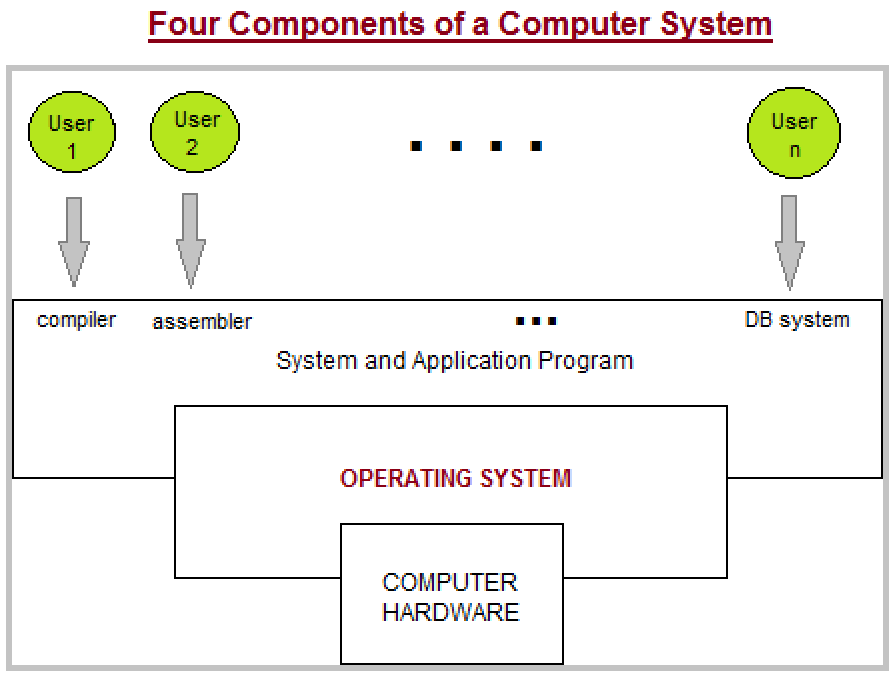
Introduction to Operating Systems

operating system

- operating system is the resource manager i.e. it can manage the resource of a computer system internally.
- The resources are processor, memory, files, and I/O devices.
- **In simple terms, an operating system is the interface between the user and the machine.**



Operating System Management Tasks

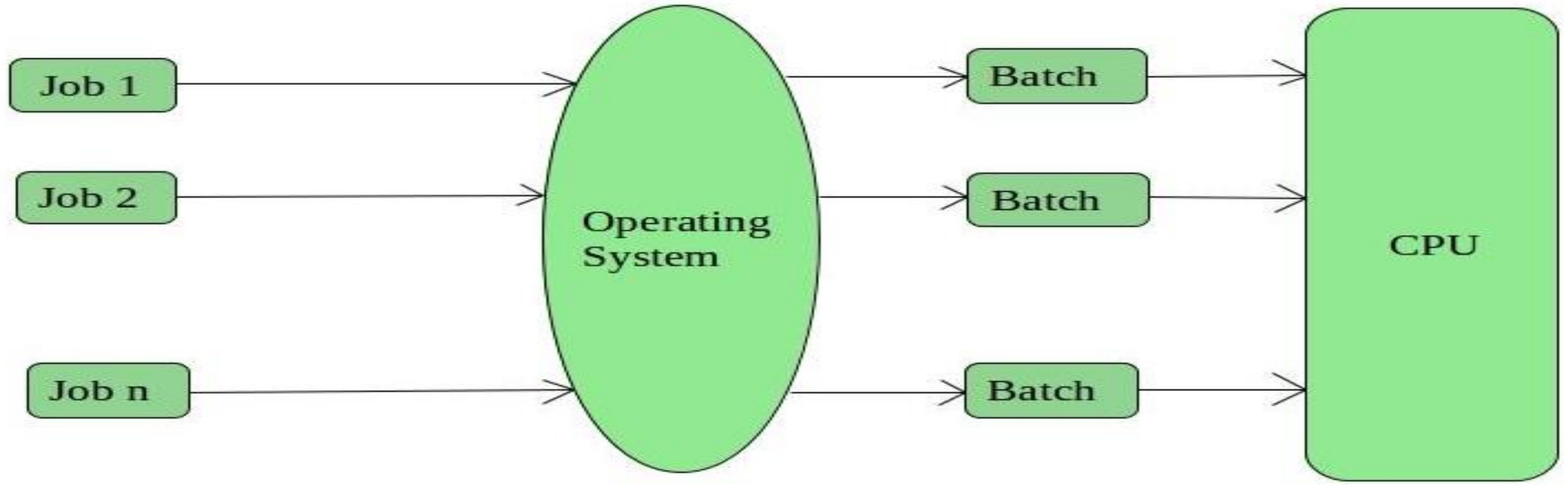
- **Processor management ---** which involves putting the tasks into order and pairing them into manageable size before they go to the CPU.
- **Memory management ---** which coordinates data to and from RAM (random-access memory) and determines the necessity for virtual memory.
- **Device management ---** which provides interface between connected devices.
- **Storage management ---** which directs permanent data storage.
- **Application ---** which allows standard communication between software and your computer.
- **User interface ---** which allows you to communicate with your computer.

Types of Operating Systems

Batch Operating System –

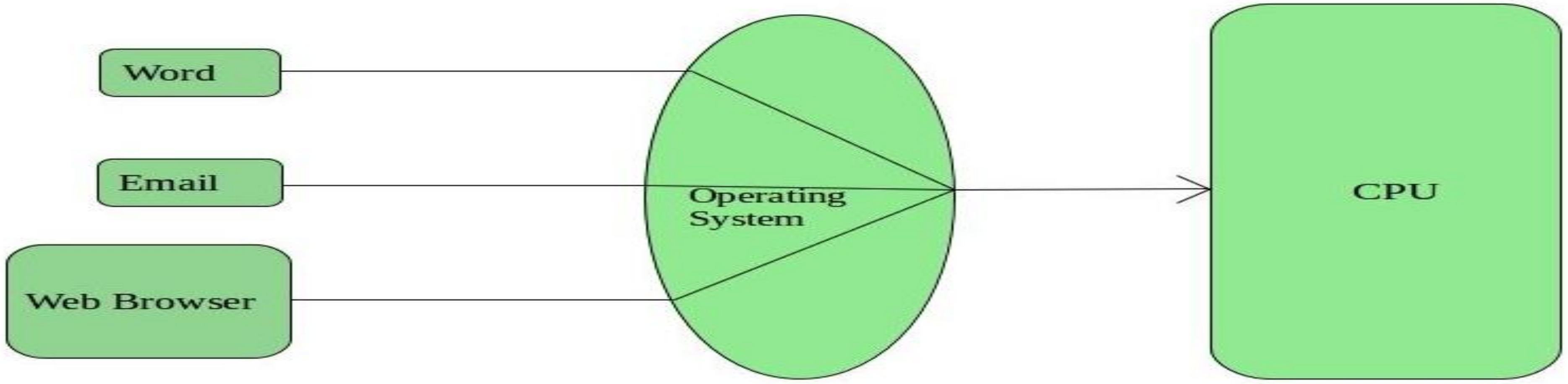
Batch Operating System

- This type of operating system do not interact with the computer directly.
- There is an operator which takes similar jobs having same requirement and group them into batches.
- It is the responsibility of operator to sort the jobs with similar needs.



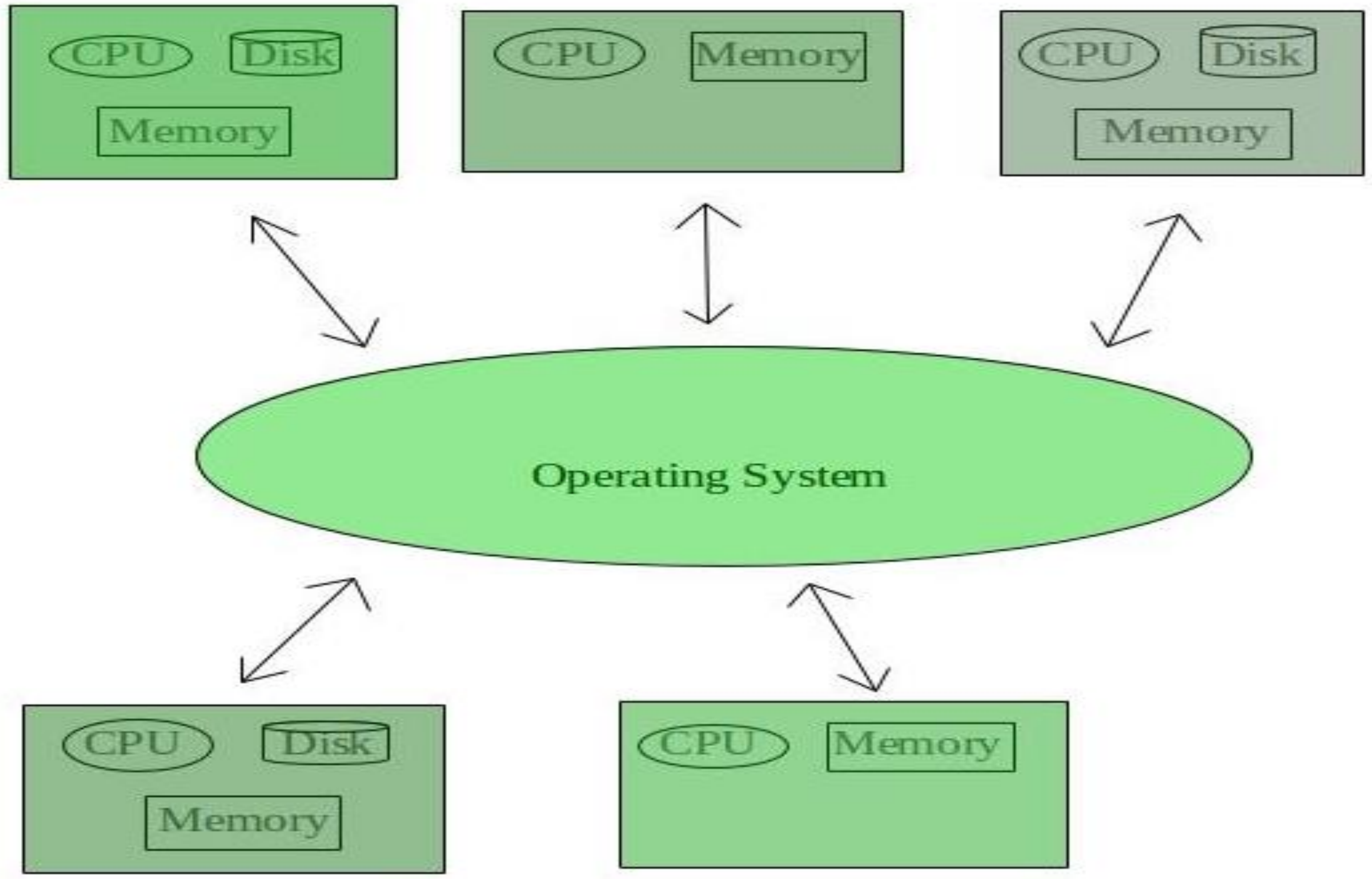
Time-Sharing Operating Systems

- Each task has given some time to execute, so that all the tasks work smoothly.
- Each user gets time of CPU as they use single system.
- These systems are also known as Multitasking Systems.
- The task can be from single user or from different users also.
- The time that each task gets to execute is called quantum.
- After this time interval is over OS switches over to next task.



