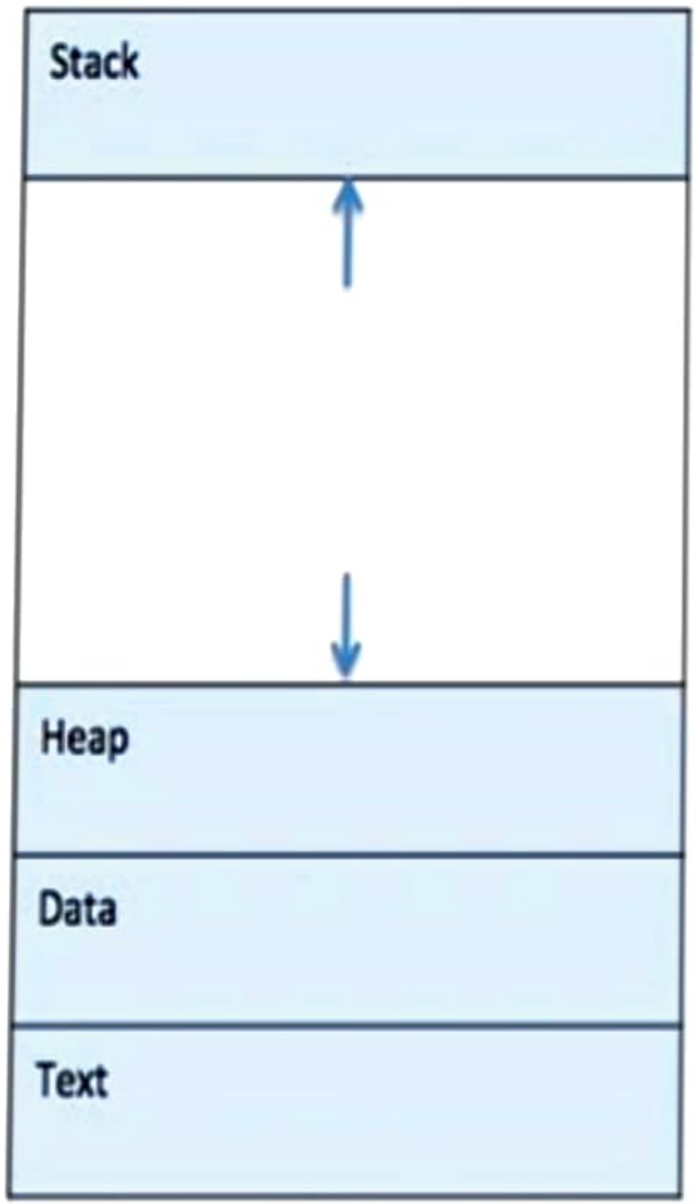
Process

- A process is basically a program in execution. The execution of a process must progress in a sequential fashion.
- A process is defined as an entity which represents the basic unit of work to be implemented in the system.
- To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.
- When a program is loaded into the memory and it becomes a process, it can be divided into four sections — stack, heap, text and data. The following image shows a simplified layout of a process inside main memory –


Stack
Heap
Data
Text

Stack

- The process Stack contains the temporary data such as method/function parameters, return address and local variables.

- **Heap**
- This is dynamically allocated memory to a process during its run time.

- **Text**
- This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.

- **Data**
- This section contains the global and static variables.

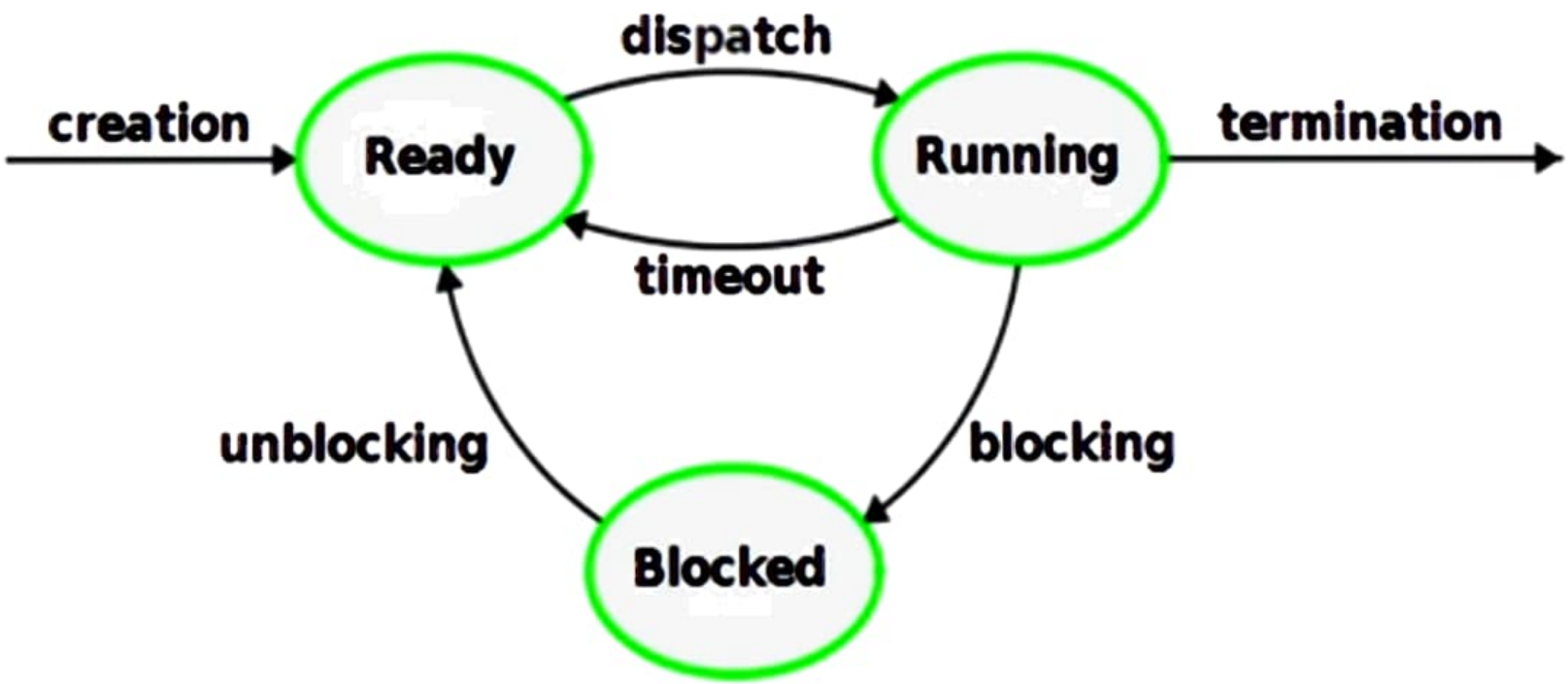
Process Relationships Life Cycle

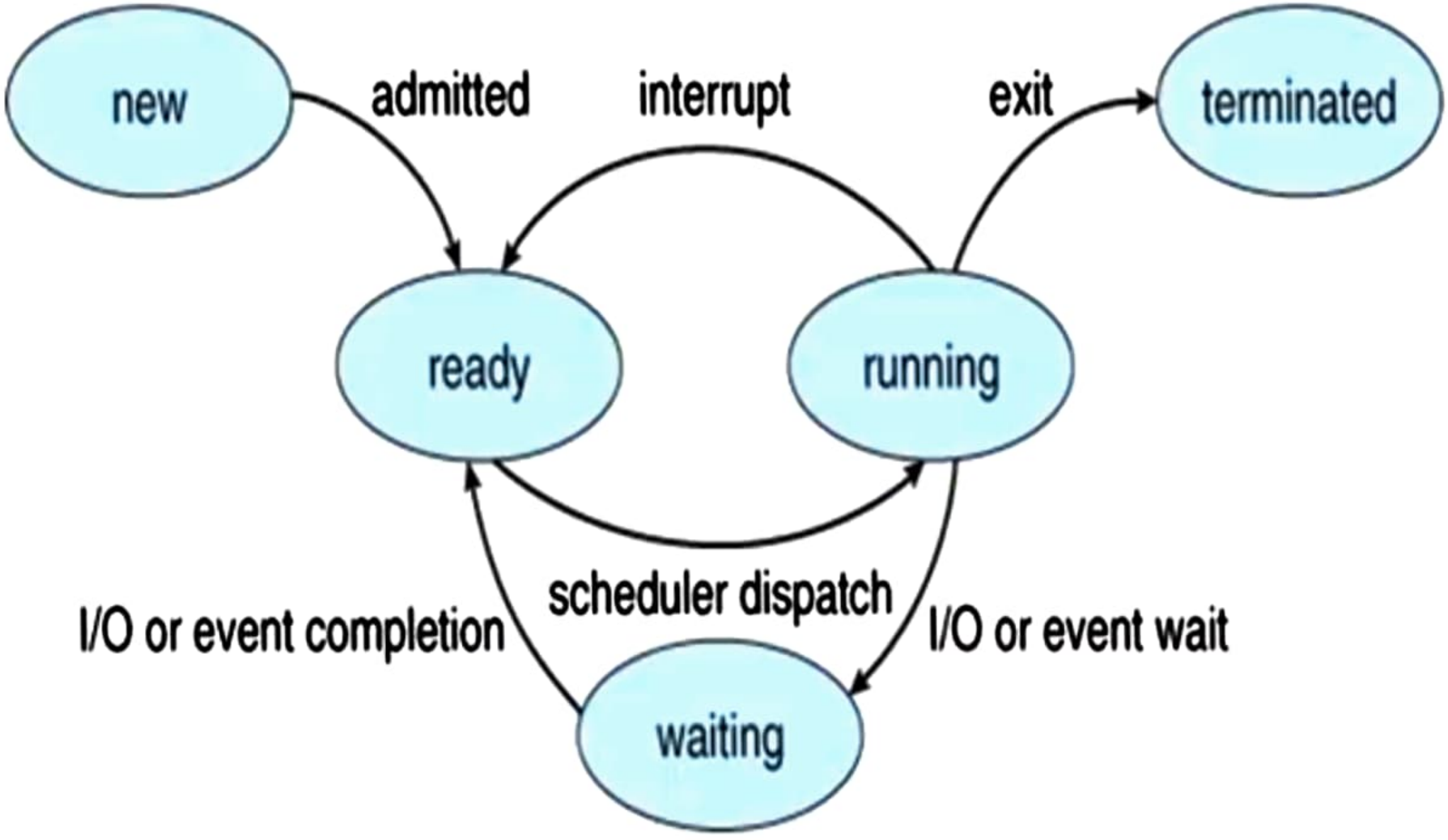
When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized

