CPU Scheduling: Scheduling criteria, Levels of Scheduling, Scheduling algorithms: Pre-emptive and Non Pre-emptive Scheduling Algorithm Multiple processor scheduling.

CPU Scheduling Criteria

Last Updated : 12 Mar, 2024

CPU scheduling is essential for the system's performance and ensures that processes are executed correctly and on time. Different CPU scheduling algorithms have other properties and the choice of a particular algorithm depends on various factors. Many criteria have been suggested for comparing CPU scheduling algorithms.

What is CPU scheduling?

CPU Scheduling is a process that allows one process to use the CPU while another process is delayed due to unavailability of any resources such as I / O etc, thus making full use of the CPU. In short, CPU scheduling decides the order and priority of the processes to run and allocates the CPU time based on various parameters such as CPU usage, throughput, turnaround, waiting time, and response time. The purpose of CPU Scheduling is to make the system more efficient, faster, and fairer.

Criteria of CPU Scheduling

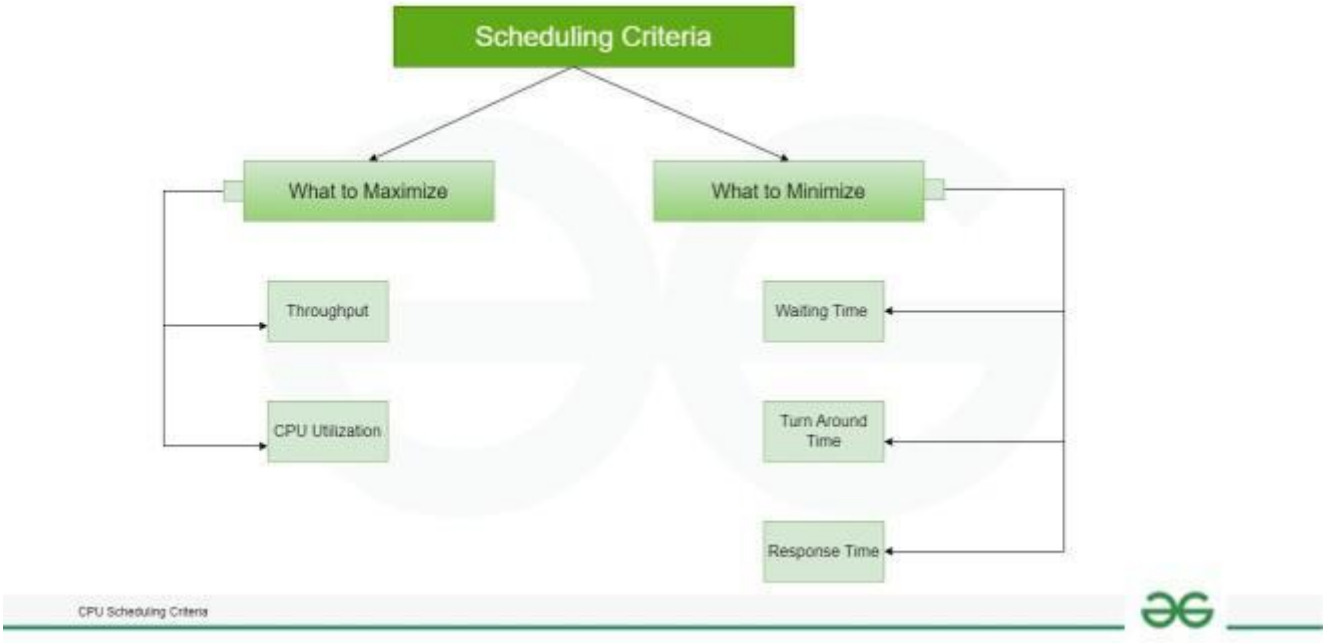
CPU Scheduling has several criteria. Some of them are mentioned below.

1. CPU utilization

The main objective of any CPU scheduling algorithm is to keep the CPU as busy as possible. Theoretically, CPU utilization can range from 0 to 100 but in a real-time system, it varies from 40 to 90 percent depending on the load upon the system.

2. Throughput

A measure of the work done by the CPU is the number of processes being executed and completed per unit of time. This is called throughput. The throughput may vary depending on the length or duration of the processes.



CPU Scheduling Criteria

3. Turnaround Time

For a particular process, an important criterion is how long it takes to execute that process. The time elapsed from the time of submission of a process to the time of completion is known as the turnaround time. Turn-around time is the sum of times spent waiting to get into memory, waiting in the ready queue, executing in CPU, and waiting for I/O.

Turn Around Time = Completion Time – Arrival Time.