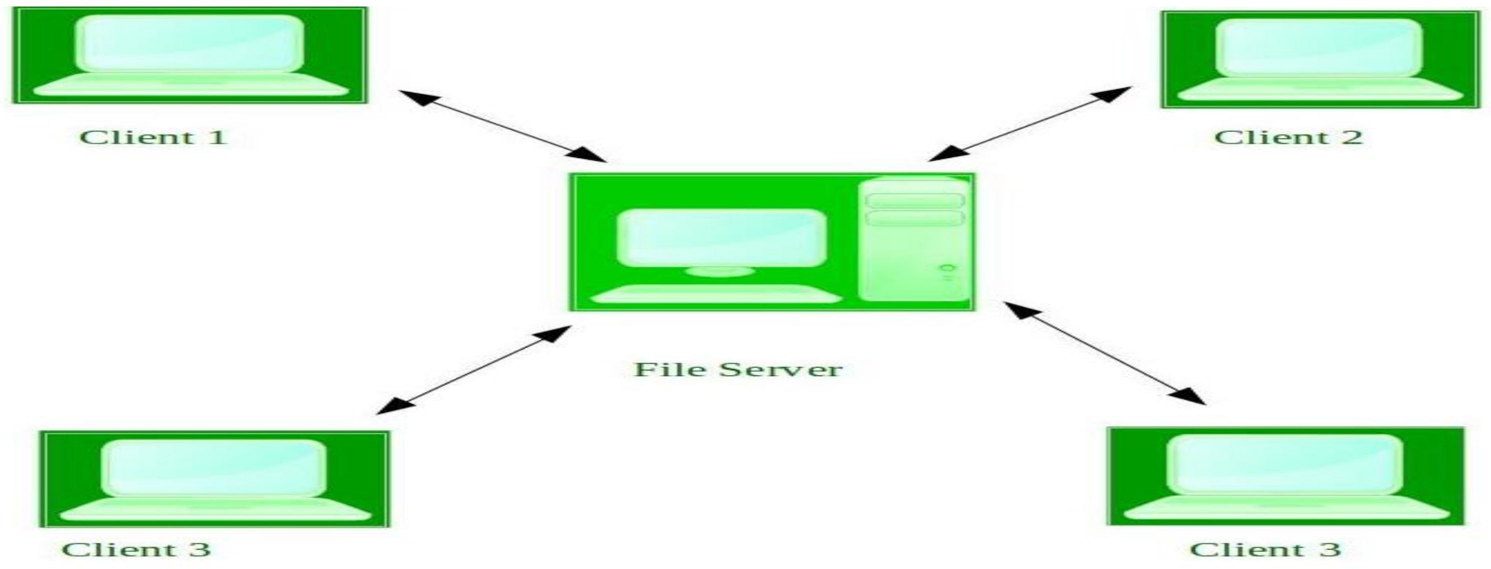
Distributed Operating System

- These types of operating system is a recent advancement in the world of computer technology and are being widely accepted all-over the world
- Various autonomous interconnected computers communicate each other using a shared communication network.
- Independent systems possess their own memory unit and CPU.
- These are referred as **loosely coupled systems** or distributed systems.
- These systems processors differ in sizes and functions.
- The major benefit of working with these types of operating system is that it is always possible that one user can access the files or software which are not actually present on his system but on some other system connected within this network i.e., remote access is enabled within the devices connected in that network.


CPU
Disk
Memory
CPU
Memory
CPU
Disk
Memory
Operating System
CPU
Disk
Memory
CPU
Memory

Network Operating System

- These systems runs on a server and provides the capability to manage data, users, groups, security, applications, and other networking functions.
- These type of operating systems allows shared access of files, printers, security, applications, and other networking functions over a small private network.
- Important aspect of Network Operating Systems is that all the users are well aware of the underlying configuration, of all other users within the network, their individual connections etc. and that’s why these computers are popularly known as **tightly coupled systems**.



Real-Time Operating System

- These types of OSs serves the real-time systems.
- The time interval required to process and respond to inputs is very small.
- This time interval is called **response time**.
- **Real-time systems** are used when there are time requirements are very strict like missile systems, air traffic control systems, robots etc.

Two types of Real-Time Operating System are

Hard Real-Time Systems:

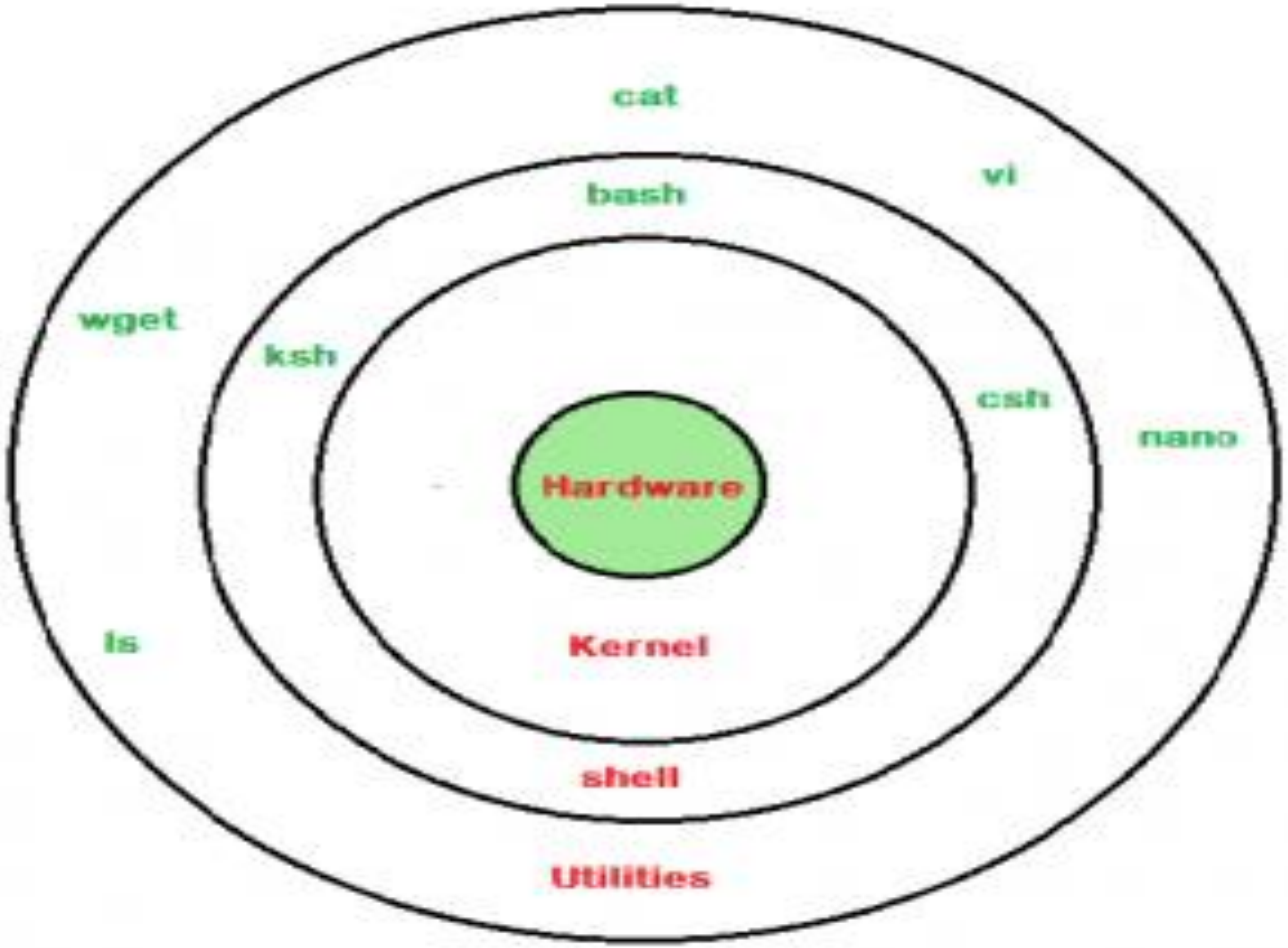
- These OSs are meant for the applications where time constraints are very strict and even the shortest possible delay is not acceptable.
- These systems are built for saving life like automatic parachutes or air bags which are required to be readily available in case of any accident.
- Virtual memory is almost never found in these systems.

Soft Real-Time Systems:

- These OSs are for applications where for time-constraint is less

strict.

