KDKCE,Nagpur

Department of Information Technology

Operating System

UNIT-2

Process management is a core function of an Operating System (OS). It deals with creating, scheduling, and coordinating processes to ensure efficient CPU utilization and smooth system performance.
Key points to understand:

- **Single-tasking systems**: Easy to manage since only one process runs at a time.
- **Multiprogramming/multitasking systems:** More complex, as multiple processes need to share the CPU efficiently.
- **Resource sharing:** Active processes may share memory and other resources, requiring careful management.
- **Process synchronization**: Necessary when processes interact or communicate to avoid conflicts.

Process Control Block

Process Control Block (PCB)

A Process Control Block (PCB) is a **data structure** used by the operating system to **keep track of process information and manage execution**.

- **Each process** is given a unique Process ID (PID) for identification.
- **The PCB stores details** such as process state, program counter, stack pointer, open files, and scheduling info.

- **During a state transition**, the OS updates the PCB with the latest execution data.
- **It also includes register** values, CPU quantum, and process priority.
- **The Process Table is an array** of PCBs that maintains information for all active processes.

Structure of the Process Control Block

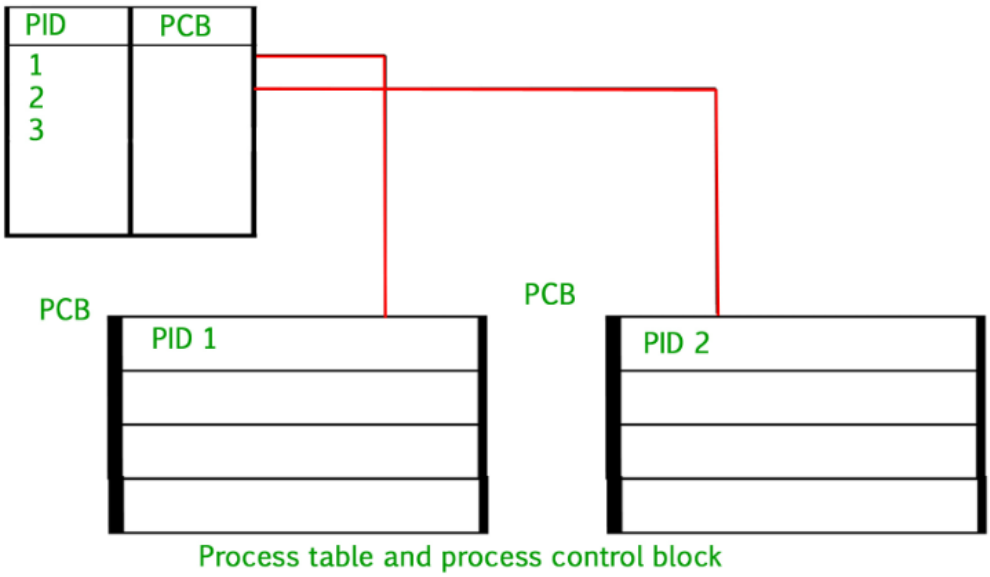
A Process Control Block (PCB) is a data structure used by the operating system to manage information about a process. The process control keeps track of many important pieces of information needed to manage processes efficiently. The diagram helps explain some of these key data items.

Key information stored in a PCB

- **Process State:** The current state of the process (e.g., new, running, waiting, ready).
- **Process ID:** A unique identifier for the process.
- **Program Counter:** The address of the next instruction to be executed.
- **CPU Registers:** The values of the CPU registers at the time the process was last run.
- **Memory Management Information:** Details about memory allocation, such as page tables, segment tables, base, and limit registers.
- **Scheduling Information:** The priority of the process and other scheduling-related data.
- **Accounting Information:** Records of resources used, like CPU time and total execution time.
- **I/O Status Information:** Information about I/O devices allocated to the process, such as the list of open files.

Process Table

A **process table** is a data structure maintained by the operating system to keep track of all active processes. It contains an entry for the Process Control Block (PCB) of each process, which stores essential information like process ID, state, program counter, CPU registers, memory usage and resource allocations.



Process table and process control block

Please note that the process table is not limited to only ID and reference to PCB, but in many OS designs, its main role is to map a PID to its corresponding PCB. Some implementations may store a few extra quick-access fields (like state, parent PID, or scheduling info) directly in the process table for efficiency, but the bulk of process information is always in the PCB.