4. Waiting Time

A scheduling algorithm does not affect the time required to complete the process once it starts execution. It only affects the waiting time of a process i.e. time spent by a process waiting in the ready queue.

Waiting Time = Turnaround Time – Burst Time.

5. Response Time

In an interactive system, turn-around time is not the best criterion. A process may produce some output fairly early and continue computing new results while previous results are being output to the user. Thus another criterion is the time taken from submission of the process of the request until the first response is produced. This measure is called response time.

Response Time = CPU Allocation Time(when the CPU was allocated for the first) – Arrival Time

6. Completion Time

The completion time is the time when the process stops executing, which means that the process has completed its burst time and is completely executed.

7. Priority

If the operating system assigns priorities to processes, the scheduling mechanism should favor the higher-priority processes.

8. Predictability

A given process always should run in about the same amount of time under a similar system load.

Importance of Selecting the Right CPU Scheduling Algorithm for Specific Situations

It is important to choose the correct CPU scheduling algorithm because different algorithms have different priorities for different CPU scheduling criteria.Different algorithms have different strengths and weaknesses. Choosing the wrong CPU scheduling algorithm in a given situation can result in suboptimal performance of the system.

Example: Here are some examples of CPU scheduling algorithms that work well in different situations.

Round Robin scheduling algorithm works well in a time-sharing system where tasks have to be completed in a short period of time. SJF scheduling algorithm works best in a batch processing system where shorter jobs have to be completed first in order to increase throughput.Priority scheduling algorithm works better in a real-time system where certain tasks have to be prioritized so that they can be completed in a timely manner.

Factors Influencing CPU Scheduling Algorithms

There are many factors that influence the choice of CPU scheduling algorithm. Some of them are listed below.

- The number of processes.

- The processing time required.
- The urgency of tasks.
- The system requirements.

Selecting the correct algorithm will ensure that the system will use system resources efficiently, increase productivity, and improve user satisfaction.

CPU Scheduling Algorithms

There are several CPU Scheduling Algorithms, that are listed below.

- First Come First Served (FCFS)
- Shortest Job First (SJF)
- Longest Job First (LJF)
- Priority Scheduling
- Round Robin (RR)
- Shortest Remaining Time First (SRTF)
- Longest Remaining Time First (LRTF)

Thread Scheduling

Last Updated : 22 Sep, 2023

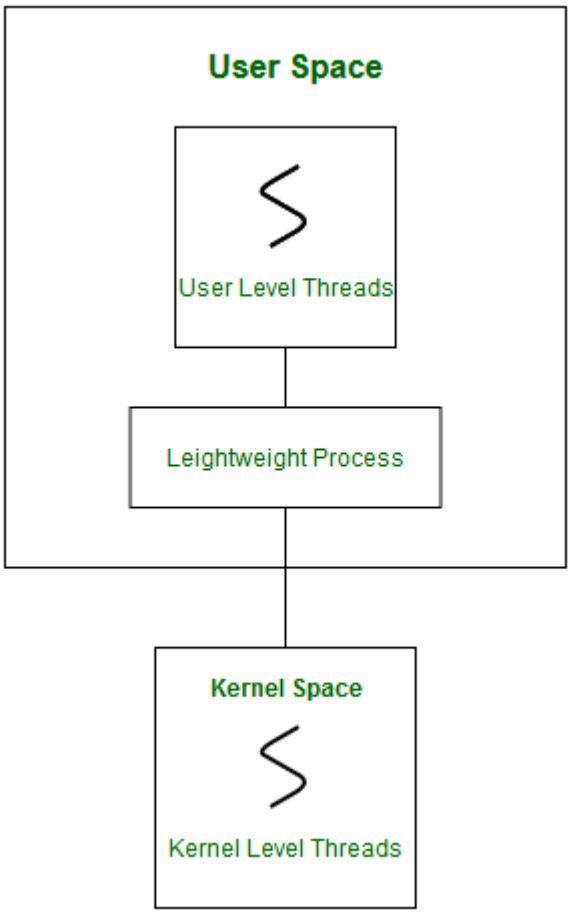
There is a component in Java that basically decides which thread should execute or get a resource in the operating system.

Scheduling of threads involves two boundary scheduling.

1. Scheduling of user-level threads (ULT) to kernel-level threads (KLT) via lightweight process (LWP) by the application developer.
2. Scheduling of kernel-level threads by the system scheduler to perform different unique OS functions.