Operating-System Structure


cat
vi
bash
wget
ksh
csh
nano
Hardware
ls
Kernel
shell
Utilities

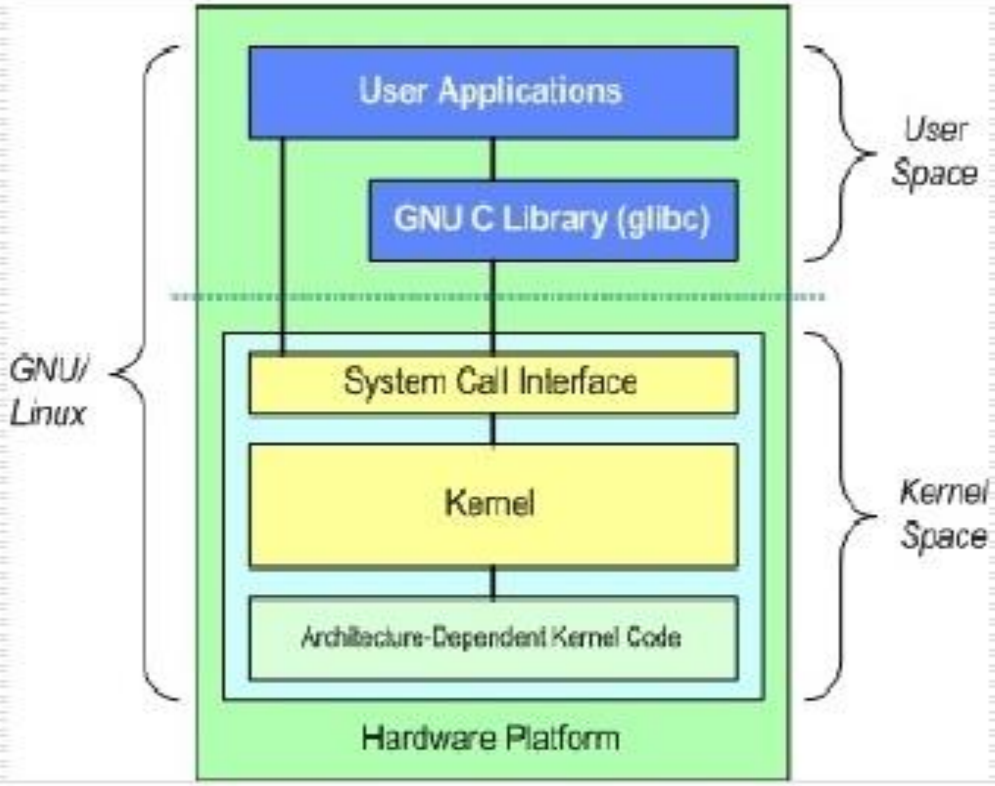
User Space vs Kernel Space

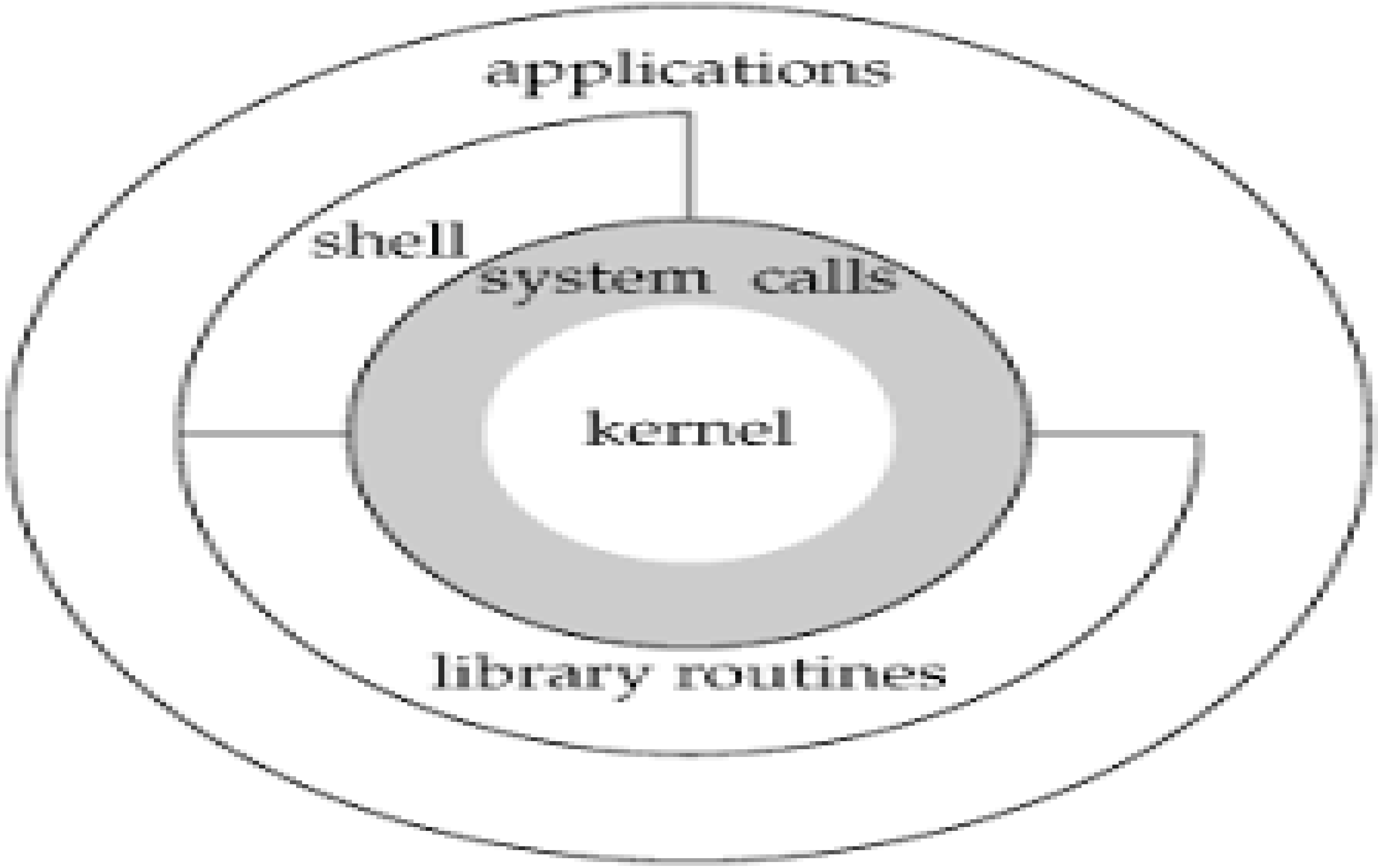
User Space

- User space is the memory area where all user mode application work and this memory can be swapped out when necessary
- User space process normally runs in its own virtual memory space and unless explicitly requested, cannot access the memory of other processes.

Kernel Space

- Kernel Space us strictly reserved for running the kernel (OS background process), kernel extensions and most device drivers
- Linux kernel space gives full access to the hardware, although some exceptions runs in user space. (The graphic system most people use with Linux does not run in kernel in contrast to that found in Microsoft Windows)




applications
shell
system calls
kernel
library routines

Process

What is a Process?

- A process is a program in execution.
- Process is not as same as program code but a lot more than it.
- A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity.
- Attributes held by process include hardware state, memory, CPU etc.

Process memory

Process memory is divided into four sections for efficient working :

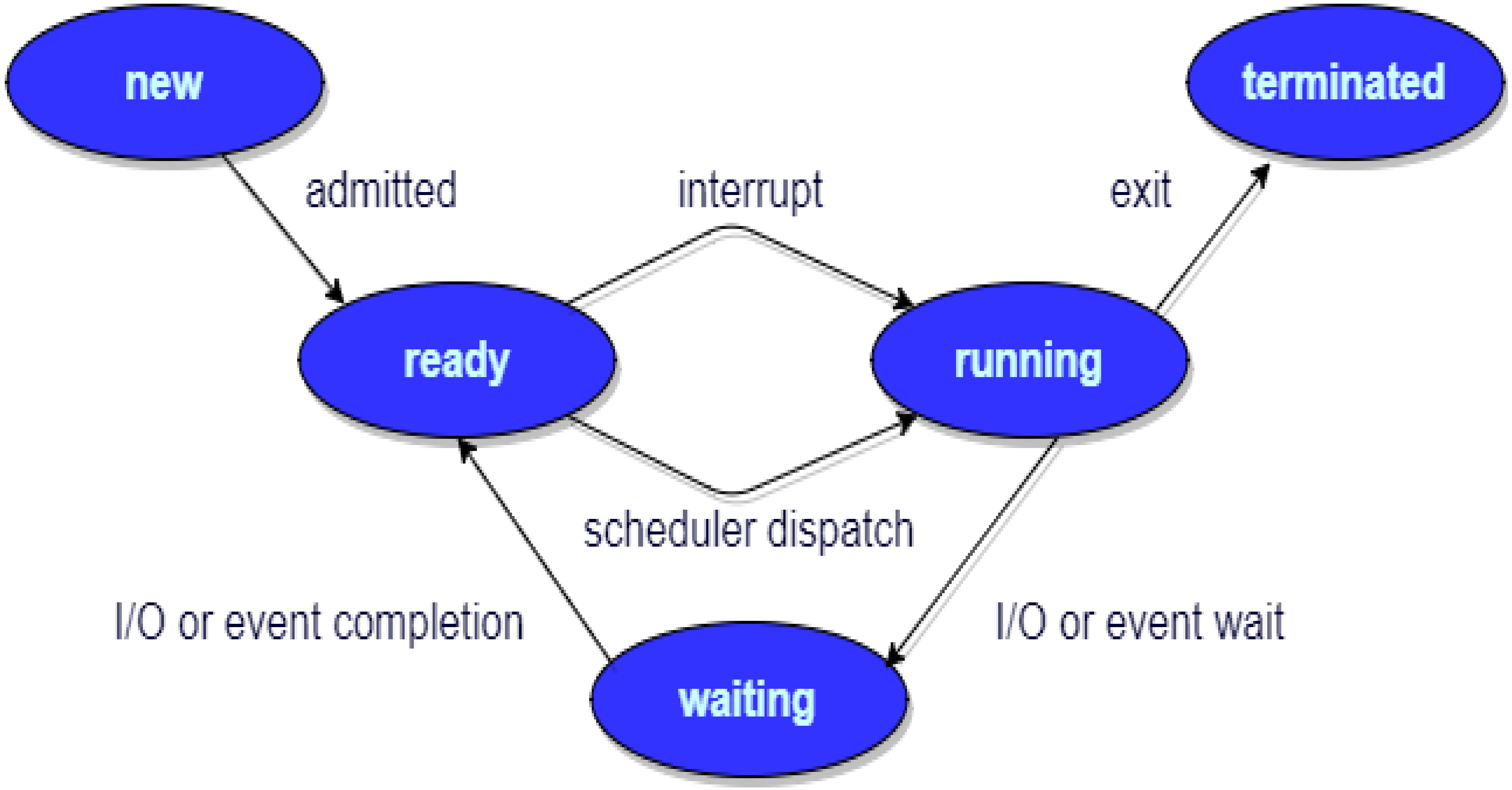
- **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- **Data section** is made up the global and static variables, allocated and initialized prior to executing the main.
- **Heap** is used for the dynamic memory allocation, and is managed via calls to new, delete, free, etc.
- **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared.