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

Different states of a Process

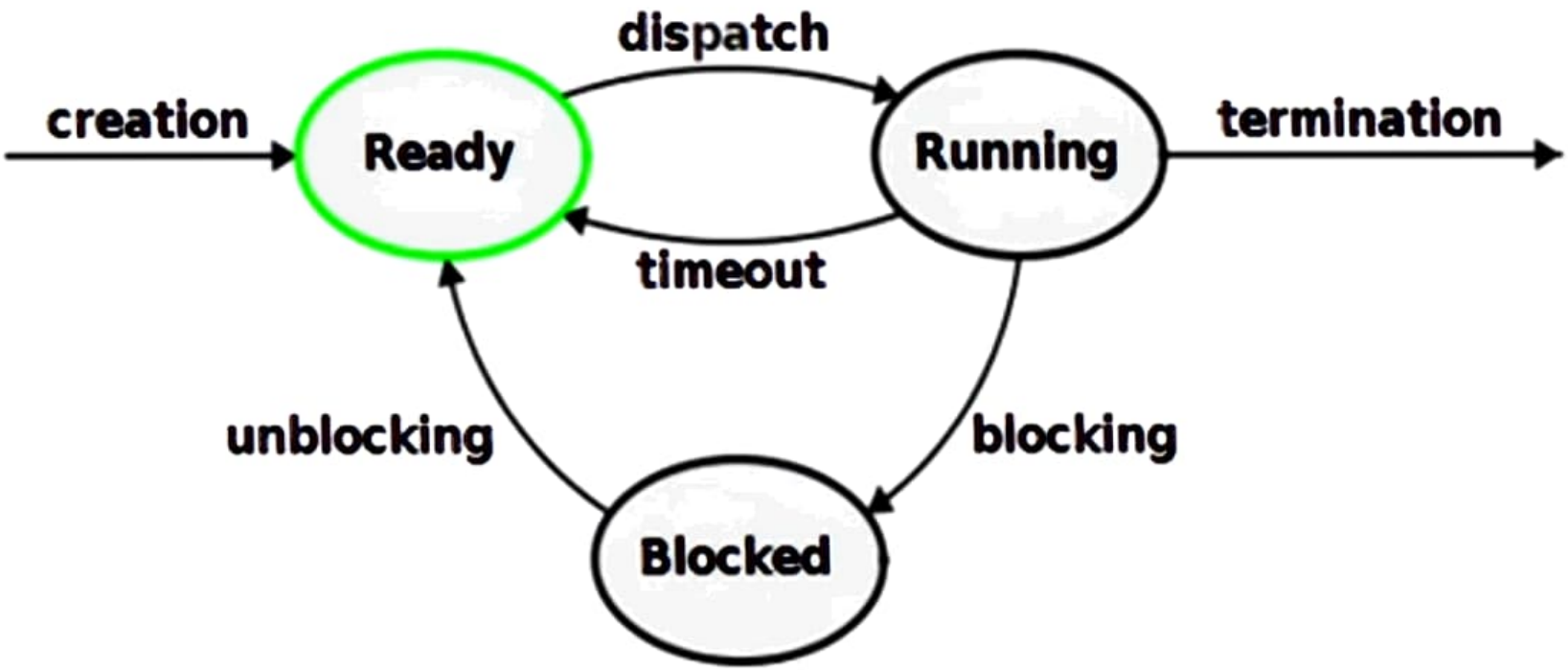
The state of a process represent its execution status.




new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

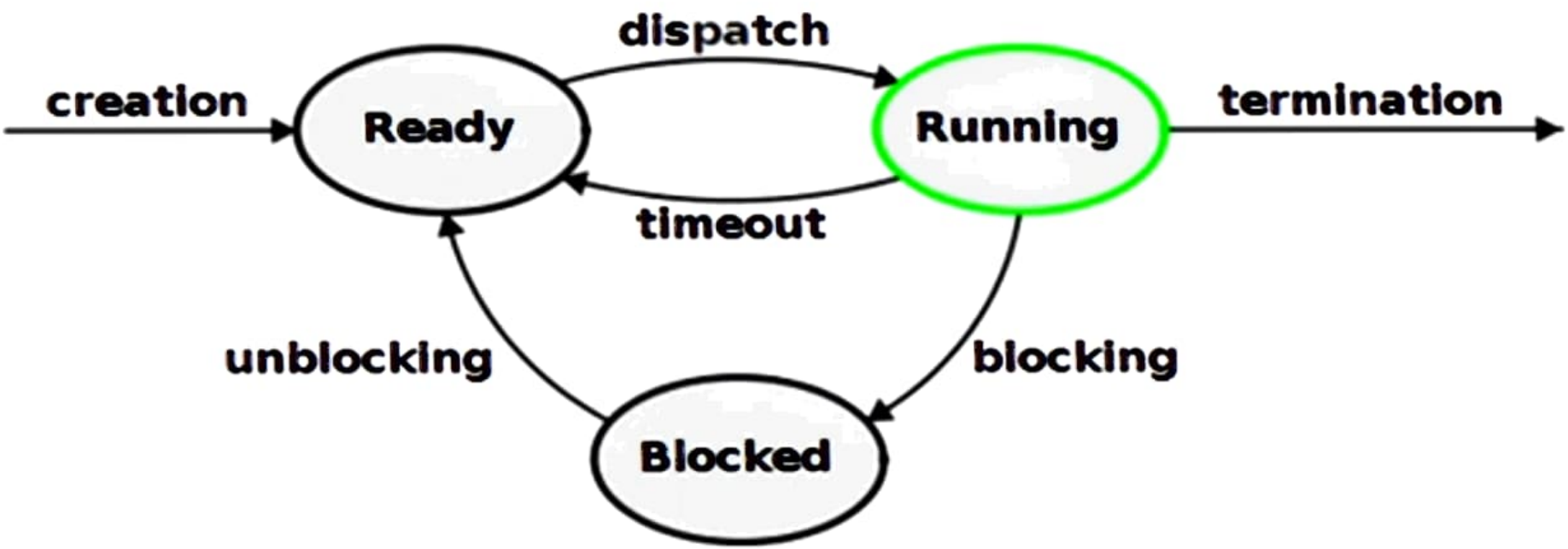
Ready

A process in the ready state has all of the resources that it needs for further execution except for a processor. It is normally held in a ready queue until a processor becomes available.



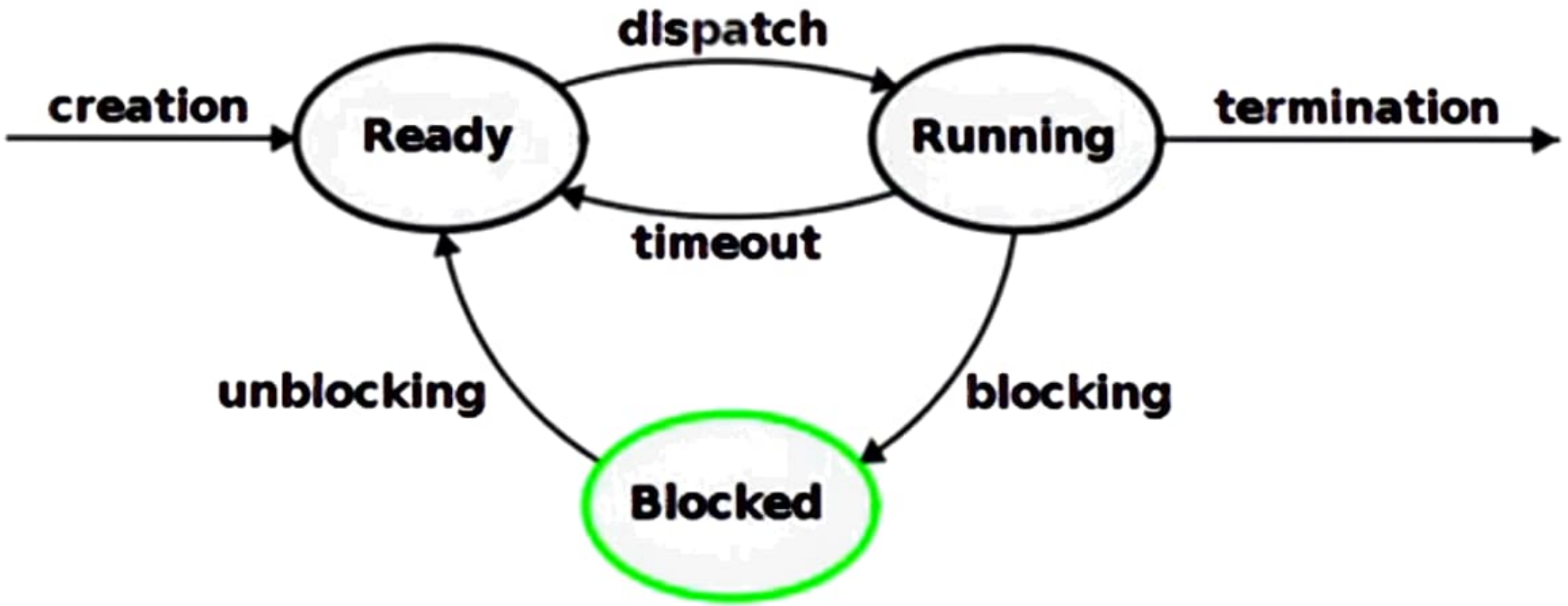
Running

A process in the running state has all of the resources that it needs for further execution, including a processor.



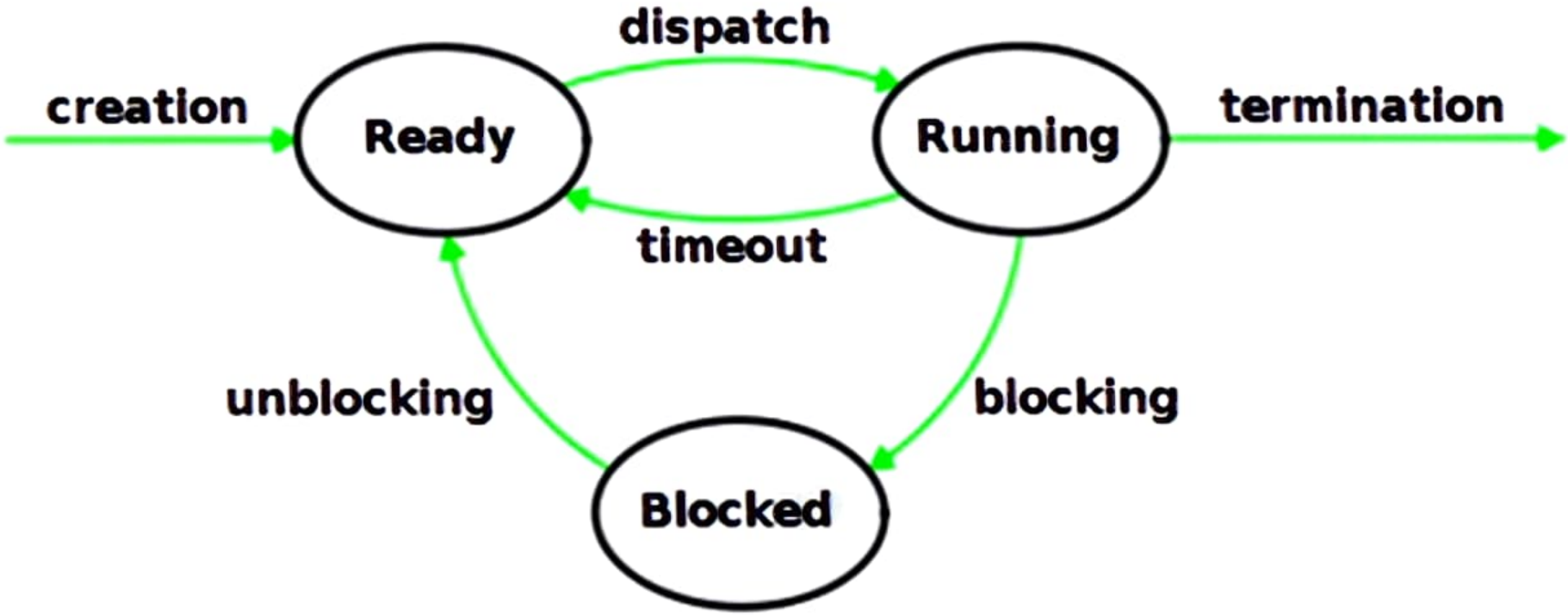
Blocked/Waiting

A process that needs some resource other than a processor for further execution is in a blocked state. It is usually placed in a queue waiting for the needed resource.