Lightweight Process (LWP)

Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources. Thread library schedules which thread of a process to run on which LWP and how long. The number of LWPs created by the thread library depends on the type of application. In the case of an I/O bound application, the number of LWPs depends on the number of user-level threads. This is because when an LWP is blocked on an I/O operation, then to invoke the other ULT the thread library needs to create and schedule another LWP. Thus, in an I/O bound application, the number of LWP is equal to the number of the ULT. In the case of a CPU-bound application, it depends only on the application. Each LWP is attached to a separate kernel-level thread.



In real-time, the first boundary of thread scheduling is beyond specifying the scheduling policy and the priority. It requires two controls to be specified for the User level threads: Contention scope, and Allocation domain. These are explained as following below.

Contention Scope

The word contention here refers to the competition or fight among the User level threads to access the kernel resources. Thus, this control defines the extent to which contention takes place. It is defined by the application developer using the thread library.
Depending upon the extent of contention it is classified as-

- **Process Contention Scope (PCS) :**
  The contention takes place among threads **within a same process**. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs (priority as specified by the application developer during thread creation).
- **System Contention Scope (SCS) :**
  The contention takes place among **all threads in the system**. In this case, every SCS thread is associated to each LWP by the thread library and are scheduled by the system scheduler to access the kernel resources.
  In LINUX and UNIX operating systems, the POSIX Pthread library provides a function *Pthread_attr_setscope* to define the type of contention scope for a thread during its creation.

```
int Pthread_attr_setscope(pthread_attr_t *attr, int scope)
```

The first parameter denotes to which thread within the process the scope is defined.
The second parameter defines the scope of contention for the thread pointed. It takes two values.

```
PTHREAD_SCOPE_SYSTEM
```