Different Process States

Processes in the operating system can be in any of the following states:

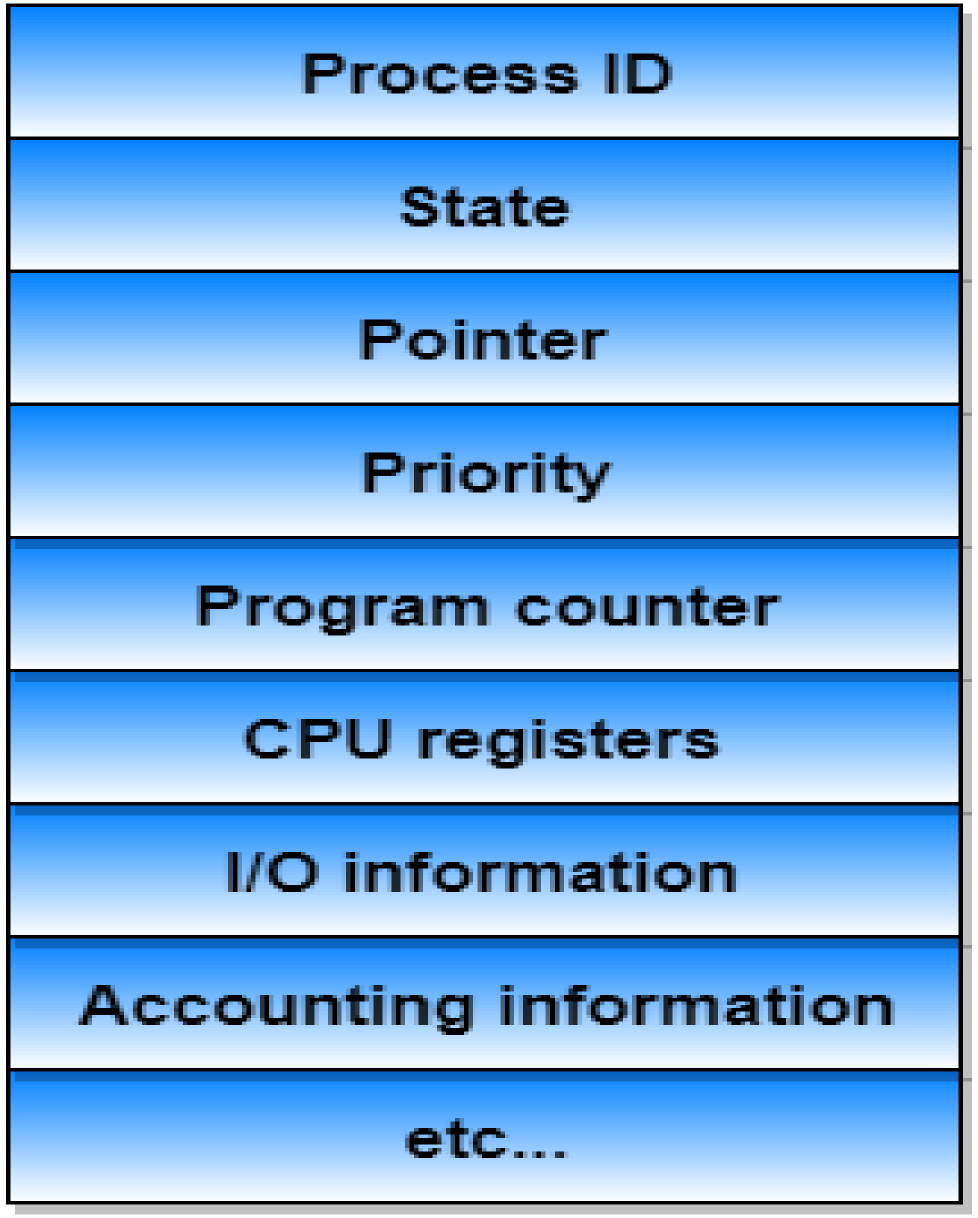
- NEW- The process is being created.
- READY- The process is waiting to be assigned to a processor.
- RUNNING- Instructions are being executed.
- WAITING- The process is waiting for some event to occur(such as an I/O completion or reception of a signal).
- TERMINATED- The process has finished execution

Different Process States



Process Control Block

- There is a Process Control Block for each process, enclosing all the information about the process. It is a data structure, which contains the following:
- **Process State**: It can be running, waiting etc.
- **Process ID** and the **parent process ID**.
- CPU registers and Program Counter. **Program Counter** holds the address of the next instruction to be executed for that process.
- **CPU Scheduling** information: Such as priority information and pointers to scheduling queues.
- **Memory Management information**: For example, page tables or segment tables.
- **Accounting information**: The User and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information**: Devices allocated, open file tables, etc.


Process ID
State
Pointer
Priority
Program counter
CPU registers
I/O information
Accounting information
etc...

What is 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**.
- The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs.
- For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

Scheduling fell into one of the two general categories:

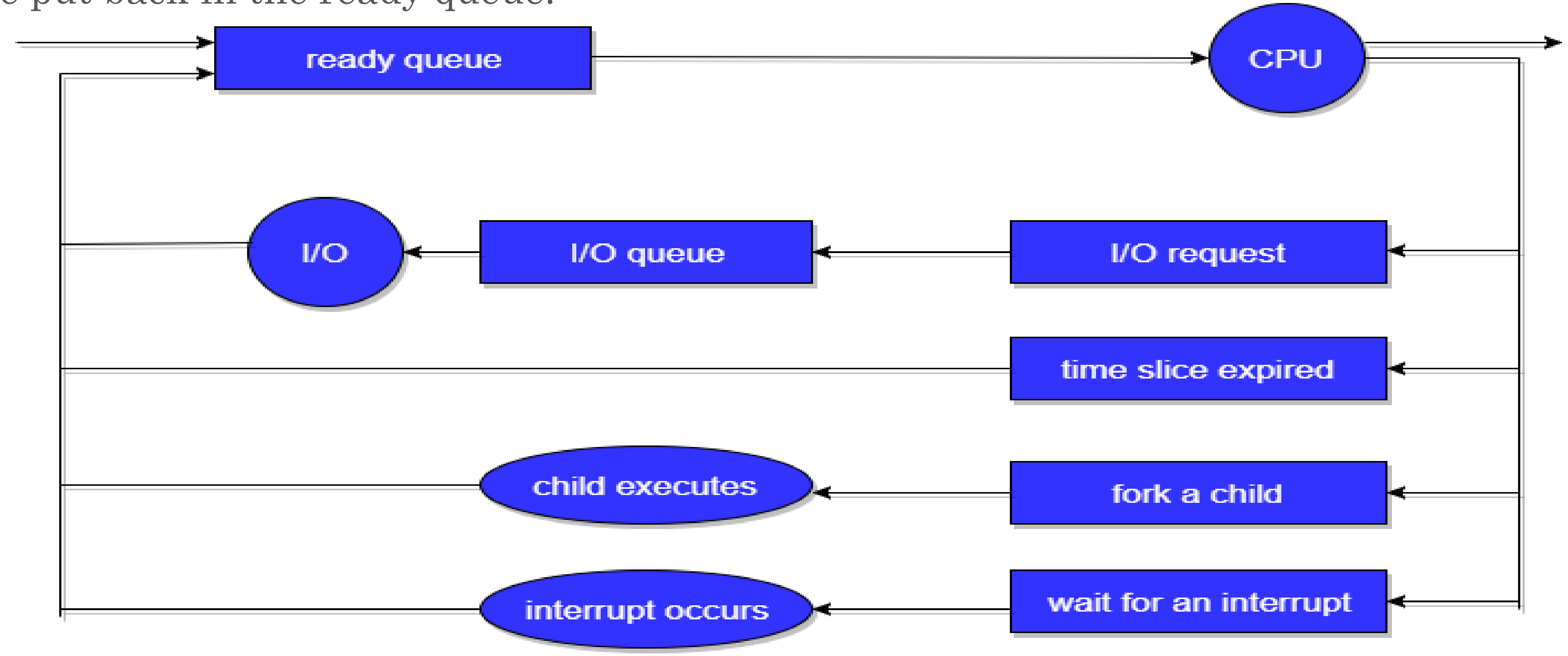
- **Non Pre-emptive Scheduling:** When the currently executing process gives up the CPU voluntarily.
- **Pre-emptive Scheduling:** When the operating system decides to favour another process, pre-empting the currently executing process

What are 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**.
- 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 subprocess 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.



Operations on Process

Process Creation

- Through appropriate system calls, processes may create other processes.
- The process which creates other process, is termed the **parent** of the other process, while the created sub-process is termed its **child**.
- Each process is given an integer identifier, termed as process identifier, or PID. The parent PID (PPID) is also stored for each process.
- A child process may receive some amount of shared resources with its parent depending on system implementation.

There are two options for the parent process after creating the child :

- Wait for the child process to terminate before proceeding.
- Run concurrently with the child, continuing to process without waiting.
- It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation.

Process Termination

Processes may also be terminated by the system for a variety of reasons, including :

- The inability of the system to deliver the necessary system resources.
- In response to a KILL command or other unhandled process interrupts.
- A parent may kill its children if the task assigned to them is no longer needed
- If the parent exits, the system may or may not allow the child to continue without a parent.
- When a process ends, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process already became an orphan.
- The processes which are trying to terminate but cannot do so because their parent is not waiting for them are termed **orphans**.
- These are eventually inherited by init as orphans and killed off.

Zombie Process:

- A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process.
- A child process always first becomes a zombie before being removed from the process table.
- The parent process reads the exit status of the child process which reaps off the child process entry from the process table.