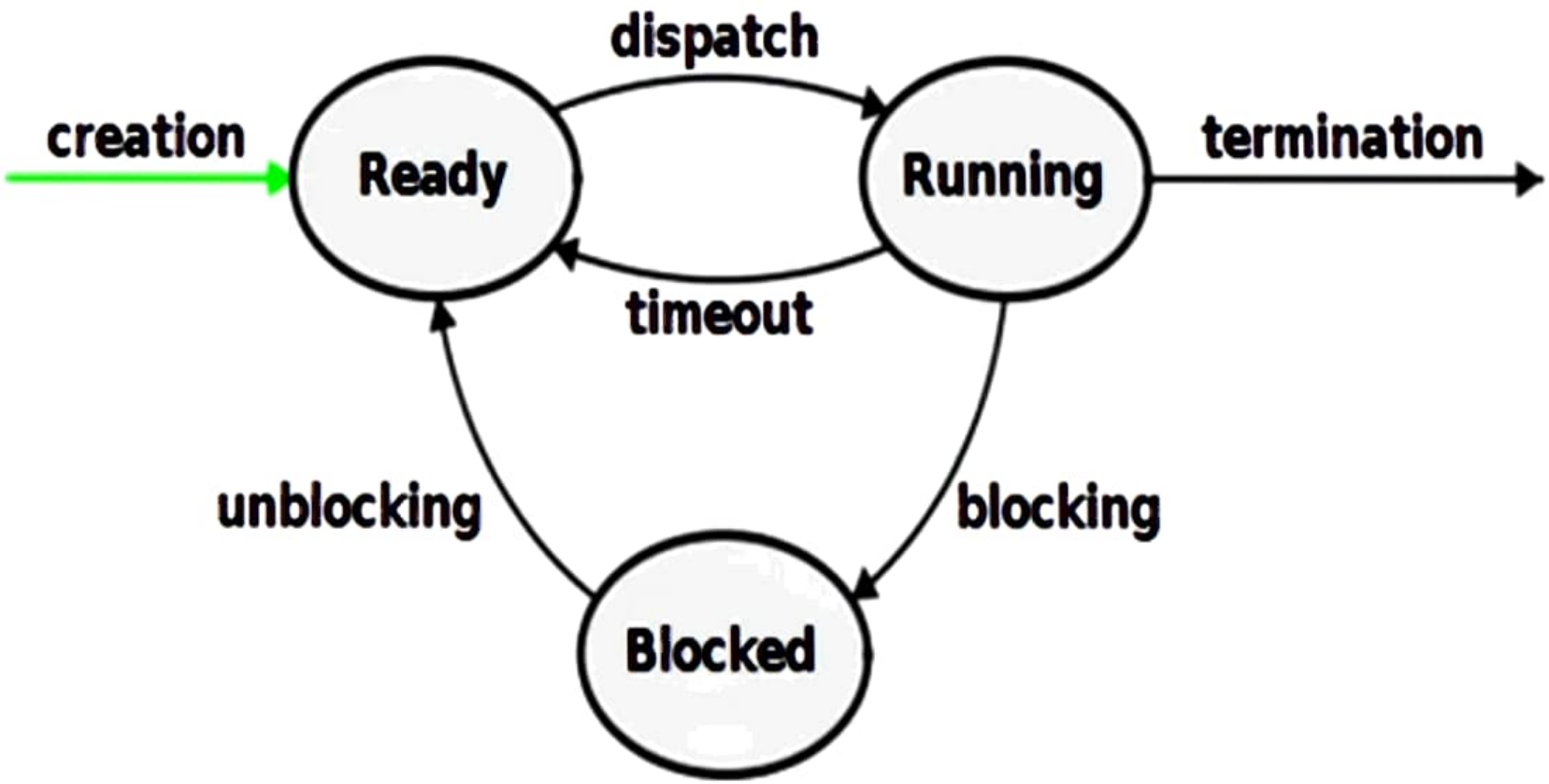


Transitions

The transitions of a process represent changes of its execution state. The transitions can be described in terms of their causes and the resulting actions taken by the operating system. An active process is normally in one of the six states in the diagram. The arrows show how the process changes states.



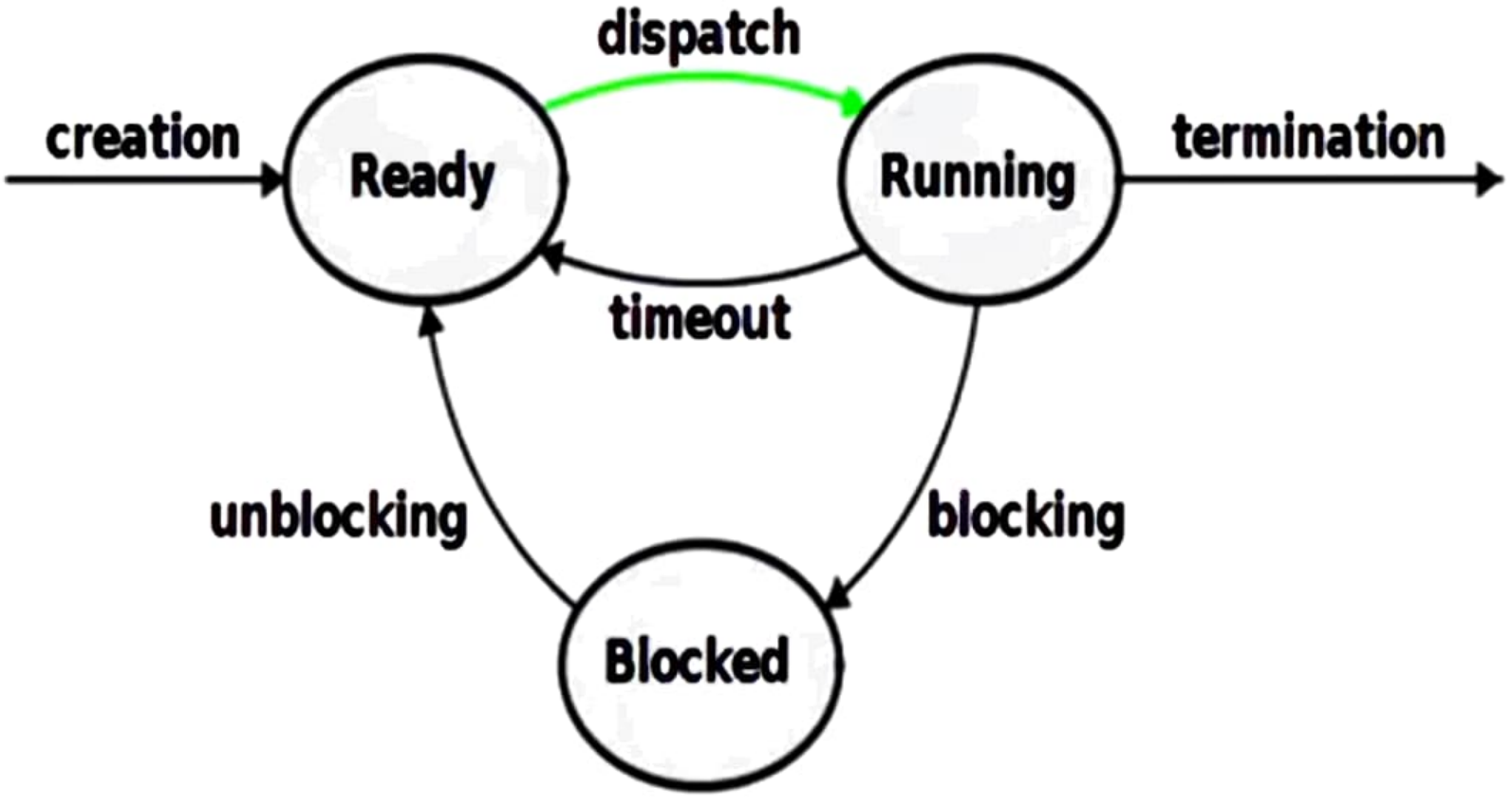
Creation


dispatch
creation
Ready
Running
termination
timeout
unblocking
blocking
Blocked

- **Cause**
- The creation transition is caused by a system call instruction for loading a program.

- **Action**
- A process control block is created for the program. It is initialized so that the process starts with cleared registers and PC set to the program's start (main) address. Usually the operating system sets up three open files: standard input, standard output, and standard error.

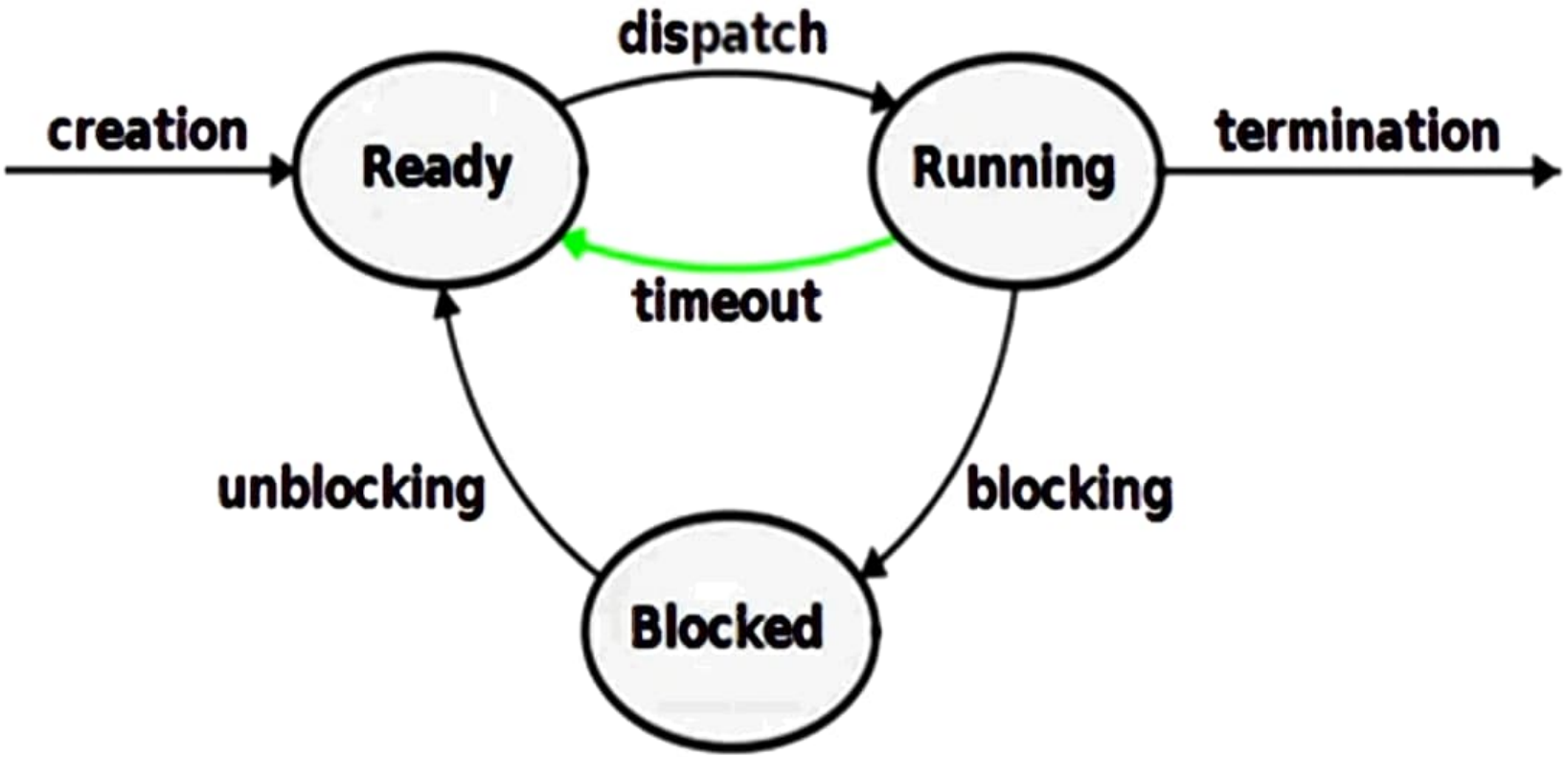
Dispatch


dispatch
creation
Ready
Running
termination
timeout
unblocking
blocking
Blocked

- **Cause**
- A process is dispatched when a processor is free to execute the process and the operating system has scheduled the process to run next. *Scheduling* involves selecting one of the ready processes to run next. The choice is often based on which ready process has gone the longest time since it last had a running execution status, but the choice may also involve prioritization of processes.

- **Action**
- Saved information about the process's register and PC contents is loaded into the processor. The PC contents are typically loaded by executing a jump instruction which, in effect, resumes execution of process code from where it left off.

