waits too long in a lower-priority queue** may be moved to a **higher-priority queue.**
6. **This form of aging prevents starvation.**

Consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher priority queue



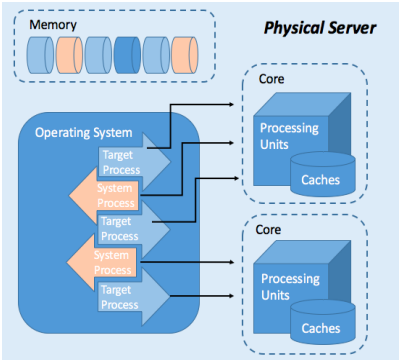
• The method used to determine when to demote a process to a lower priority queue
• The method used to determine which queue a process will enter when that process needs service

### Multiple-Processor Scheduling

When multiple processors are available, then the scheduling gets more complicated, because now there is more than one CPU which must be kept busy and in effective use at all times.

***Load sharing*** revolves around balancing the load between multiple processors.

1. Multi-processor systems may be **heterogeneous,** (different kinds of CPUs), or **homogenous,** (all the same kind of CPU). Even in the latter case there may be special scheduling constraints, such as devices which are connected via a private bus to only one of the CPUs.
2. **Approaches to Multiple-Processor Scheduling**
3. One approach to multi-processor scheduling is ***asymmetric multiprocessing,*** in which one processor is the master, controlling all activities and running all kernel code, while the other runs only user code. This approach is relatively simple, as there is no need to share critical system data.
4. Another approach is ***symmetric multiprocessing, SMP,*** where each processor schedules its own jobs, either from a common ready queue or from separate ready queues for each processor.
5. Virtually all modern OS's support SMP, including XP, Win 2000, Solaris, Linux, and Mac OSX.




### Processor Affinity

High cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity—that is, a process** has an affinity for the processor on which it is currently running.

**Processor affinity takes several forms.**

1. When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so that situation known as **soft affinity.**
2. Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors.
3. In contrast, some systems provide system calls that support **hard affinity,** thereby allowing a process to specify a subset of processors on which it may run.
4. Many systems provide both soft and hard affinity.
5. For example, Linux implements soft affinity, but it also provides the sched setaffinity() system call, which supports hard affinity.

### Load Balancing

On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.

**Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system.

1. ***Push migration*** involves a separate process that runs periodically, (ex. every 200 milliseconds), and moves processes from heavily loaded processors onto less loaded ones.
2. ***Pull migration*** involves idle processors taking processes from the ready queues of other processors.
3. Push and pull migration are not mutually exclusive.

**Note** that moving processes from processor to processor to achieve load balancing works against the principle of processor affinity, and if not carefully managed, the savings gained by balancing the system can be lost in rebuilding caches

### Multicore Processors

Traditionally, SMP systems have allowed several threads to run concurrently by providing multiple physical processors. However, a recent practice in computer hardware has been to place multiple processor cores on the same physical chip, resulting in a **multicore processor.**By assigning multiple kernel threads to a single processor, memory stall can be avoided (or reduced) by running one thread on the processor while the other thread waits for memory.