Orphan Process:

A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called Orphan process

CPU Scheduling

What is CPU Scheduling?

- CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state).
- The aim of CPU scheduling is to make the system efficient, fast and fair.
- Whenever the CPU becomes idle, the operating system must select one of the processes in the **ready queue** to be executed.
- The selection process is carried out by the short-term scheduler (or CPU scheduler).
- The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

CPU Scheduling: Dispatcher

- Another component involved in the CPU scheduling function is the **Dispatcher**.
- The dispatcher is the module that gives control of the CPU to the process selected by the **short-term scheduler**.

Types of CPU Scheduling

CPU scheduling decisions may take place under the following four circumstances:

- When a process switches from the **running** state to the **waiting** state(for I/O request or invocation of wait for the termination of one of the child processes).
- When a process switches from the **running** state to the **ready** state (for example, when an interrupt occurs).
- When a process switches from the **waiting** state to the **ready** state(for example, completion of I/O).
- When a process **terminates**.
- In circumstances 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 in circumstances 2 and 3.
- When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is **non-preemptive**; otherwise the scheduling scheme is **preemptive**.

Non-Preemptive Scheduling

- Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

Preemptive Scheduling

- In this type of Scheduling, the tasks are usually assigned with priorities.
- At times it is necessary to run a certain task that has a higher priority before another task although it is running.
- Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

CPU Scheduling Criteria

There are many different criterias to check when considering the **"best"** scheduling algorithm, they are:

CPU Utilization

- To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

Throughput

- It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

Turnaround Time

- It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

Waiting Time

- The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

Load Average

- It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

Response Time

- Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).

- **CPU Burst:** The time required by Process for execution.

Optimization Criteria

- Max CPU Utilization
- Max Throughput
- Min Turnaround time
- Min Waiting time
- Min response time

Scheduling Algorithms

To decide which process to execute first and which process to execute last to achieve maximum CPU utilisation, computer scientists have defined some algorithms, they are:

- **First Come First Serve(FCFS) Scheduling**
- **Shortest-Job-First(SJF) Scheduling**
- **Priority Scheduling**
- **Round Robin(RR) Scheduling**

First Come First Serve (FCFS) Scheduling

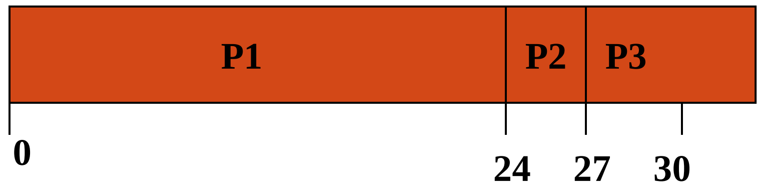
- Policy: Process that requests the CPU *FIRST* is allocated the CPU *FIRST*.
  - FCFS is a non-preemptive algorithm.
- Implementation - using FIFO queues
    - incoming process is added to the tail of the queue.
    - Process selected for execution is taken from head of queue.
- Performance metric - Average waiting time in queue.
- Gantt Charts are used to visualize schedules.

First-Come, First-Served(FCFS) Scheduling

- Example

Process	Burst Time
P1	24
P2	3
P3	3

Gantt Chart for Schedule



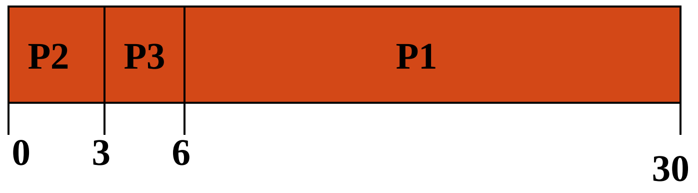
- Suppose the arrival order for the processes is
  - P1, P2, P3
- Waiting time
  - P1 = 0;
  - P2 = 24;
  - P3 = 27;
- Average waiting time
  - (0+24+27)/3 = 17

FCFS Scheduling (cont.)

- Example

Process	Burst Time
P1	24
P2	3
P3	3

Gantt Chart for Schedule



- Suppose the arrival order for the processes is
  - P2, P3, P1
- Waiting time
  - P1 = 6; P2 = 0; P3 = 3;
- Average waiting time
  - (6+0+3)/3 = 3 , better..
- *Convoy Effect*:
  - short process behind long process, e.g. 1 CPU bound process, many I/O bound processes.

What is Convoy Effect?

- Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time.
- This essentially leads to poor utilization of resources and hence poor performance.

Shortest-Job-First(SJF) Scheduling

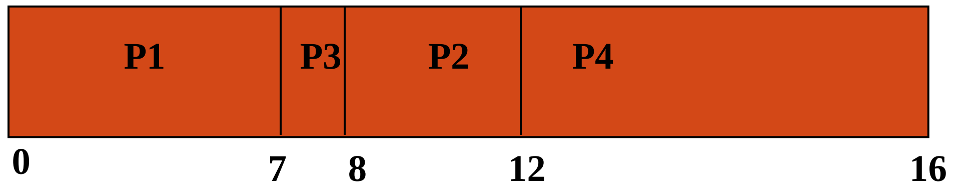
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two Schemes:
  - **Scheme 1: Non-preemptive**
    - Once CPU is given to the process it cannot be preempted until it completes its CPU burst.
  - **Scheme 2: Preemptive**
    - If a new CPU process arrives with CPU burst length less than remaining time of current executing process, preempt. Also called Shortest-Remaining-Time-First (SRTF).
  - SJF is optimal - gives minimum average waiting time for a given set of processes.

Non-Preemptive SJF Scheduling

- Example

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Gantt Chart for Schedule



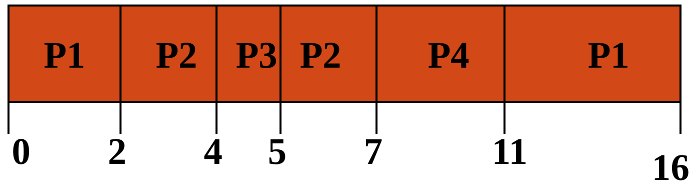
Average waiting time = [0+(8-2)+(7-4)+(12-5)]/4 = 4

Preemptive SJF Scheduling(SRTF)

- Example

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Gantt Chart for Schedule



Average waiting time = (11-2)+(5-2-2)+(4-4)+(5-7))/4 = 3

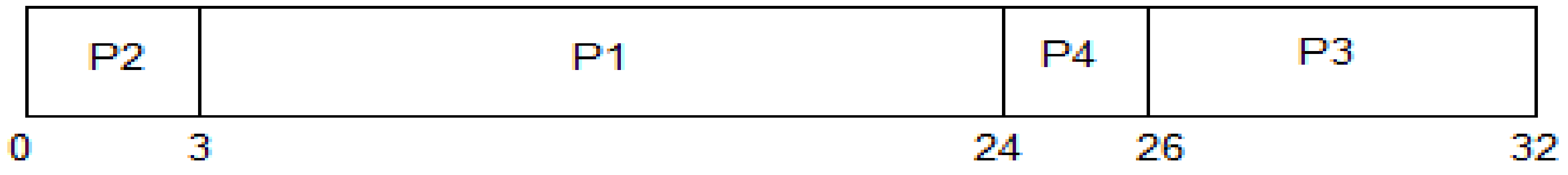
Priority Scheduling

- Priority is assigned for each process.
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Priority Scheduling

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,

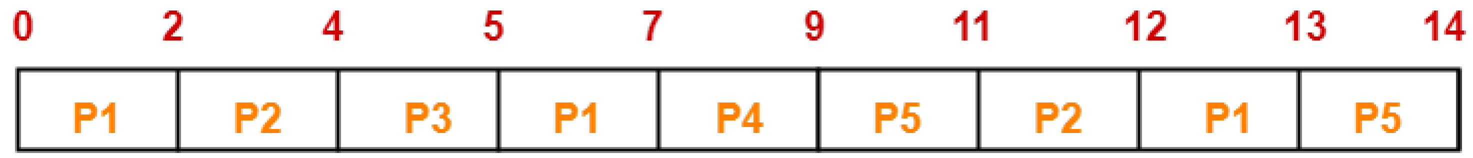


The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

Round Robin Scheduling

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preemptied and other process executes for given time period.
- Context switching is used to save states of preemptied processes.

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



Gantt Chart

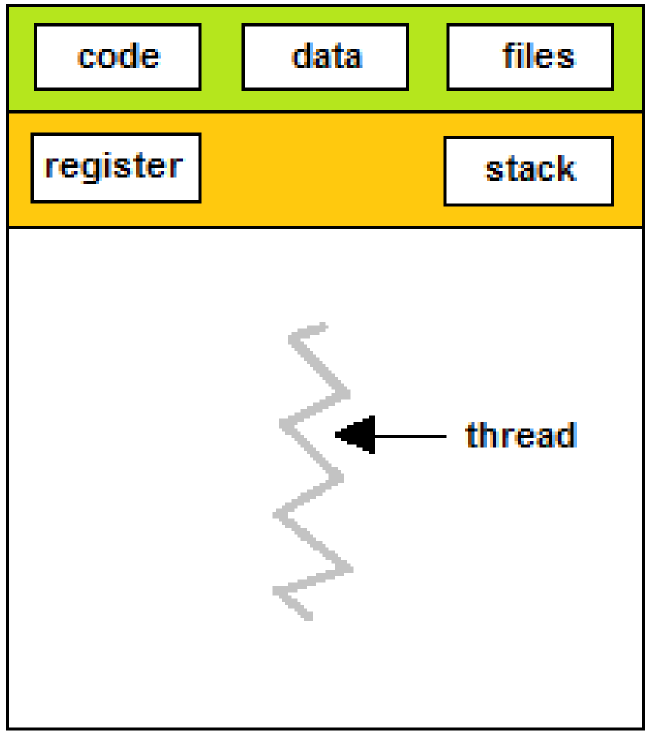
Process Id	**Exit time**	**Turn Around time**	**Waiting time**
P1	13	13 – 0 = 13	13 – 5 = 8
P2	12	12 – 1 = 11	11 – 3 = 8
P3	5	5 – 2 = 3	3 – 1 = 2
P4	9	9 – 3 = 6	6 – 2 = 4
P5	14	14 – 4 = 10	10 – 3 = 7