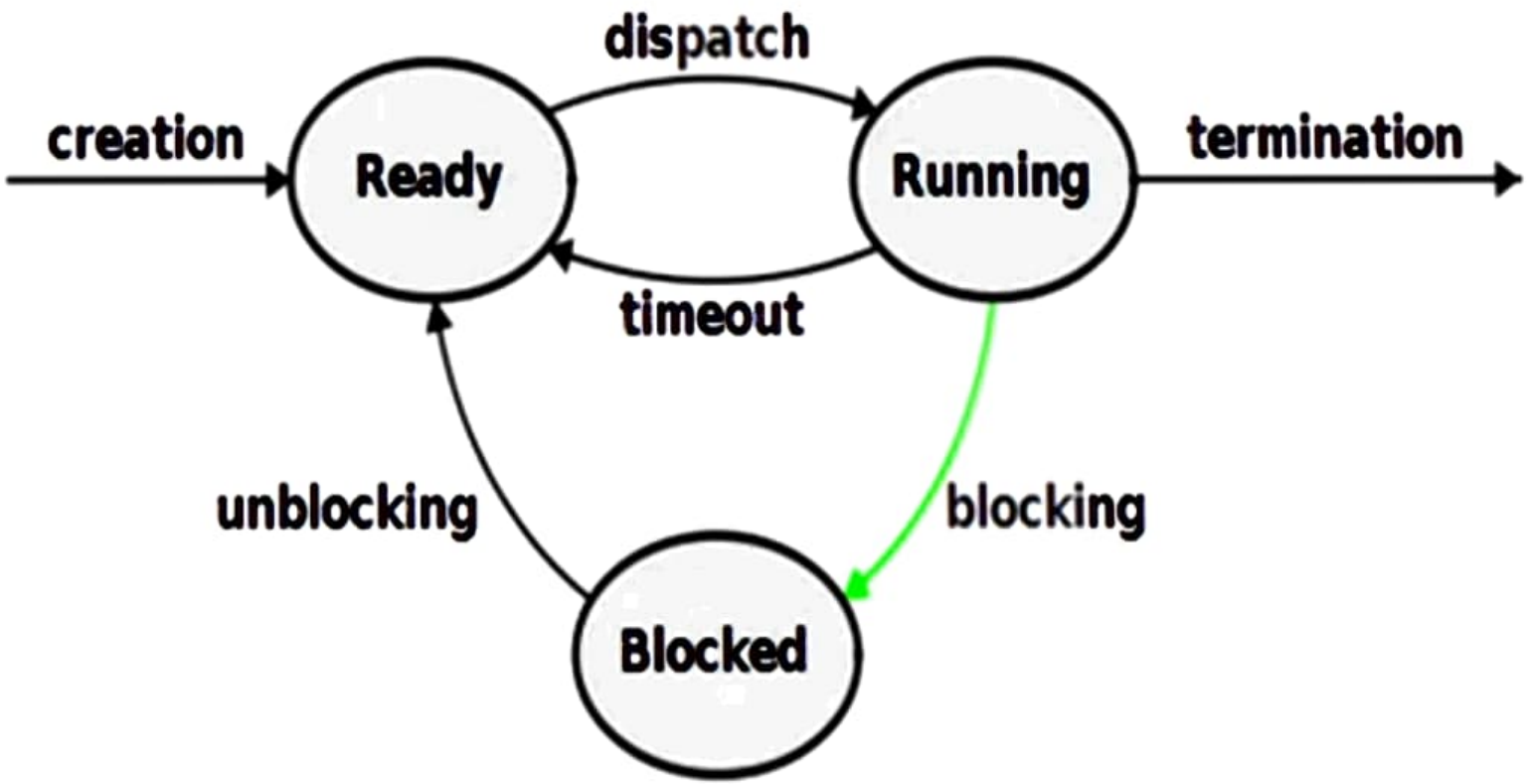
Timeout


dispatch
creation
Ready
Running
termination
timeout
unblocking
blocking
Blocked

- **Cause**
- A timeout is triggered by an external interrupt from a timer device.

- **Action**
- Information about the process's register and PC contents is saved into the PCB for the process. The process then goes into the ready state, where it enters a queue with other ready processes. The operating system will the schedule one of the ready processes and dispatch it.

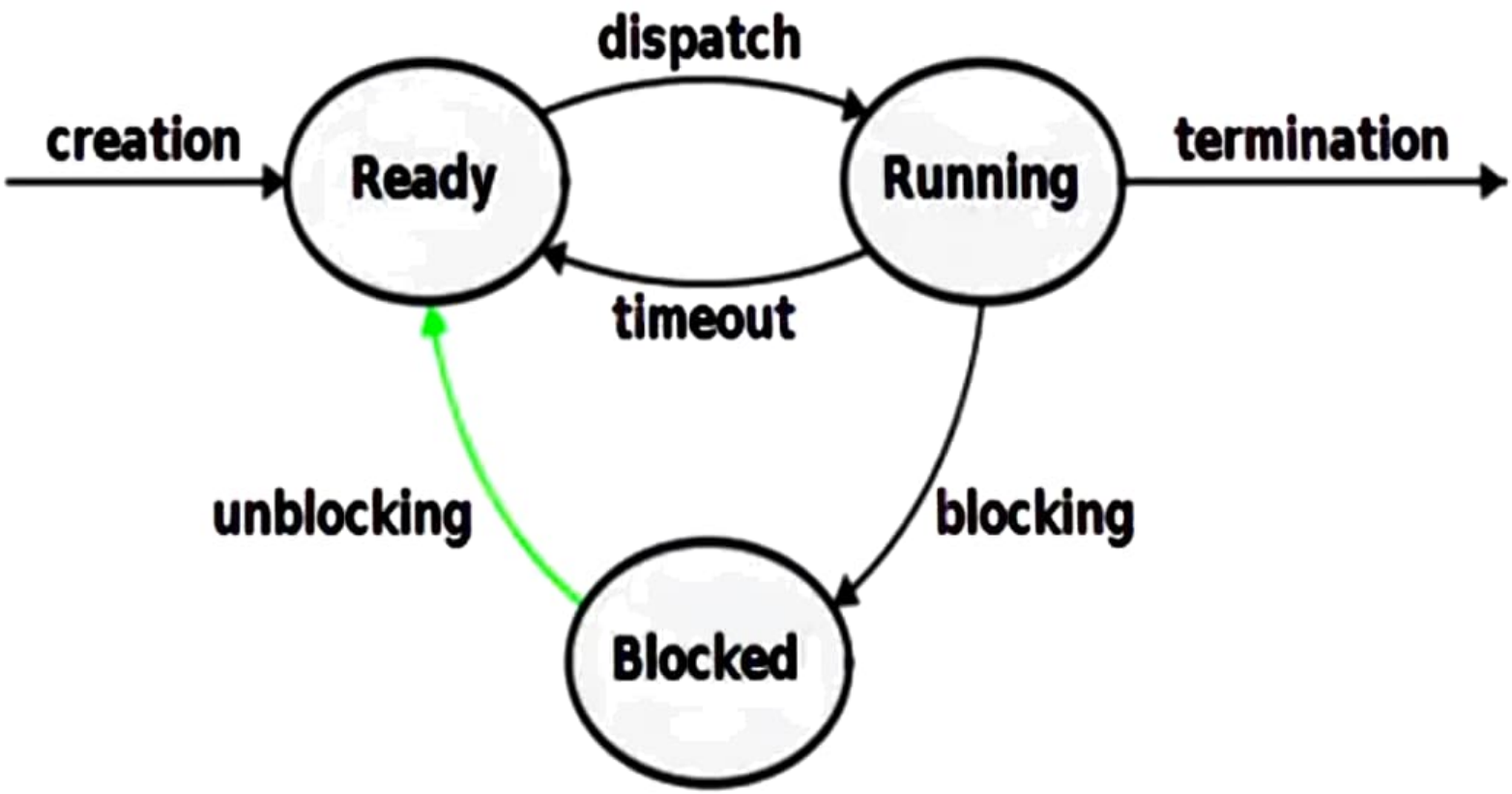
Blocking


dispatch
creation
Ready
Running
termination
timeout
unblocking
blocking
Blocked

- **Cause**
- A blocking transition is caused by the process making an operating system request (syscall) that must be satisfied before it can continue executing. The most common type of request is a request for input.

- **Action**
- The operating system will initiate an action to satisfy the request. For example, for file input from a disk, the operating system will send a signal to the disk initiating the fetch of a block from the disk. The process is put into a blocked state, where it cannot execute until its request is satisfied.

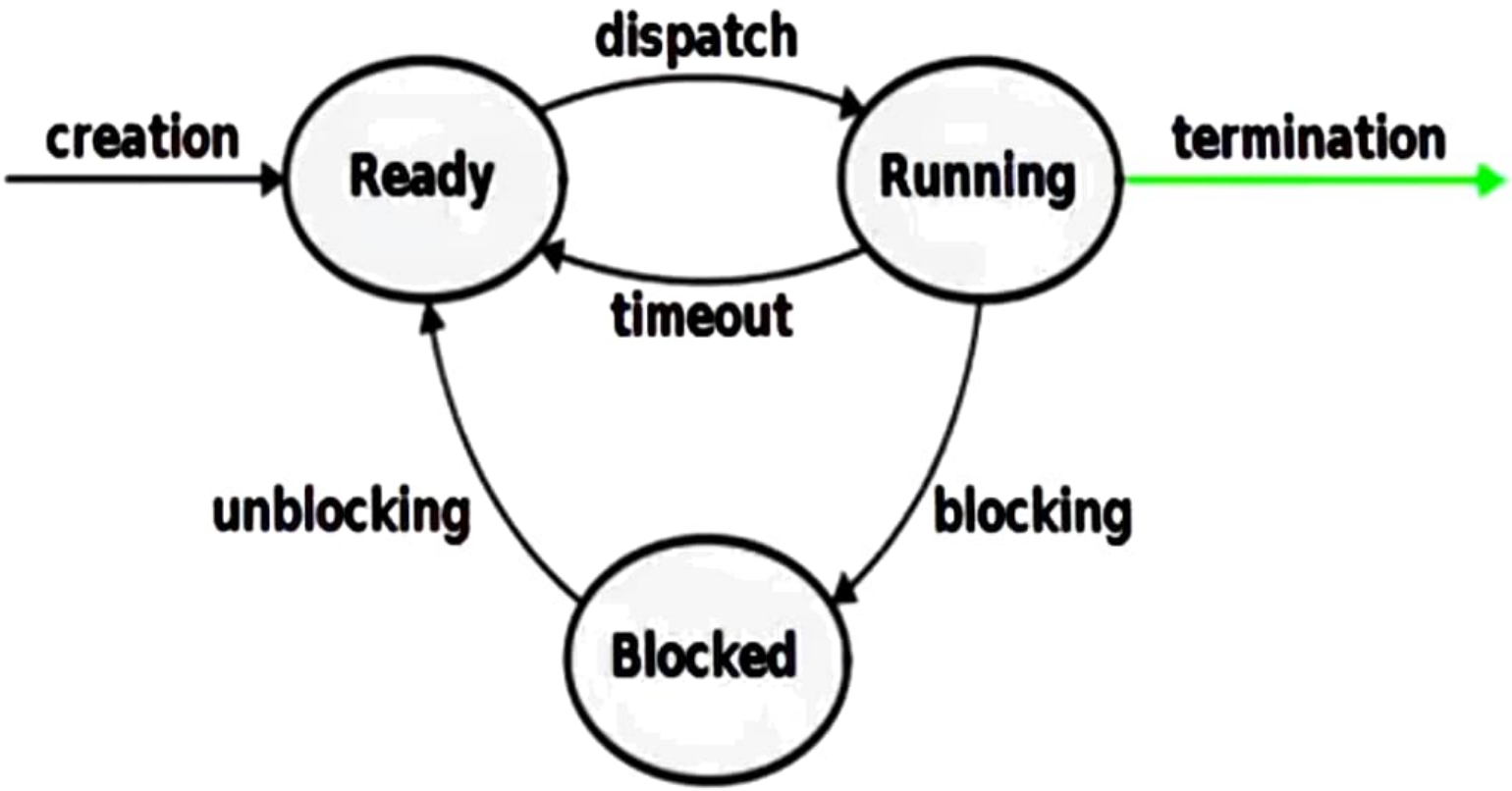
Unblocking


dispatch
creation
Ready
Running
termination
timeout
unblocking
blocking
Blocked

- **Cause**
- The unblocking transition is triggered by satisfaction of the request that lead to blocking. For example, if a process requested file input from a disk, the satisfaction will occur several milliseconds later when the disk sends an external interrupt indicating that it is ready to transfer the requested block.