Roles of Process Table & Process Control Block

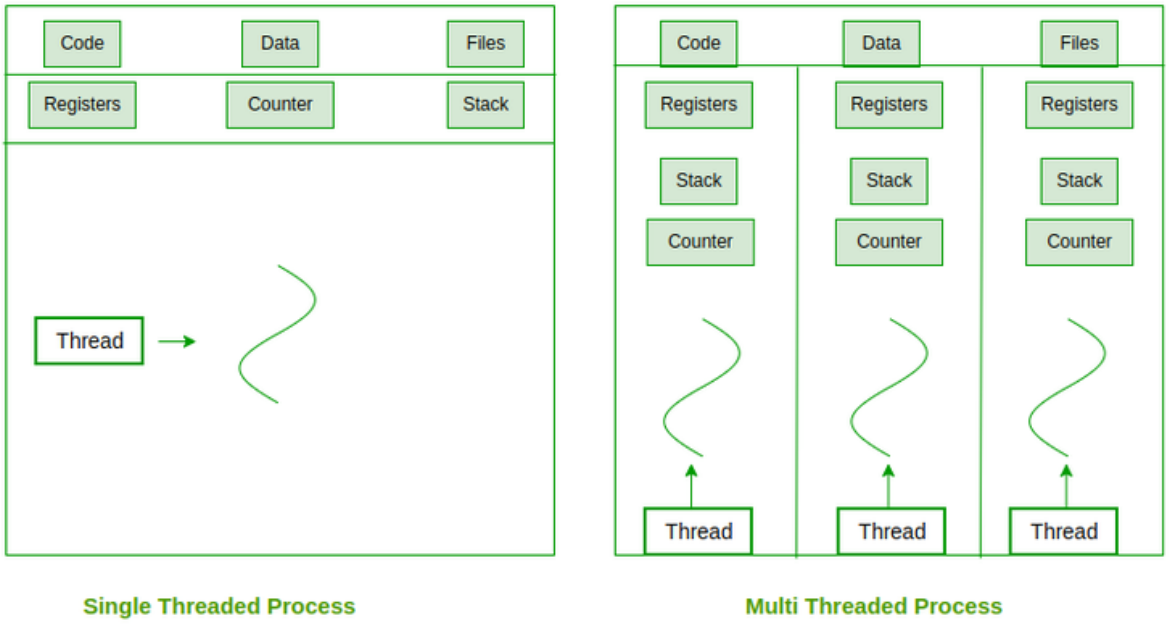
- **Help in Scheduling** : Contains priority levels, CPU burst times and arrival times. Assists the CPU scheduler in selecting the best process to run next.
- **Facilitate Context Switching :** Stores CPU registers, stack pointers and other context-related data.
- **Interrupt Handling:** The PCB also contains information about the interrupts that a process may have generated and how they were handled by the operating system.
- **Enable Inter-Process Communication (IPC) :** PCB Includes pointers or flags related to shared memory, pipes or message queues.
- **Support Resource Allocation** : Tracks allocated resources (memory, files, I/O devices) per process.
- **Essential for Process Synchronization and Deadlock Handling :** Maintains info like locks held/requested. Helps the OS detect and handle deadlocks or race conditions.
- **Security and Access Control** : Contains user ID, group ID, permission levels, etc. Enforces process-level access controls and restrictions.
- **Debugging and Monitoring** : Process Table is used by monitoring tools (like top, ps) to display process info. Helps in performance tuning and debugging applications.
- **Virtual Memory Management:** The PCB may contain information about a process virtual memory management, such as page tables and page fault handling.

- **Real-Time Systems:** Real-time operating systems may require additional information in the PCB, such as deadlines and priorities, to ensure that time-critical processes are executed in a timely manner.

Thread

A thread is a single sequence stream within a process. Threads are also called lightweight processes as they possess some of the properties of processes. Each thread belongs to exactly one process.

- In an operating system that supports multithreading, a process can consist of many threads.
- All threads belonging to the same process share code section, data section, and OS resources (e.g. open files and signals), but each thread has its own (thread control block) - thread ID, program counter, register set, and a stack.



Single Threaded Process Multi Threaded Process

Why Do We Need Threads (and Their Benefits)

Threads are needed in modern operating systems and applications because they:

- **Improve Application Performance**: Threads can run in parallel, making programs execute faster.
- **Increase Responsiveness**: Even if one thread is busy, another can return results or handle user actions immediately.
- **Enable Concurrency**: Multiple things can happen at once, such as background saving, formatting, and user input in Microsoft Word or Google Docs.
- **Simplify Communication**: Since threads share the same memory space, they can directly exchange data without special inter-process communication mechanisms.
- **Support Prioritization**: Like processes, threads can have priorities; the highest-priority thread gets scheduled first.