- Average waiting time = (8 + 8 + 2 + 4 + 7) / 5 = 29 / 5 = 5.8 unit

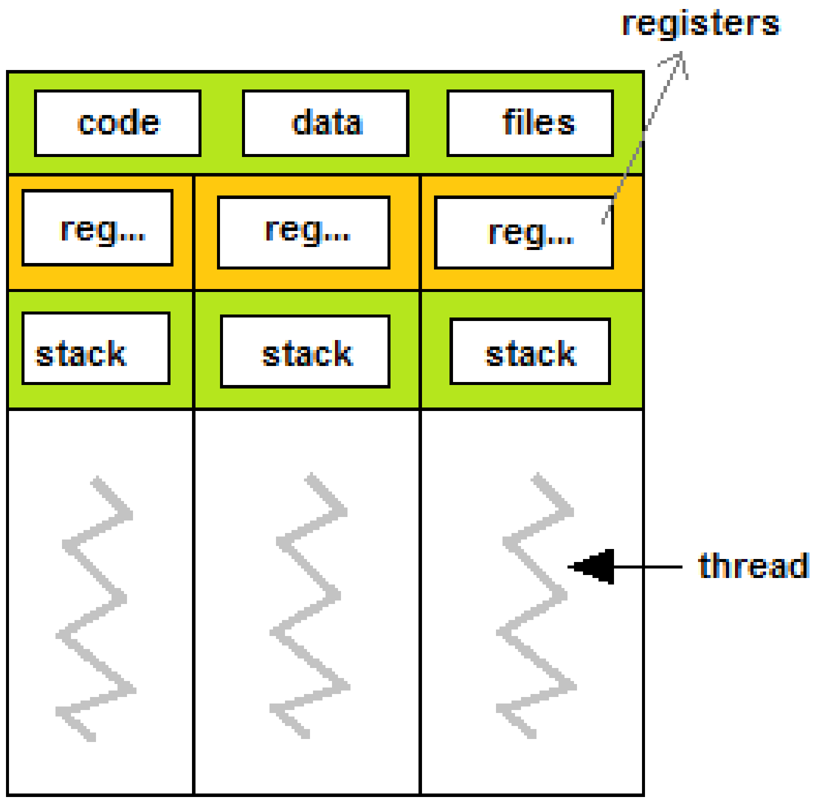
What is Thread?

- **Thread** is an execution unit which consists of its own program counter, a stack, and a set of registers.
- Threads are also known as Lightweight processes.
- Threads are popular way to improve application through parallelism.
- The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.
- As each thread has its own independent resource for process execution, multpile processes can be executed parallely by increasing number of threads.

- **Program counter** that keeps track of which instruction to execute next,
- **System registers** which hold its current working variables,
- **Stack** which contains the execution history.



single-threaded process



multithreaded process

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Types of Thread

There are two types of threads:

User threads,

Are above the kernel and without kernel support.

These are the threads that application programmers use in their programs.

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.

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

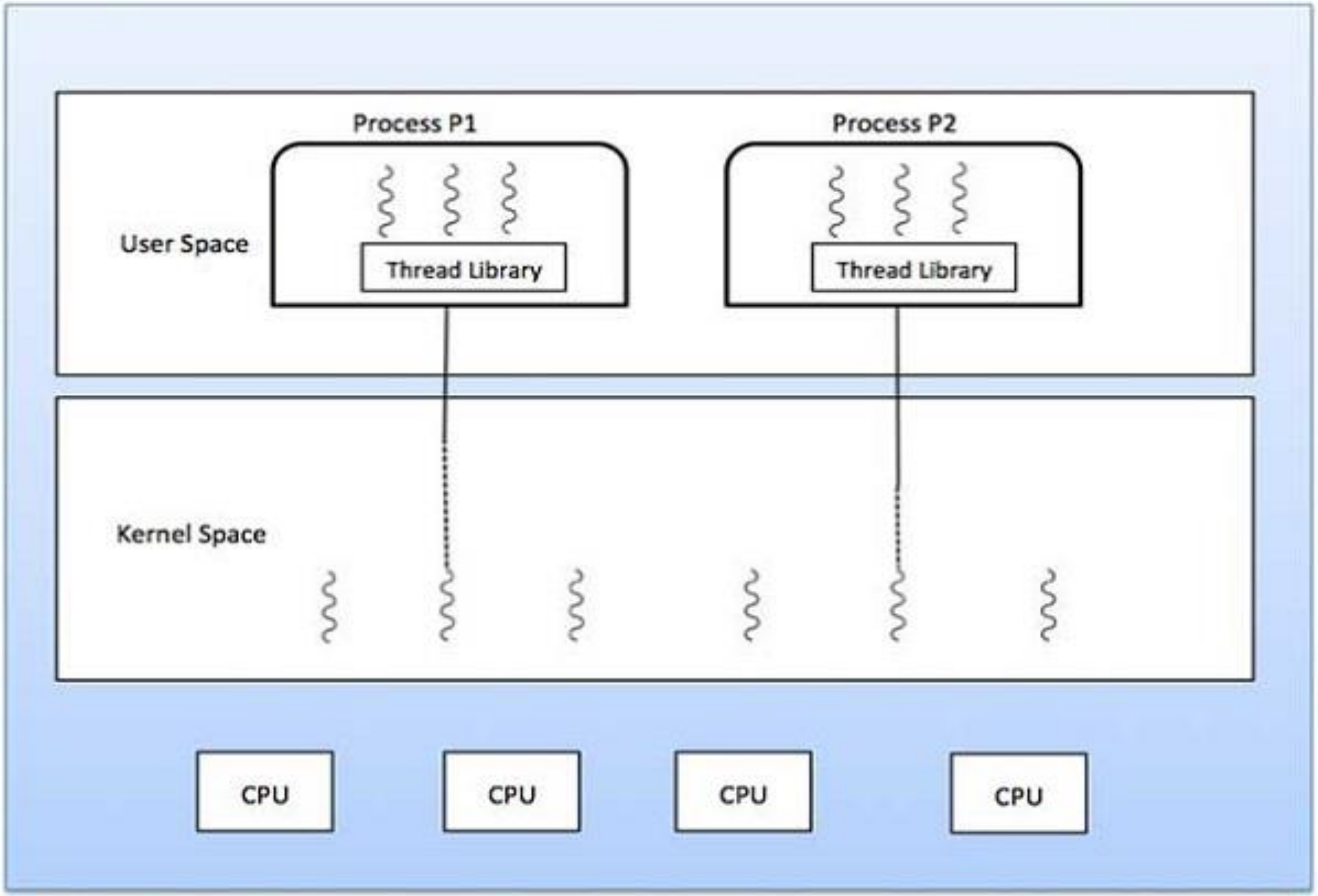
Multithreading Models

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

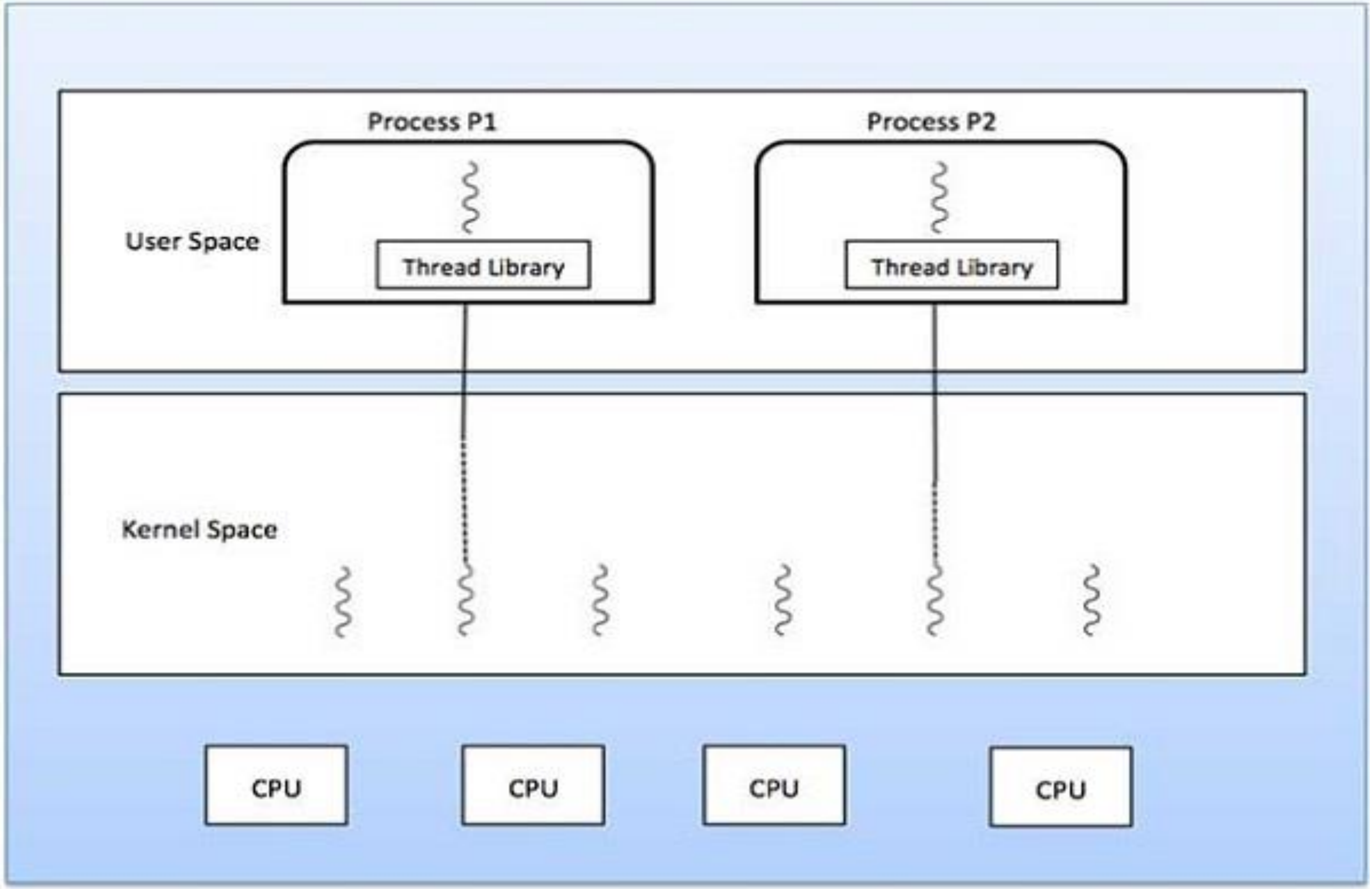
Many to One Model

- 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 efficient in nature.