- **Action**
- After the operating system has handled the request satisfaction it puts the process into the ready state, entering it into the ready queue. In the file read example, handling the request means storing the block contents in a file structure for the process.

Termination



- **Cause**
- The termination transition may be triggered by an exit syscall from the process (normal termination) or by a processor exception (abnormal termination).

- **Action**
- The operating system frees up any resources used by the process. If the termination is abnormal an error message is displayed.

Process Control Block

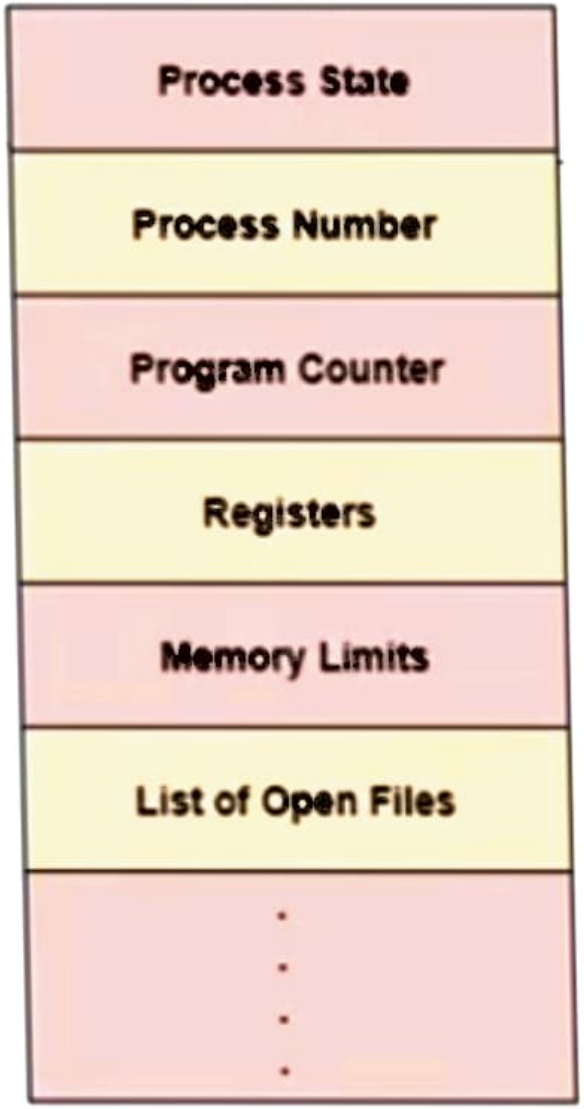
- Process Control Block is a data structure that contains information of the process related to it. The process control block is also known as a task control block, entry of the process table, etc.
- It is very important for process management as the data structuring for processes is done in terms of the PCB. It also defines the current state of the operating system.

- **Structure of the Process Control Block**
- The process control stores many data items that are needed for efficient process management. Some of these data items are explained with the help of the given diagram

Process Control Block

- Process Control Block is a data structure that contains information of the process related to it. The process control block is also known as a task control block, entry of the process table, etc.
- It is very important for process management as the data structuring for processes is done in terms of the PCB. It also defines the current state of the operating system.

- **Structure of the Process Control Block**
- The process control stores many data items that are needed for efficient process management. Some of these data items are explained with the help of the given diagram



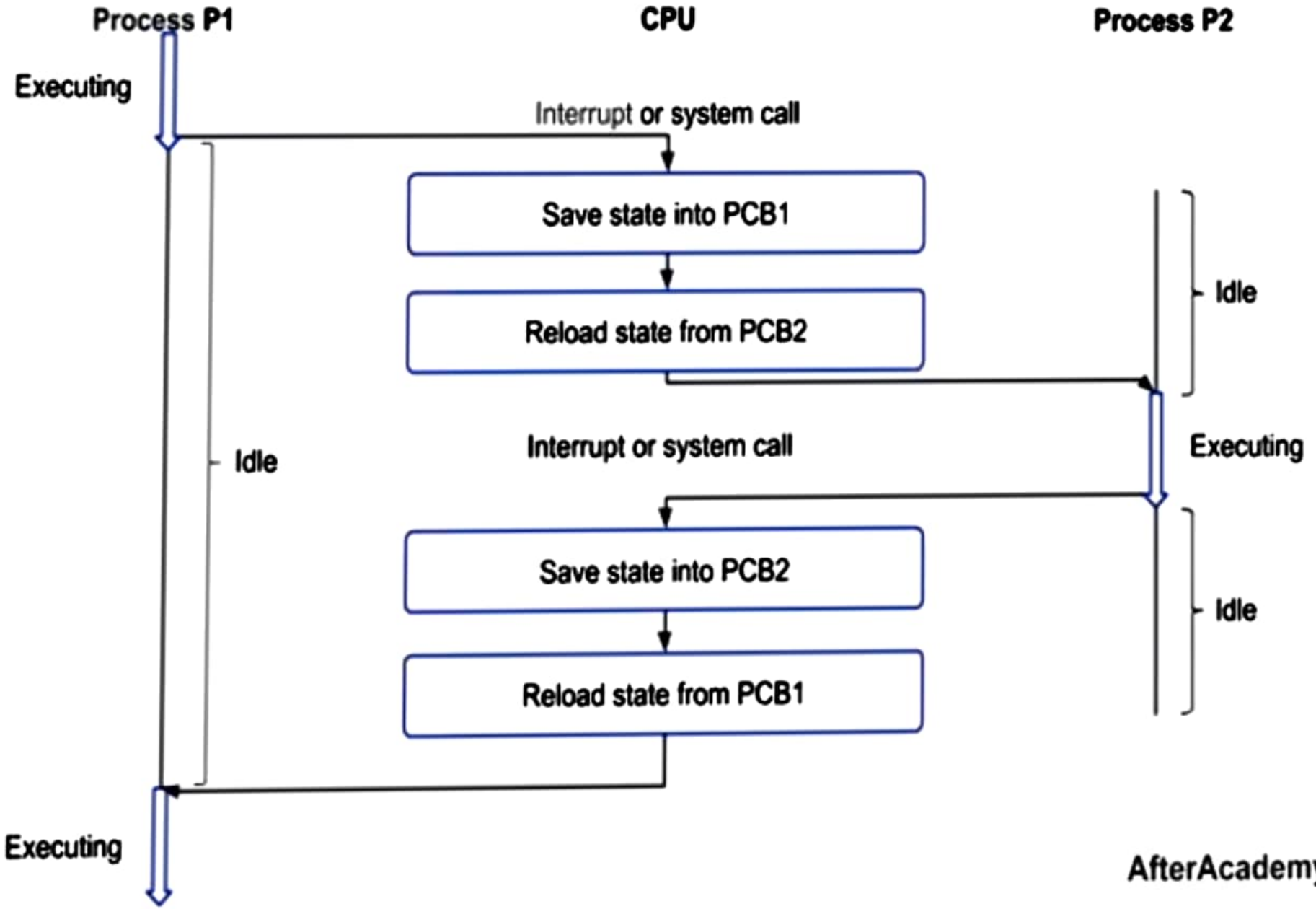
Process Control Block (PCB)

- **Process State**
- This specifies the process state i.e. new, ready, running, waiting or terminated.
- **Process Number**
- This shows the number of the particular process.
- **Program Counter**
- This contains the address of the next instruction that needs to be executed in the process.
- **Registers**
- This specifies the registers that are used by the process. They may include accumulators, index registers, stack pointers, general purpose registers etc.
- **Memory limits** – This field contains the information about memory management system used by operating system. This may include the page tables, segment tables etc.
- **Open files list** – This information includes the list of files opened for a process.

Context Switching

- Context Switching involves storing the context or state of a process so that it can be reloaded when required and execution can be resumed from the same point as earlier. This is a feature of a multitasking operating system and allows a single CPU to be shared by multiple processes.

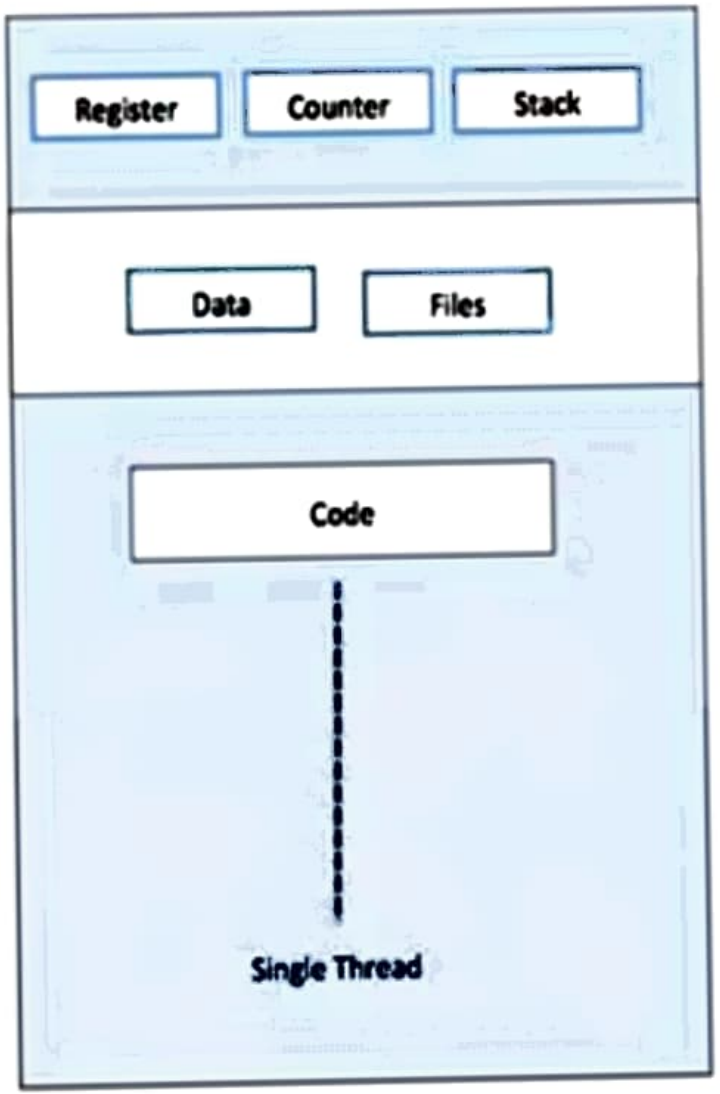
- A diagram that demonstrates context switching is as follows −
- In the following diagram, initially Process 1 is running. Process 1 is switched out and Process 2 is switched in because of an interrupt or a system call. Context switching involves saving the state of Process 1 into PCB1 and loading the state of process 2 from PCB2. After some time again a context switch occurs and Process 2 is switched out and Process 1 is switched in again. This involves saving the state of Process 2 into PCB2 and loading the state of process 1 from PCB1.