- **Efficient Context Switching**: Switching between threads takes less time than switching between processes because threads use the same address space.
- **Better Multiprocessor Utilization**: Threads from the same process can run on different processors simultaneously, speeding up execution.
- **Resource Sharing**: Threads within a process share code, data, and files, which saves resources.
- **Higher Throughput**: Dividing a process into multiple threads allows more jobs to finish per unit time.
- **Synchronization Support**: Since threads share resources, synchronization tools (locks, semaphores, etc.) ensure safe access to shared data.
- **Thread Management**: Each thread has a **Thread Control Block (TCB)** that stores its state, register values, and scheduling info for context switching.

Components of Threads

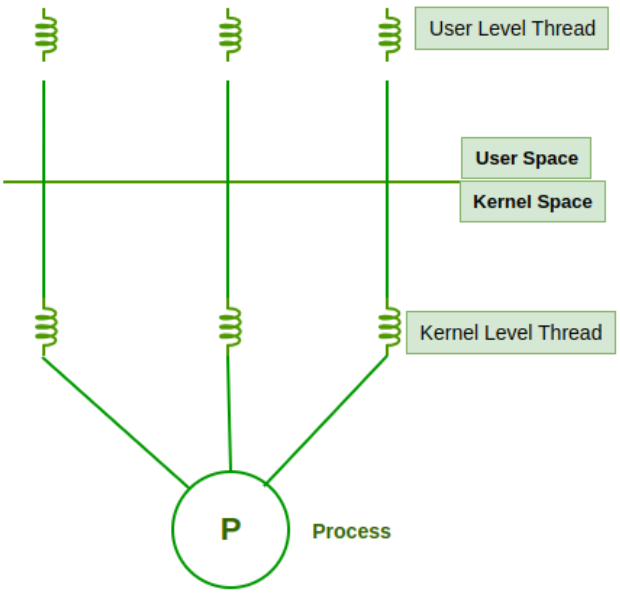
These are the basic components of the Operating System.

- **Stack Space:** Stores local variables, function calls, and return addresses specific to the thread.
- **Register Set:** Hold temporary data and intermediate results for the thread's execution.
- **Program Counter:** Tracks the current instruction being executed by the thread.

Types of Thread in Operating System

Threads are of two types. These are described below.

- User Level Thread
- Kernel Level Thread



Threads

User-Level Threads (ULTs)

- Managed entirely in user space using a thread library; the kernel is unaware of them.
- Switching between ULTs is fast since only program counter, registers, and stack need to be saved/restored.
- Do not require system calls for creation or management, making them lightweight.
- **Blocking Limitation**: If one thread makes a blocking system call, the entire process (all threads) is blocked.
- Scheduling is done by the application itself, which may not be as efficient as kernel-level scheduling.
- Cannot fully utilize multiprocessor systems because the kernel schedules processes, not individual user-level threads.

Kernel-Level Threads (KLTs)

- Managed directly by the operating system kernel; each thread has an entry in the kernel's thread table.
- The kernel schedules each thread independently, allowing true parallel execution on multiple CPUs/cores.
- Handles blocking system calls efficiently; if one thread blocks, the kernel can run another thread from the same process.
- Provides better load balancing across processors since the kernel controls all threads.
- Context switching is slower compared to ULTs because it requires switching between user mode and kernel mode.
- Implementation is more complex and requires frequent interaction with the kernel.
- Large numbers of threads may add extra load on the kernel scheduler, potentially affecting performance.

Sr. No	User Level Threads	Kernel Level Thread
1	User level thread are faster to create and manage.	Kernel level thread are slower to create and manage.
2	Implemented by a thread library at the user level.	Operating system support directly to Kernel threads.
3	User level thread can run on any operating system.	Kernel level threads are specific to the operating system.
4	Support provided at the user level called user level thread.	Support may be provided by kernel is called Kernel level threads.
5	Multithread application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Difference Between Process and Thread

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces. Threads are not independent of one another like processes are, and as a result, threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like a process, a thread has its own program counter (PC), register set, and stack space.

Sr. No.	Process	Threads
1	System calls involved in process	There is no system call involved
2	OS treats different processes differently	All user level threads treated as single task for OS
3	Different processes have different copies of data, files, code	Threads share same copy of code and data
4	Context switching is slower	Context switching is faster

5	Blocking a process will not block others	Blocking a thread will block entire process
6	Independent	Interdependent

Scheduling policies are categorized as **preemptive** (process can be interrupted, e.g., Round Robin) or **non-preemptive** (process runs to completion or waits, e.g., FCFS). Common algorithms include First-Come, First-Served (FCFS), Shortest Job First (SJF), Priority Scheduling, and Round Robin. They differ in how they handle process selection, with variations like Shortest Remaining Time First (SRTF) (preemptive SJF) and Multilevel Queue Scheduling offering more complex ways to manage processes.