Process P1
Process P2
User Space
Thread Library
Thread Library
Kernel Space
CPU
CPU
CPU
CPU

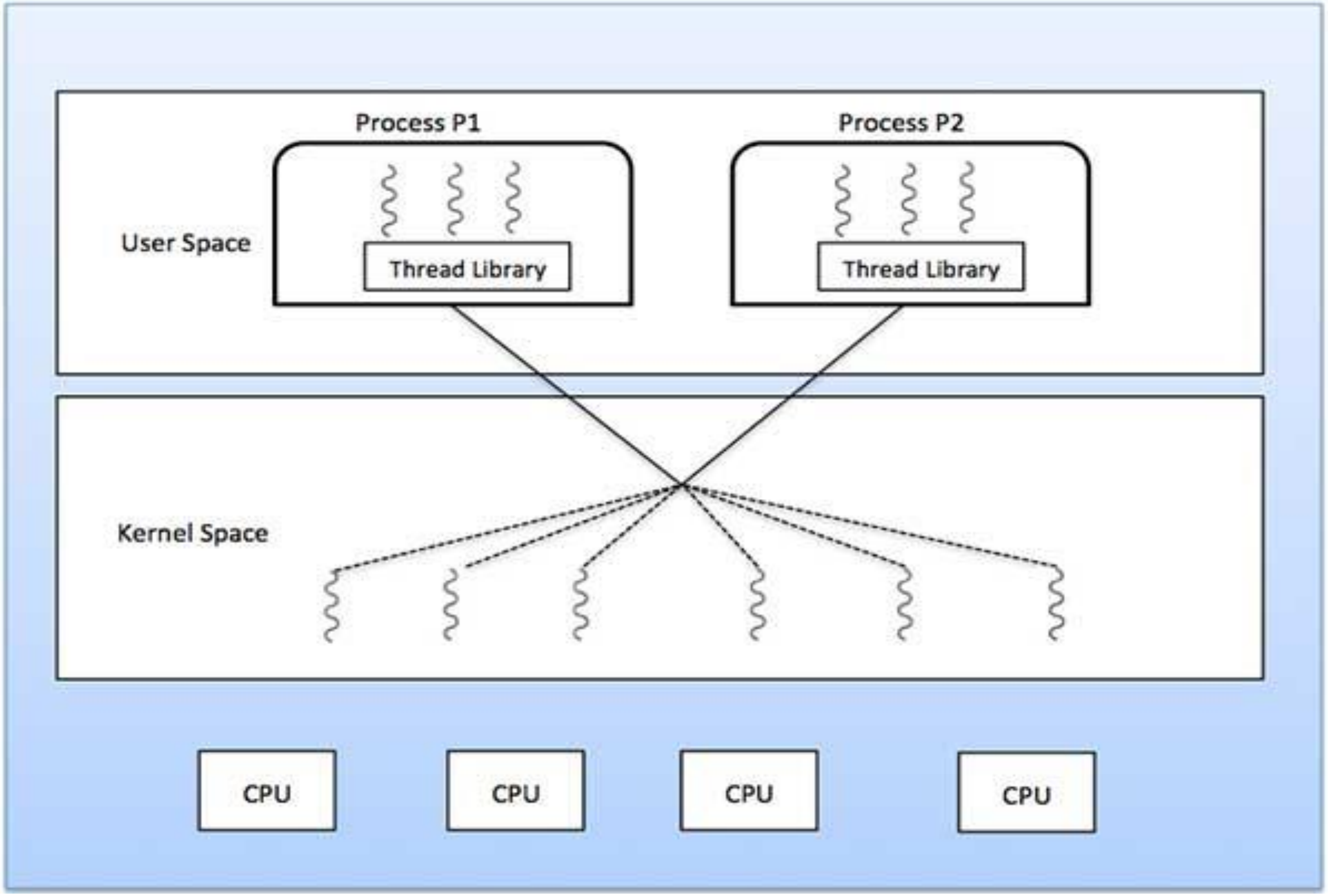
One to One Model

- The **one to one** model creates a separate kernel thread to handle each and every user thread.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.


Process P1
Process P2
User Space
Thread Library
Thread Library
Kernel Space
CPU
CPU
CPU
CPU

Many to Many Model

- The **many to many** model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of the threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.


Process P1
Process P2
User Space
Thread Library
Thread Library
Kernel Space
CPU
CPU
CPU
CPU

Benefits of Multithreading

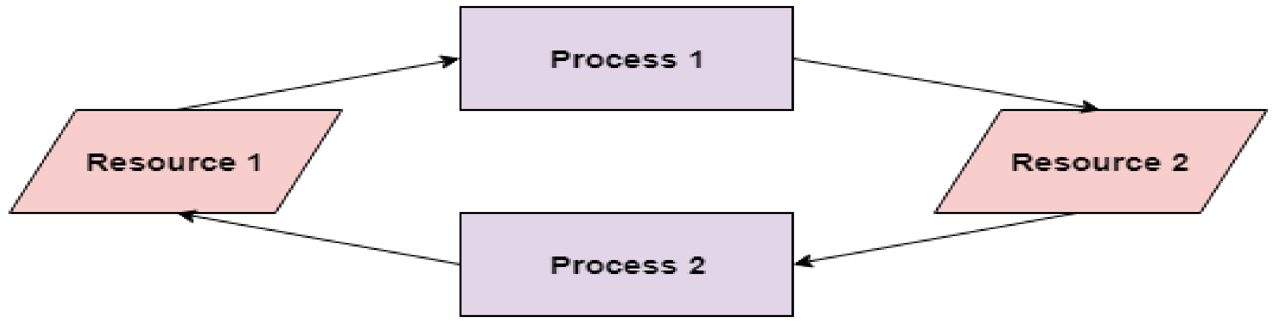
- Responsiveness
- Resource sharing, hence allowing better utilization of resources.
- Economy. Creating and managing threads becomes easier.
- Scalability. One thread runs on one CPU.
- In Multithreaded processes, threads can be distributed over a series of processors to scale.
- Context Switching is smooth. Context switching refers to the procedure followed by CPU to change from one task to another.

Multithreading Realtime Examples

- Background jobs like running application servers like Oracle application server, Web servers like Tomcat etc which will come into action whenever a request comes.
- Typing MS Word document while listening to music.
- Games are very good examples of threading. You can use multiple objects in games like cars, motor bikes, animals, people etc. All these objects are nothing but just threads that run your game application.
- Railway ticket reservation system where multiple customers accessing the server.
- Multiple account holders accessing their accounts simultaneously on the server. When you insert a ATM card, it starts a thread for perform your operations.

Deadlock

- A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.



Deadlock in Operating System

process 1 has resource 1 and needs to acquire resource 2.

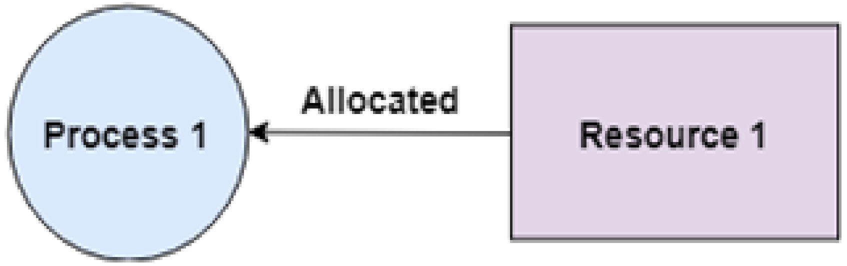
process 2 has resource 2 and needs to acquire resource 1.

Process 1 and process 2 are in deadlock as each of them needs the other's resource to complete their execution but neither of them is willing to relinquish their resources.