Process P1
CPU
Process P2
Executing
Interrupt or system call
Save state into PCB1
Reload state from PCB2
Idle
Idle
Executing
Interrupt or system call
Save state into PCB2
Reload state from PCB1
Idle
Executing
AfterAcademy

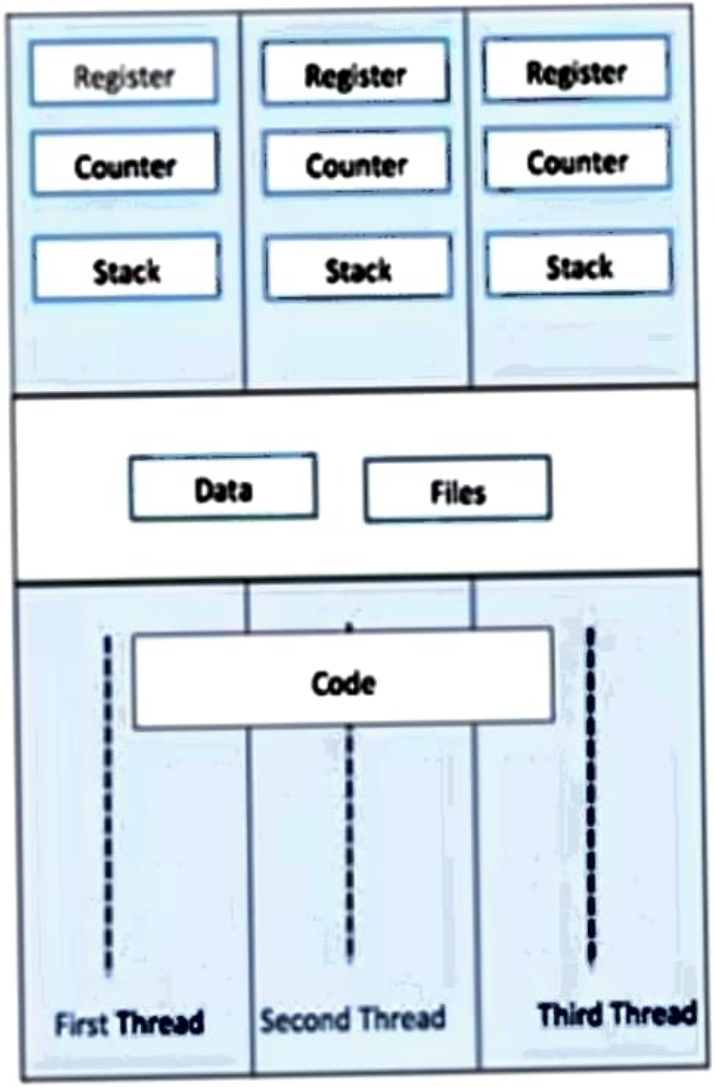
Threads

- A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.
- A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.
- A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

- Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server.
- They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.



Single Process P with single thread



Single Process P with three threads

Advantages of Thread

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

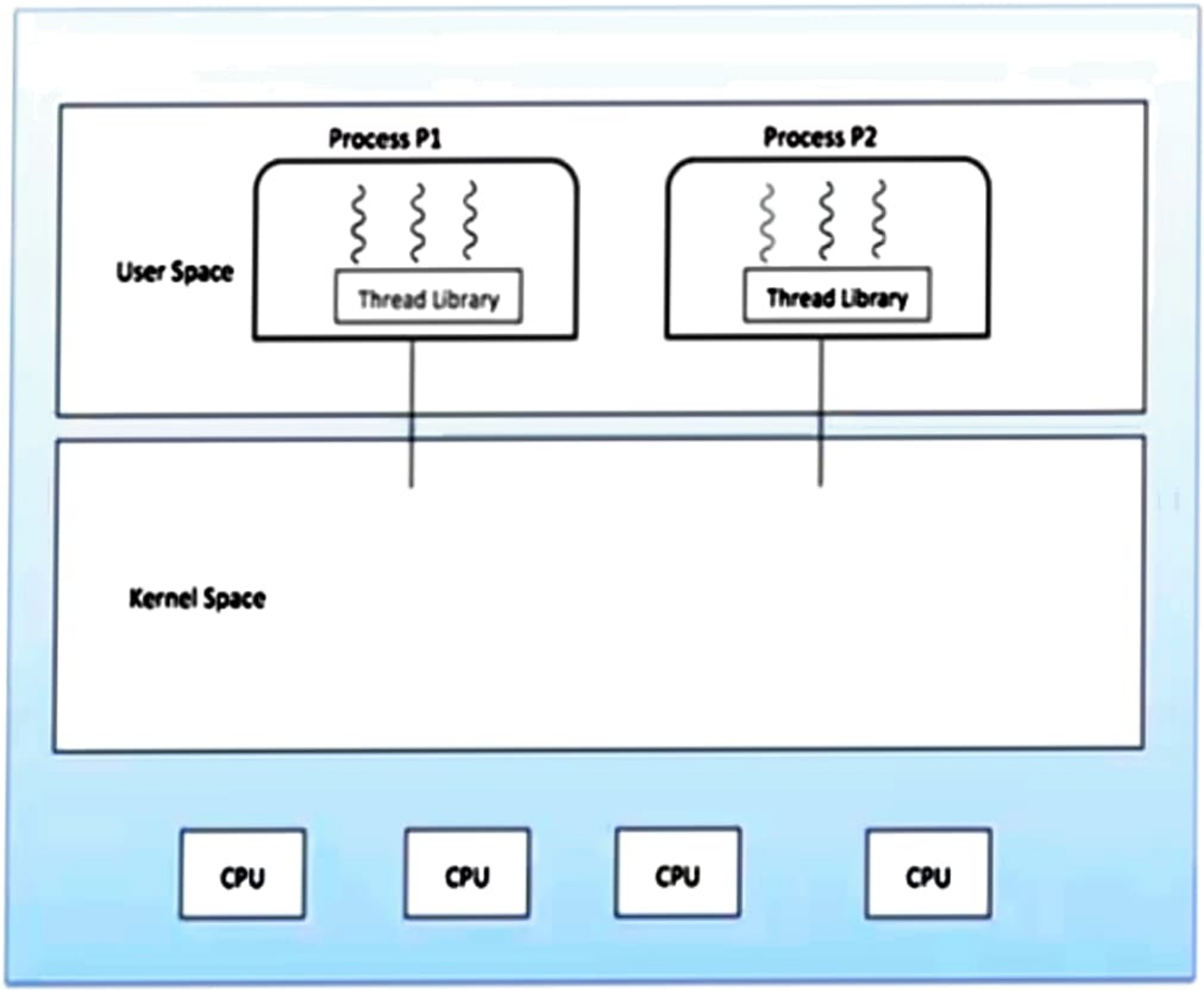
Types of Thread

- Threads are implemented in following two ways –

- **User Level Threads** – User managed threads.
- **Kernel Level Threads** – Operating System managed threads acting on kernel, an operating system core

User Level Threads

- In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.


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

Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

- In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.
- The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

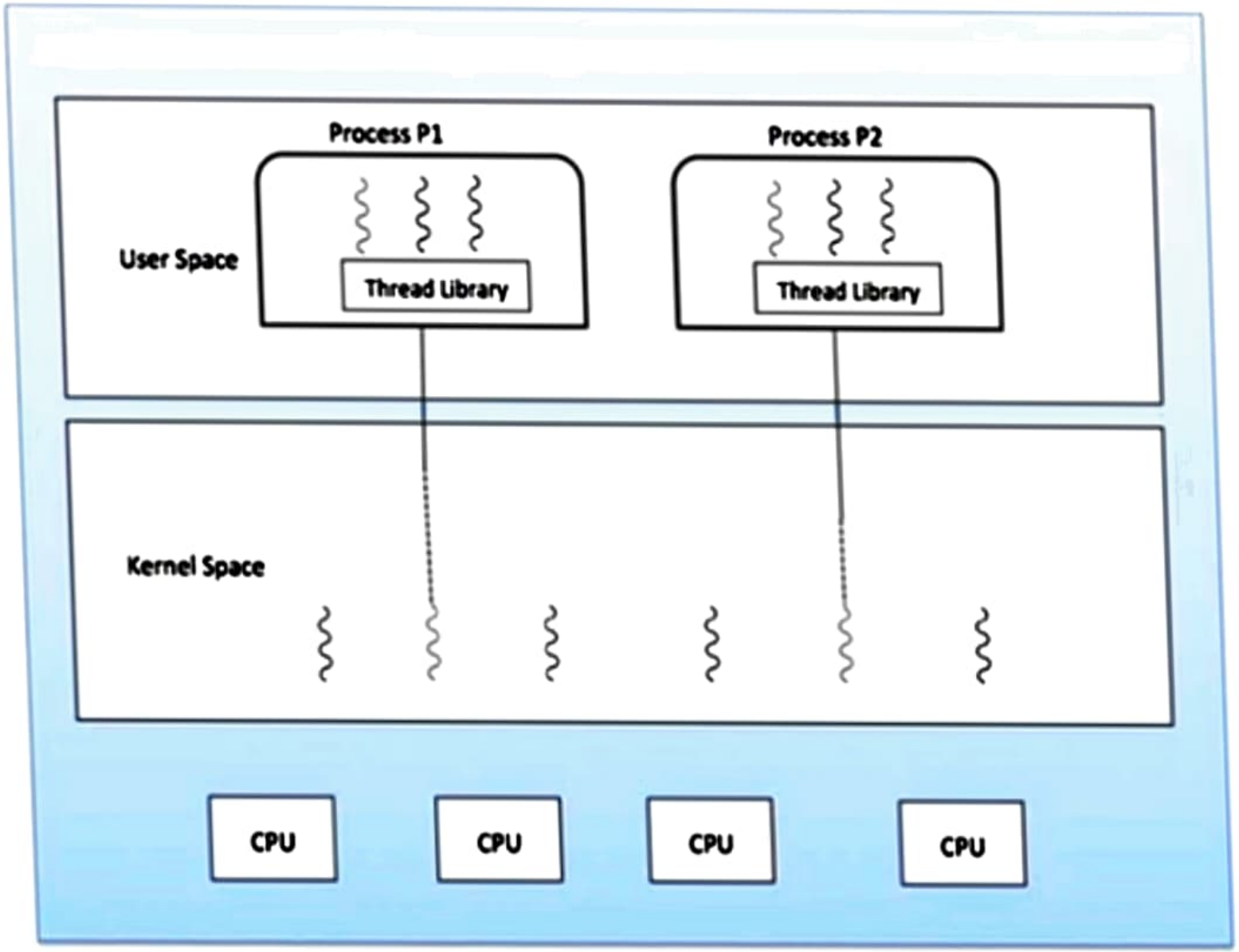
Multithreading

- Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Many to One Model

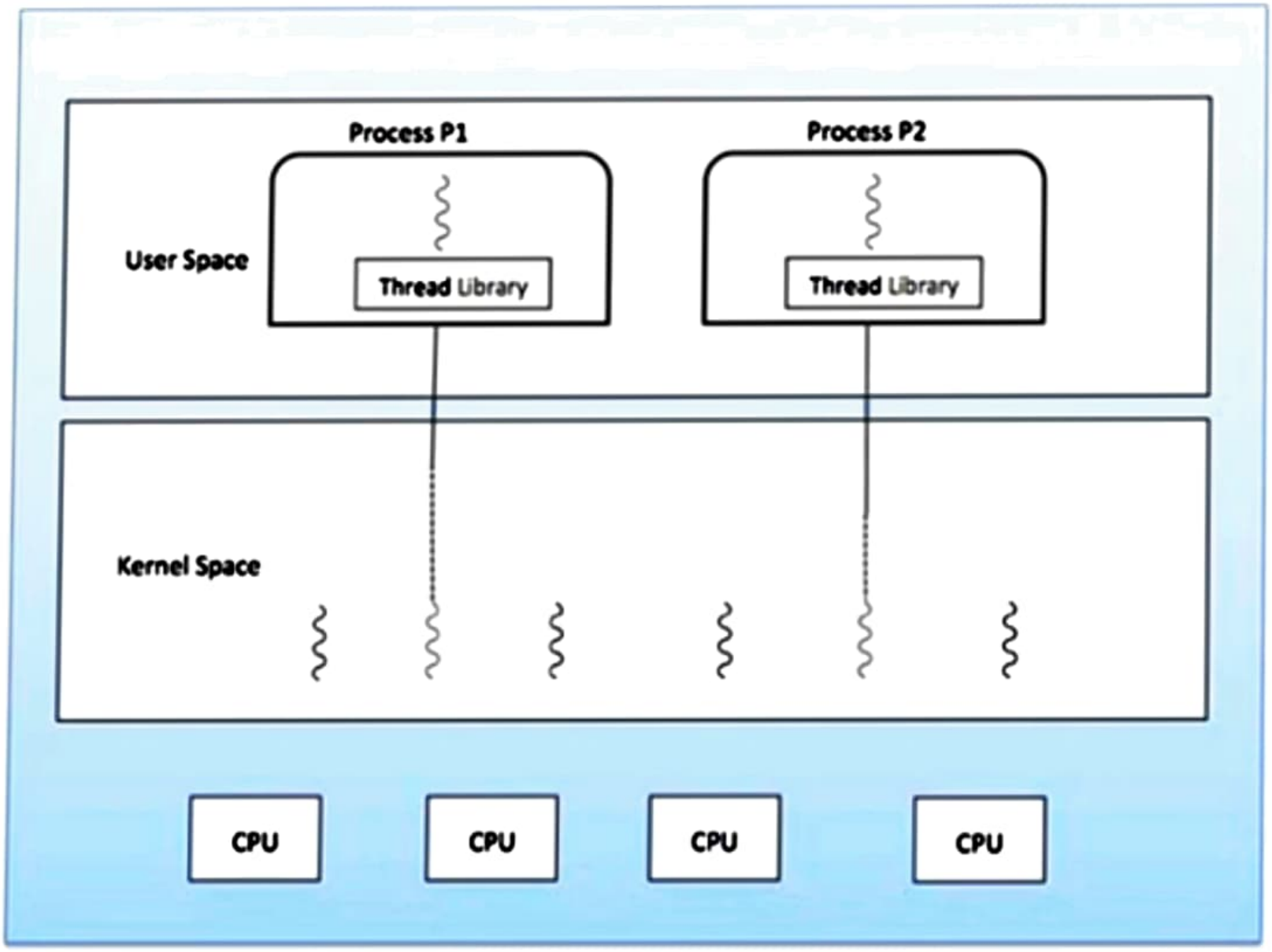
- Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.
- If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.