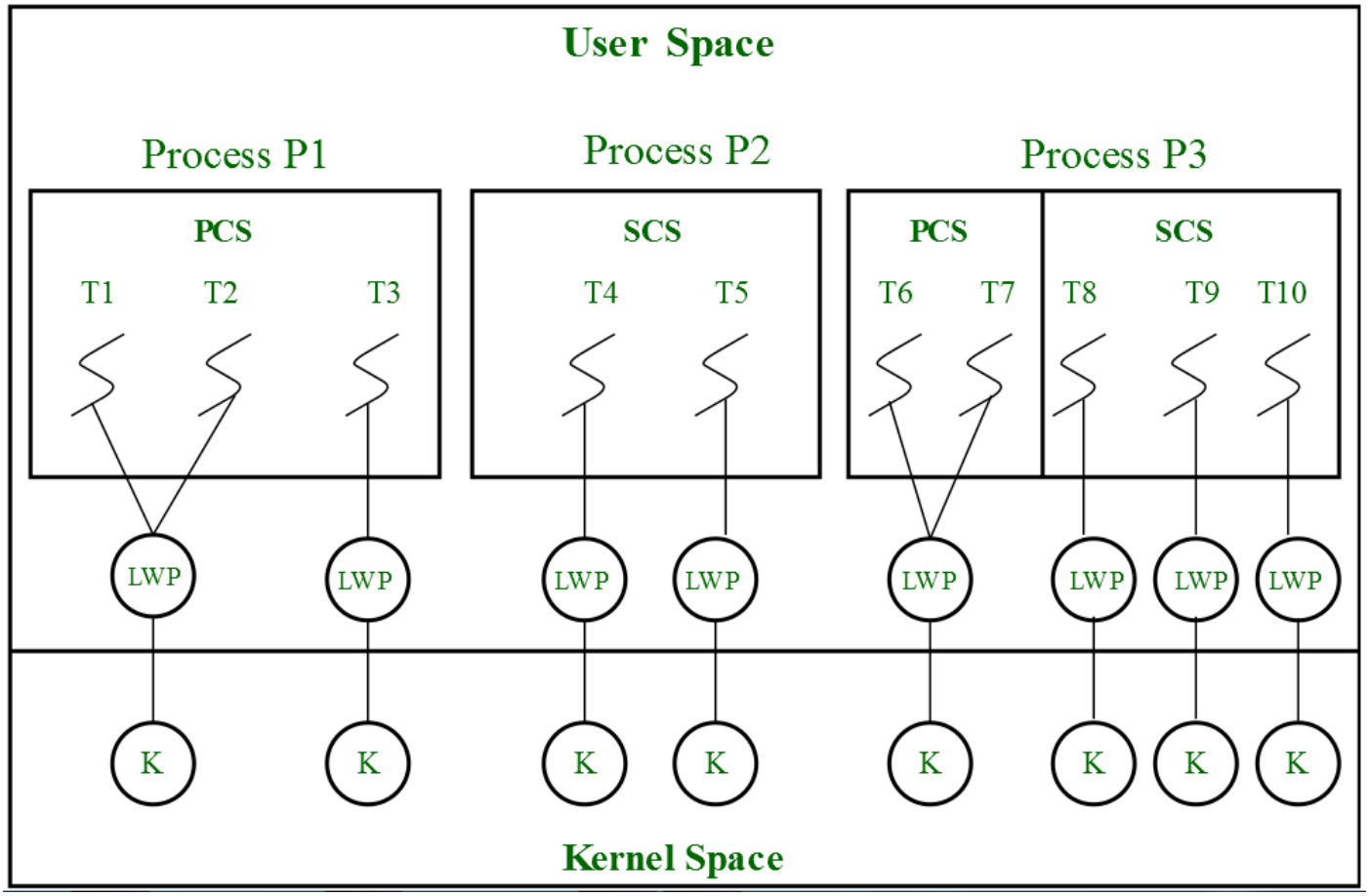
```
PTHREAD_SCOPE_PROCESS
```

If the scope value specified is not supported by the system, then the function returns *ENOTSUP*.

Allocation Domain

The allocation domain is **a set of one or more resources** for which a thread is competing. In a multicore system, there may be one or more allocation domains where each consists of one or more cores. One ULT can be a part of one or more allocation domain. Due to this high complexity in dealing with hardware and software architectural interfaces, this control is not specified. But by default, the multicore system will have an interface that affects the allocation domain of a thread.

Consider a scenario, an operating system with three process P1, P2, P3 and 10 user level threads (T1 to T10) with a single allocation domain. 100% of CPU resources will be distributed among all the three processes. The amount of CPU resources allocated to each process and to each thread depends on the contention scope, scheduling policy and priority of each thread defined by the application developer using thread library and also depends on the system scheduler. These User level threads are of a different contention scope.



In this case, the contention for allocation domain takes place as follows:

Process P1

All PCS threads T1, T2, T3 of Process P1 will compete among themselves. The PCS threads of the same process can share one or more LWP. T1 and T2 share an LWP and T3 are allocated to a separate LWP. Between T1 and T2 allocation of kernel resources via LWP is based on preemptive priority scheduling by the thread library. A Thread with a high priority will preempt low priority threads. Whereas, thread T1 of process p1 cannot preempt thread T3 of process p3 even if the priority of T1 is greater than the priority of T3. If the priority is

equal, then the allocation of ULT to available LWPs is based on the scheduling policy of threads by the system scheduler(not by thread library, in this case).

Process P2

Both SCS threads T4 and T5 of process P2 will compete with processes P1 as a whole and with SCS threads T8, T9, T10 of process P3. The system scheduler will schedule the kernel resources among P1, T4, T5, T8, T9, T10, and PCS threads (T6, T7) of process P3 considering each as a separate process. Here, the Thread library has no control of scheduling the ULT to the kernel resources.

Process P3

Combination of PCS and SCS threads. Consider if the system scheduler allocates 50% of CPU resources to process P3, then 25% of resources is for process scoped threads and the remaining 25% for system scoped threads. The PCS threads T6 and T7 will be allocated to access the 25% resources based on the priority by the thread library. The SCS threads T8, T9, T10 will divide the 25% resources among themselves and access the kernel resources via separate LWP and KLT. The SCS scheduling is by the system scheduler.

Note:

```
For every system call to access the kernel resources, a Kernel Level thread is
created
and associated to separate LWP by the system scheduler.
Number of Kernel Level Threads = Total Number of LWP
Total Number of LWP = Number of LWP for SCS + Number of LWP for PCS
Number of LWP for SCS = Number of SCS threads
Number of LWP for PCS = Depends on application developer
```

Here,

```
Number of SCS threads = 5
Number of LWP for PCS = 3
Number of SCS threads = 5
Number of LWP for SCS = 5
Total Number of LWP   = 8 (=5+3)
Number of Kernel Level Threads = 8
```

Advantages of PCS over SCS

- If all threads are PCS, then context switching, synchronization, scheduling everything takes place within the userspace. This reduces system calls and achieves better performance.
- PCS is cheaper than SCS.
- PCS threads share one or more available LWPs. For every SCS thread, a separate LWP is associated.For every system call, a separate KLT is created.
- The number of KLT and LWPs created highly depends on the number of SCS threads created. This increases the kernel complexity of handling scheduling and synchronization. Thereby, results in a limitation over SCS thread creation, stating that, the number of SCS threads to be smaller than the number of PCS threads.
- If the system has more than one allocation domain, then scheduling and synchronization of resources becomes more tedious. Issues arise when an SCS thread is a part of more than one allocation domain, the system has to handle n number of interfaces