Necessary conditions for Deadlocks

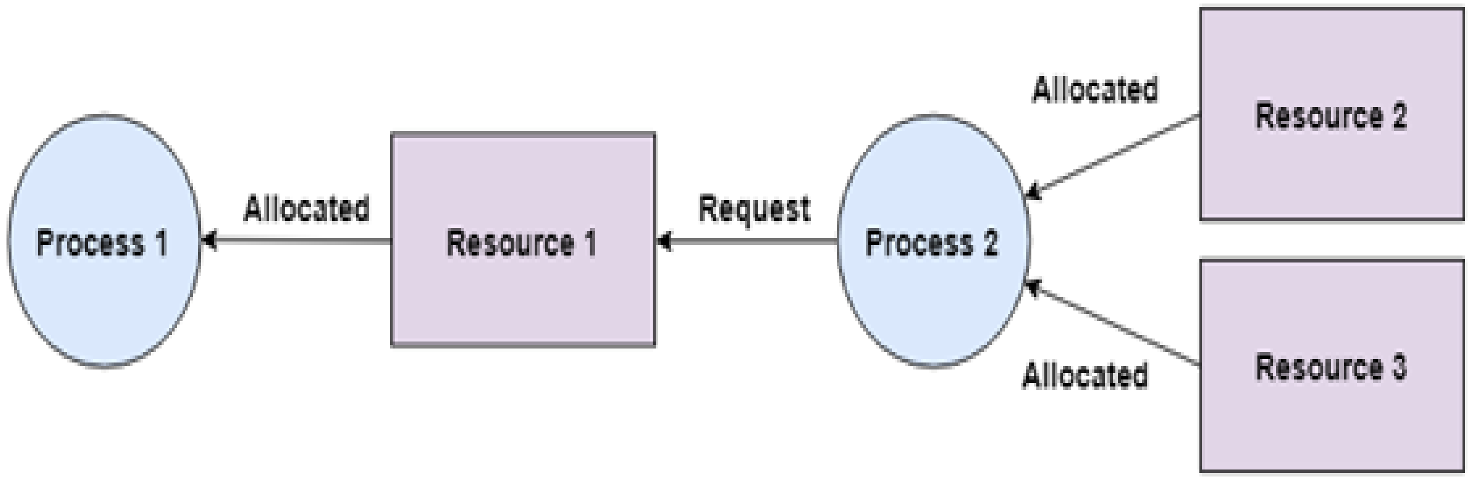
Mutual Exclusion

- There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



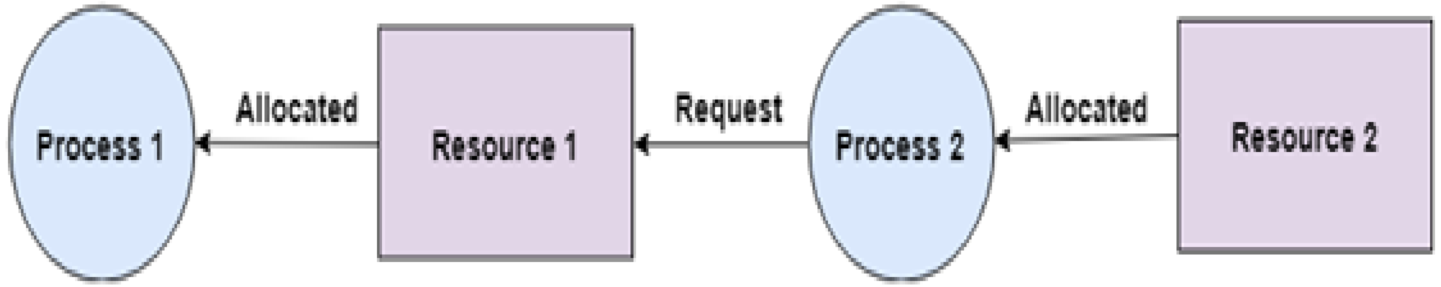
Hold and Wait

- A process can hold multiple resources and still request more resources from other processes which are holding them.
- In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



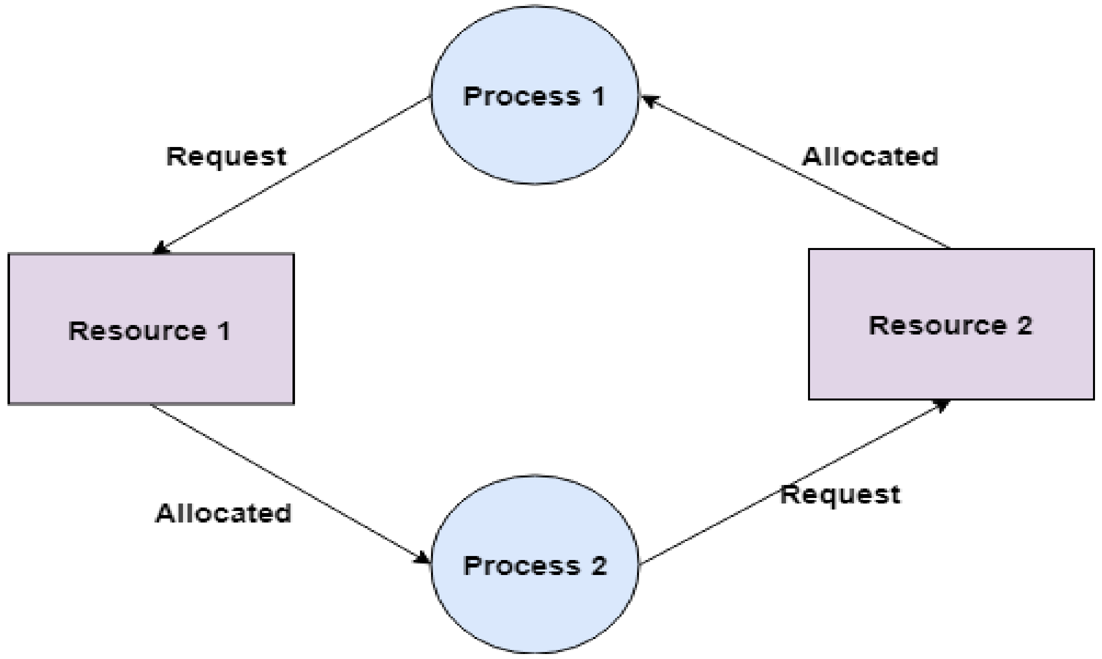
No Preemption

- A resource cannot be preempted from a process by force. A process can only release a resource voluntarily.
- Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



Circular Wait

- A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain.
- For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Deadlock Prevention

We can prevent Deadlock by eliminating any of the four conditions.

1.Mutual Exclusion

2.Hold and Wait

3.No preemption

4.Circular wait

Eliminate Mutual Exclusion

It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tape drive and printer, are inherently non-shareable.

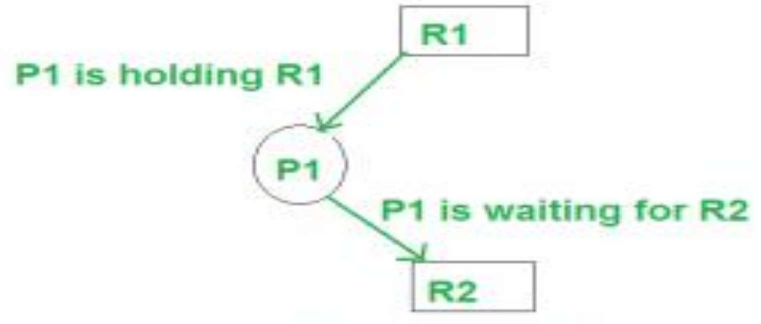
Eliminate No Preemption

Preempt resources from the process when resources required by other high priority processes.

Eliminate Hold and wait

1.Allocate all required resources to the process before the start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization.

2.The process will make a new request for resources after releasing the current set of resources. This solution may lead to starvation.



HOLD AND WAIT

Eliminate Circular Wait

Each resource will be assigned with a numerical number.

A process can request the resources increasing/decreasing order of numbering.

For Example,

if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5

such request will not be granted, only request for resources more than R5 will be granted.

Deadlock Avoidance

Deadlock avoidance can be done with Banker’s Algorithm.

Banker’s Algorithm

Bankers’s Algorithm is resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it checks for the safe state, if after granting request system remains in the safe state it allows the request and if there is no safe state it doesn’t allow the request made by the process.

Inputs to Banker's Algorithm:

1.Max need of resources by each process.

2.Currently allocated resources by each process.

3.Max free available resources in the system.

The request will only be granted under the below condition:

1.If the request made by the process is less than equal to max need to that process.

2.If the request made by the process is less than equal to the freely available resource in the system.

```
Total resources in system:
 A B C D
 6 5 7 6
```

```
Available system resources are:

A B C D
3 1 1 2
```

```
Processes (currently allocated resources):
   A B C D
P1 1 2 2 1
P2 1 0 3 3
P3 1 2 1 0
```

```
Processes (maximum resources):
   A B C D
P1 3 3 2 2
P2 1 2 3 4
P3 1 3 5 0
```

```
Need = maximum resources - currently allocated
resources. Processes (need resources):
   A B C D
P1 2 1 0 1
P2 0 2 0 1
P3 0 1 4 0
```

fork()

- Fork system call is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (parent process).
- After a new child process is created, both processes will execute the next instruction following the fork() system call.
- A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.

Zero: Returned to the newly created child process.

Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

To install C on Centos

```
# yum groupinstall 'Development Tools'
```

How to create programme

```
# nano fork.c
```

To compile

```
# gcc fork.c
```

To run

```
#./a.out
```

fork() in C

There are no arguments in fork() and the return type of fork() is integer. You have to include the following header files when fork() is used:

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

When working with fork(), <sys/types.h> can be used for type ***pid_t*** for processes ID's as pid_t is defined in <sys/types.h>.

The header file <unistd.h> is where fork() is defined so you have to include it to your program to use fork().

The return type is defined in <sys/types.h> and fork() call is defined in <unistd.h>. Therefore, you need to include both in your program to use fork() system call.

Example 1: Calling fork()

Consider the following example in which we have used the fork() system call to create a new child process:

CODE:

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

int main()
{
    fork();
    printf("Using fork() system call\n");
    return 0;
}
```

OUTPUT:

```
Using fork() system call
Using fork() system call
```

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

int main()
{
   pid_t p;
   p = fork();
   if(p==-1)
   {
      printf("There is an error while calling fork()");
   }
   if(p==0)
   {
      printf("We are in the child process");
   }
   else
   {
      printf("We are in the parent process");
   }
   return 0;
}
```

OUTPUT:

```
We are in the parent process
We are in the child process
```

To create Zombie process

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

int main()
{
int pid=fork(); //create new process

if(pid>0)
{
        sleep(60); //parent sleeps for 60 sec

}
else
{       exit(0);  // child exits before the parent & child becomes zombie

}
 return 0;
}
```

OUTPUT

```
# ./a.out  &

[1] 9006


# pstree -p 9006

a.out(9006)  ------  âââa.out(9007)          // 9007 is child PID


# ps -aux | grep 9007


user1      9007  0.0  0.0      0     0 pts/0    Z    12:19   0:00 [a.outct>
```

To create Orphan process

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

int main()
{
int pid=fork(); //create new process

if(pid>0)
{
        exit(0);  //parent exit before child

}
else if(pid==0)
{

sleep(60);  // child sleeps for 60 second
}
 return 0;
}
```

```
[user1@localhost shell]$ ./a.out &
[1] 10506
[1]+  Done                    ./a.out

[user1@localhost shell]$ ps -aux | grep a.out

user1     10510  0.0  0.0   4212    88 pts/0    S    12:32   0:00 ./a.out
user1     10515  0.0  0.0 112812   984 pts/0    R+   12:33   0:00 grep --color=auto a.out

[user1@localhost shell]$ ps -o ppid 10510
  PPID
     1

[user1@localhost shell]$ pstree -p 1

systemd(1)─┬─ModemManager(6586)─┬─{ModemManager}(6639)
           │                    └─{ModemManager}(6642)
           ├─NetworkManager(6726)─┬─dhclient(7036)
           │                      ├─{NetworkManager}(6736)
           │                      └─{NetworkManager}(6738)
           ├─VGAuthService(6584)
           ├─a.out(10510)
           ├─abrt-watch-log(6588)
           ├─abrt-watch-log(6590)
```