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

One to One Model

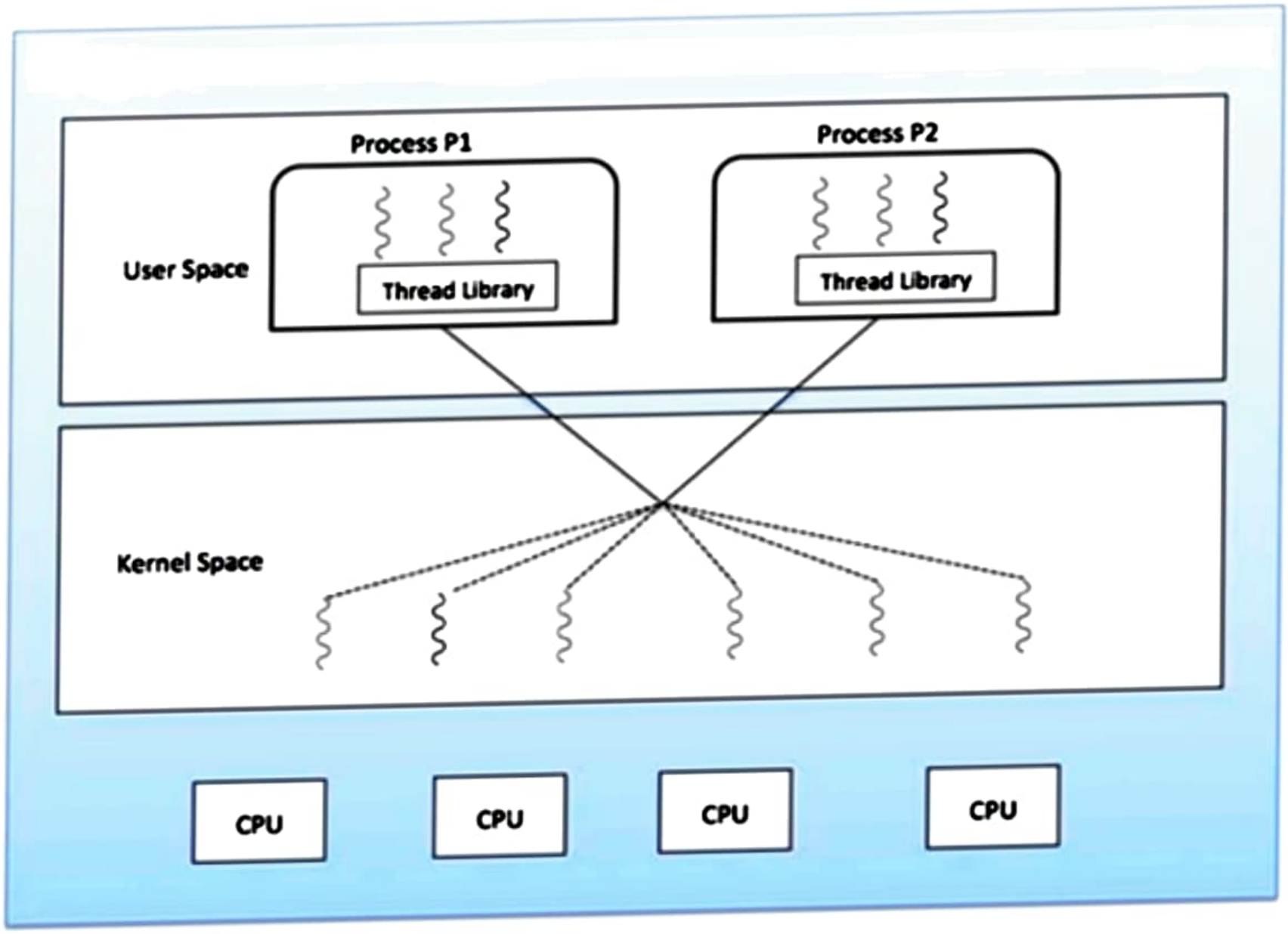
- There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

- Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.


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.
- The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 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.


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

Difference between User-Level & Kernel-Level Thread

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.

Process Scheduling

- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

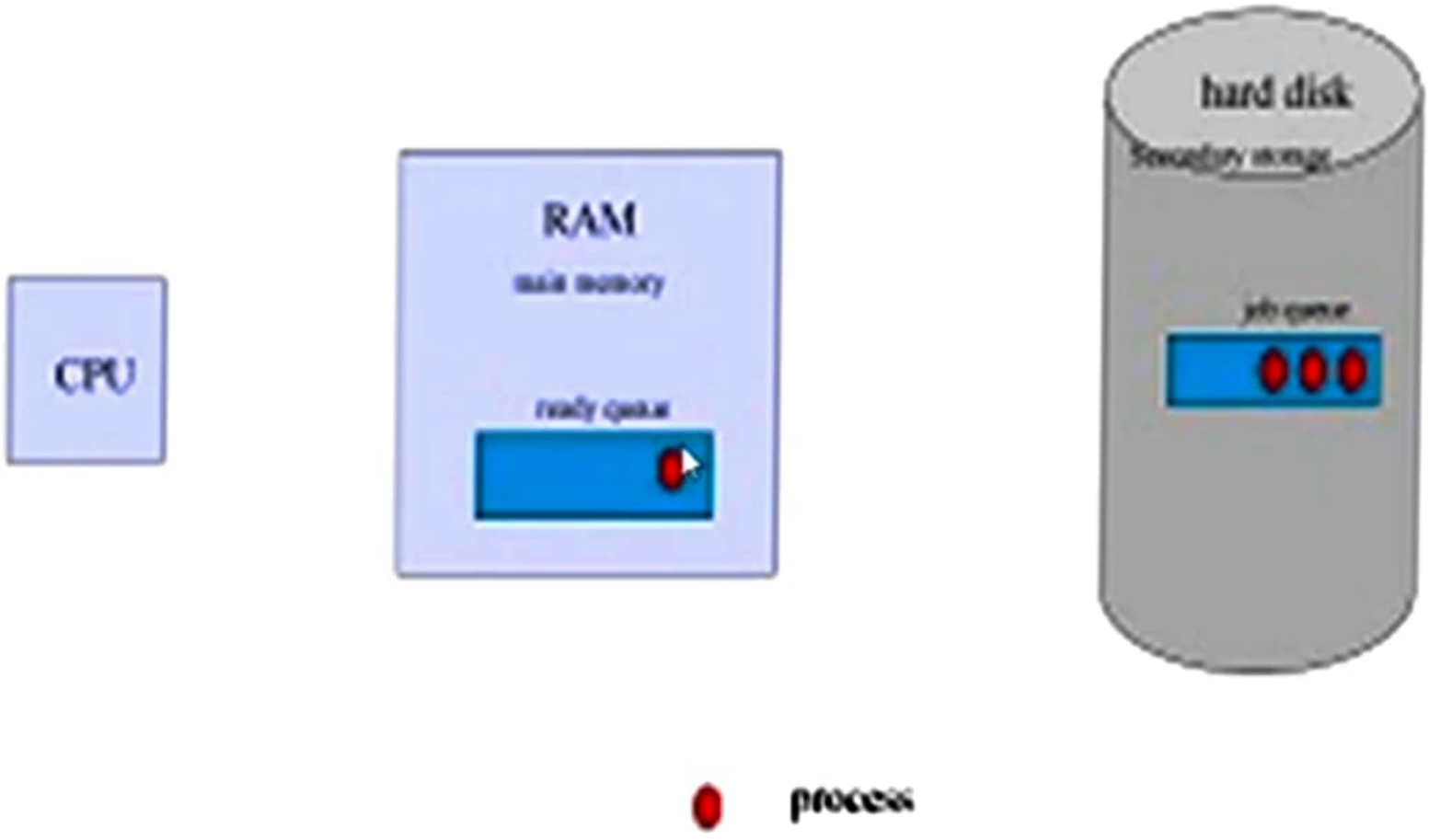
Process Scheduling Objectives

- Fairness and efficiency
- Throughput
- Scheduling Overhead Minimization
- Graceful Degradation Under Heavy Load Conditions

Schedulers

- Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

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


hard disk
RAM
CPU
process

Long Term Scheduler

- It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.
- The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

- It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.
- Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers

Medium Term Scheduler

- Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.
- A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage.
- This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix

Scheduling Criteria

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.

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).

Scheduling Algorithms

- A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

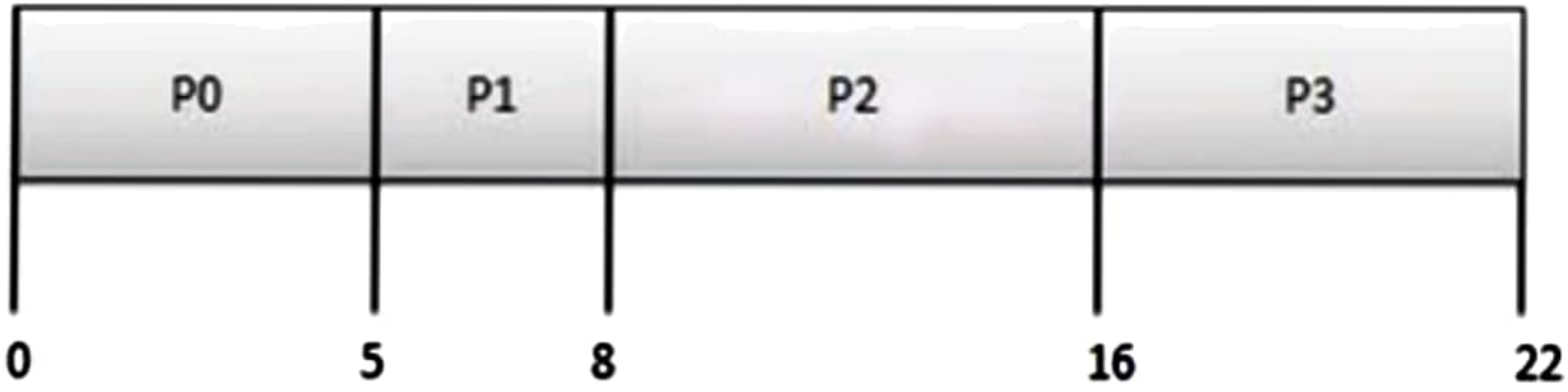
- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next/First (SJN/F) Scheduling
- Round Robin(RR) Scheduling

- These algorithms are either **non-preemptive or preemptive**.
- Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time.
- whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



- **Advantages of FCFS**
- Simple
- Easy
- First come, First serve

- **Disadvantages of FCFS**
- The scheduling method is non preemptive, the process will run to the completion.
- Due to the non-preemptive nature of the algorithm, the problem of starvation may occur.
- Although it is easy to implement, but it is poor in performance since the average waiting time is higher as compare to other scheduling algorithms.
- 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.