Scheduling Algorithms

Types of Scheduling Queues

Job Queue (In Disk)

This queue is known as the job queue, it contains all the processes or jobs in the list that are waiting to be processed. Job: When a job is created, it goes into the job queue and waits until it is ready for processing.

- Contains all submitted jobs.
- Processes are stored here in a wait state until they are ready to go to the execution stage.
- This is the first and most basic state that acts as a default storage of new jobs added to a scheduling system.
- Long Term Scheduler Picks a process from Job Queue and moves to ready queue.

Ready Queue (In Main Memory)

The Stand-by queue contains all the processes ready to be fetched from the memory, for execution. When the process is initiated, it joins the ready queue to wait for the CPU to be free. The operating system assigns a process to the executing processor from this queue based on the scheduling algorithm it implements.

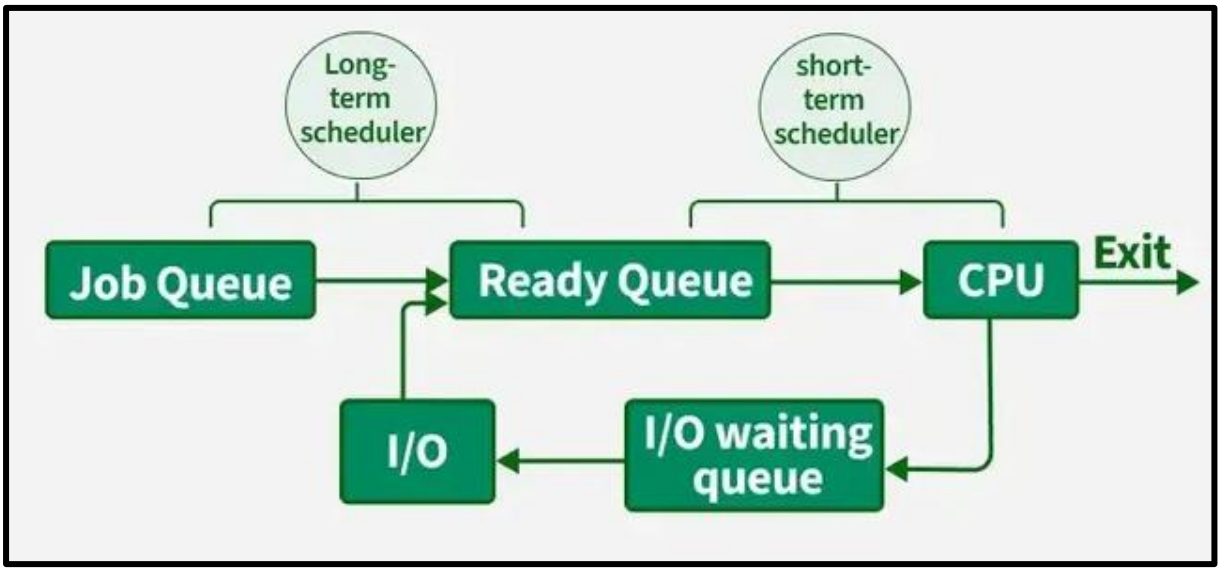
- Contains processes (mainly their PCBs) waiting for the CPU to execute various processes it contains.
- They are controlled using a scheduling algorithm like FCFS, SJF, or Priority Scheduling.
- Short Term Scheduler picks a process from Ready Queue and moves the selected process to running state.

Block or Device Queues (In Main Memory)

The processes which are blocked due to unavailability of an I/O device are added to this queue. Every device has its own block queue.

Flow of Movement in the above Queues

The below diagram shows movements of processes in different queues.



- All processes are initially in the Job Queue.
- A new process is initially put in the Ready queue by scheduler. 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:
  1) The process could issue an I/O request, and then be placed in a Device queue.
  2) The process could create a new subprocess and wait for its termination.
  3) The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Terminologies Used in CPU Scheduling

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

Turn Around Time = Completion Time – Arrival Time

- **Waiting Time(W.T):** Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time – Burst Time

Different Types of CPU Scheduling Algorithms

There are mainly two types of scheduling methods:

- **Preemptive Scheduling:** Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to the ready state.
- **Non-Preemptive Scheduling:** Non-Preemptive scheduling is used when a process terminates, or when a process switches from running state to waiting state.