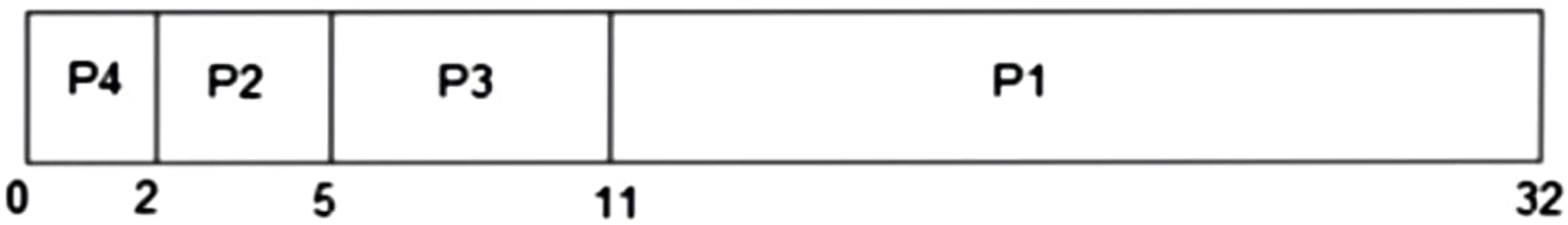
Shortest Job Next/First (SJN/F)

- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :



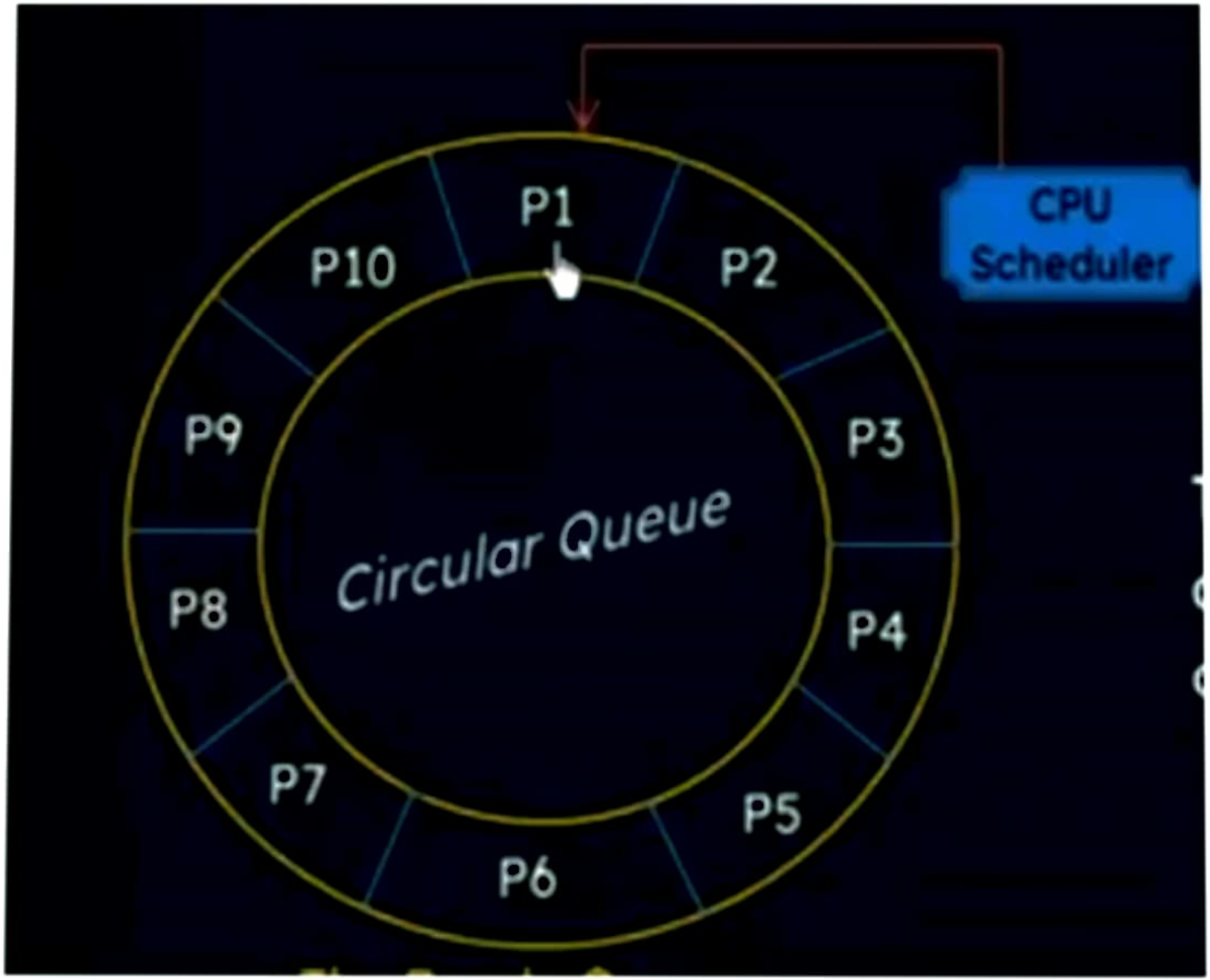
Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

- **Advantages of SJF**
- Maximum throughput
- Minimum average waiting and turnaround time

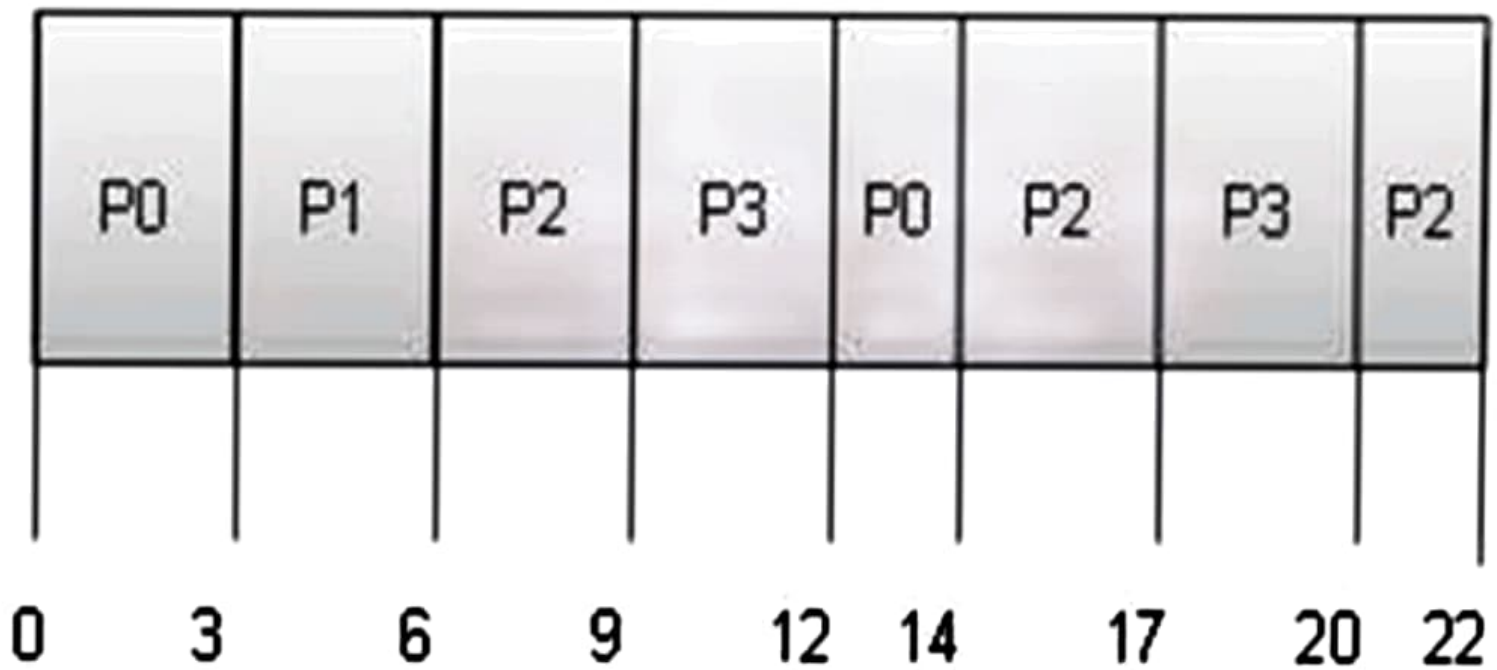
- **Disadvantages of SJF**
- May suffer with the problem of starvation
- It is not implementable because the exact Burst time for a process can't be known in advance.

Round Robin Scheduling

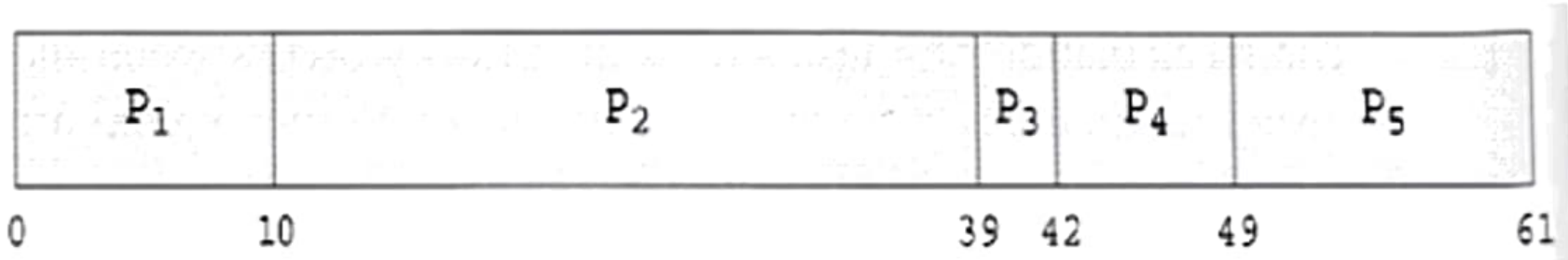
- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.


P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
Circular Queue
CPU
Scheduler

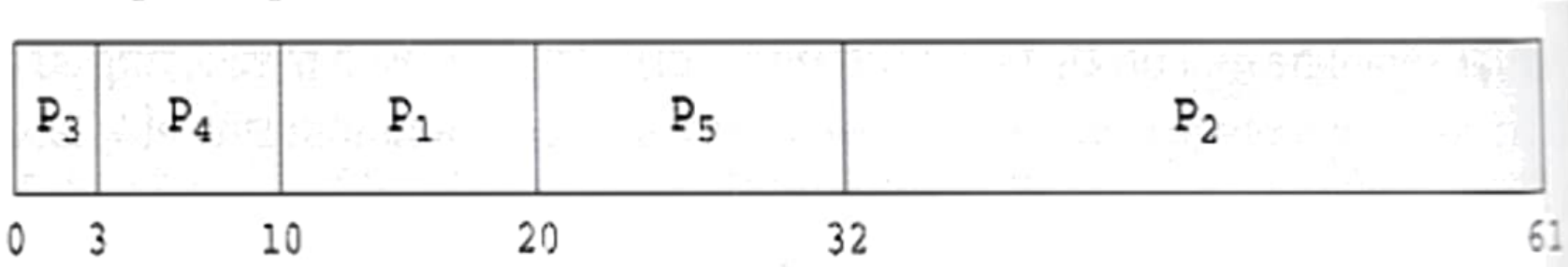
Quantum = 3



FCFS:



Non-preemptive SJF:



Round Robin: