Chapter 5 – CPU Scheduling

CPU scheduling is the basis of multi-programmed operating systems. By switching the CPU among processes, the OS can make the computer more productive. In this chapter we introduce basic CPU-scheduling concepts and present several CPU-scheduling algorithms.

Contents

5.1 Basic Concepts

5.1.1 CPU-I/O Burst Cycle
5.1.2 CPU Scheduler
5.1.3 Preemptive and Non-preemptive Scheduling
5.1.4 Dispatcher

5.2 Scheduling Criteria

5.3 Scheduling Algorithms

5.3.1 Frist-Come, Frist-Served Scheduling
5.3.2 Shortest Job Frist Scheduling
5.3.3 Round-Robin Scheduling
5.3.4 Priority Scheduling
5.3.5 Multilevel Queue Scheduling

5.4 Thread Scheduling

5.5 Multiprocessor Scheduling

5.6 Real-Time CPU Scheduling

Chapter Objectives

- *Describe various CPU scheduling algorithms*
- *Assess CPU scheduling algorithms based on scheduling criteria*
- *Explain the issues related to multiprocessor and multicore scheduling*
- *Describe various real-time scheduling algorithms*

5.1 Basic Concepts

In a single core system, only one program can run at a time, others keep waiting until the CPU becomes available. A process executes until it has to wait for I/O, or until its termination. While a process waits for I/O, the CPU just sits idle. The objective of multiprogramming is to keep some process running at all times so that the CPU usage can be maximized and the waiting time can be minimized. So, in those systems several processes are kept in memory at a time. When one process has to wait, the OS reassigns

the CPU core to another process, so the CPU does not have to sit idle. This pattern continues – every time one process has to wait for an event, another process gets the CPU.

The CPU is one of the primary computer resources. So its scheduling is central to an OS design.

5.1.1 CPU-I/O Burst Cycle

As we know, process execution consists of a cycle of CPU execution (**CPU burst**) and I/O wait (**I/O burst**). Processes alternate between these two states – a process execution starts with a CPU burst, followed by an I/O burst, which is followed by another CPU burst and so on. Eventually, the final CPU burst ends with a system request to terminate execution.

The durations of CPU bursts have been measured extensively. In general, **short** CPU bursts occur in **large** number and **long** CPU bursts occurs **less**. An I/O-bound program typically has many short CPU bursts, whereas a CPU bound program can have a few long CPU bursts. This distribution can be important in implementing a CPU-scheduling algorithm.

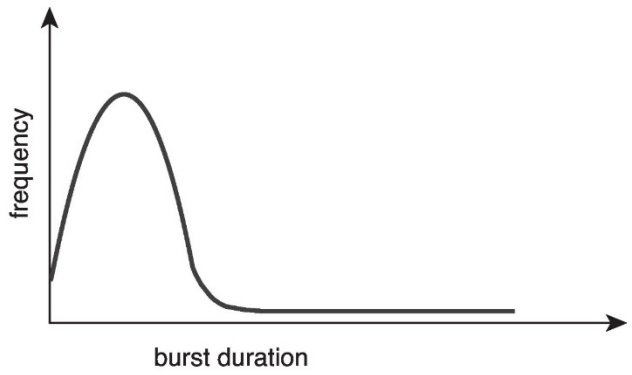


Figure 1 *Histogram of CPU-burst durations*

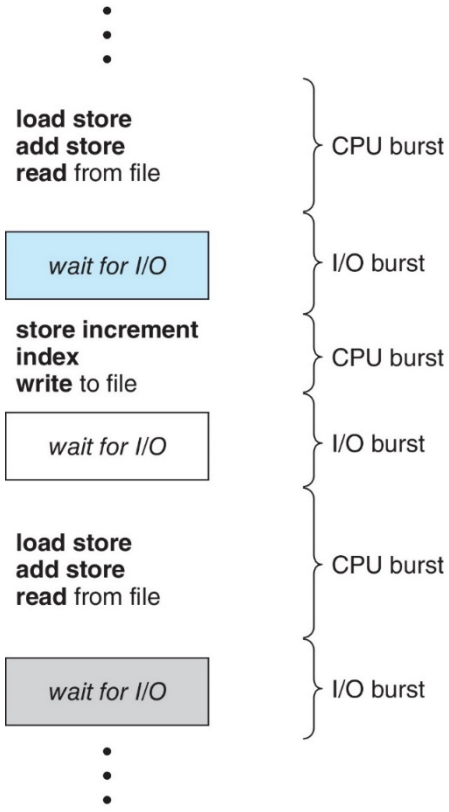


Figure 2 *Alternating sequence of CPU and I/O bursts.*

5.1.2 CPU Scheduler

Whenever the CPU becomes idle, the OS must select one of the processes from the **ready queue** to execute. This is done by the **CPU scheduler** which selects one from the processes that are loaded into memory and ready to execute. Then it allocates the CPU core to that process. According to the scheduling algorithm, the ready queue can be implemented in various ways such as a FIFO queue, a priority queue, a tree or even an unordered linked list.

5.1.3 Preemptive and Non-preemptive Scheduling

CPU scheduling decisions take place when a **process**:

1. Switches from **running** to **waiting** state (for example, for an I/O request or wait() system call)
2. Switches from **running** to **ready** state (for example when an interrupt occurs)
3. Switches from **waiting** to **ready** state (for example, at completion of I/O) or
4. When a process terminates and the CPU becomes available

In the cases **1** and **4**, CPU becomes idle and a new process must be selected for execution. When CPU scheduling is done only in these cases, it is called **non-preemptive** or **cooperative** scheduling, that is – once the CPU is allocated to a process, it keeps the CPU until it terminates or has to wait for an event.

If scheduling is also done in the situations **2** and **3**, that is called **preemptive** scheduling, that is – the scheduler can **preempt** a running process to give CPU access to another ready process.

All modern OS (including Windows, macOS, Linux and UNIX) use **preemptive** scheduling.

5.1.4 Dispatcher

The **Dispatcher** is the module that gives control of the CPU core to the process selected by the CPU scheduler. Dispatching involves:

- Switching context from one process to another
- Switching to user mode
- Point to the proper location in the user program to resume it.

The time it takes for the dispatcher to stop one process and start another running is called **dispatch latency**.

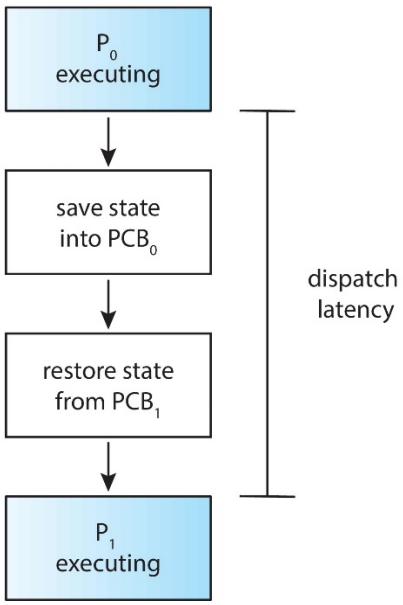


Figure 3 *The role of the dispatcher*

5.2 Scheduling Criteria

Different CPU scheduling algorithms have different properties, and the choice of a proper one is crucial for performance. Following are the criteria for judging a scheduling algorithm:

- **CPU utilization** – The percentage of time the CPU is busy executing processes. The goal is to keep the CPU busy as much as possible.
- **Throughput** – Number of processes that complete execution per time unit, it is the measure of work done.
- **Turnaround time** – Amount of time to execute a particular process; it is the interval from the time of submission of a process to the time of its completion, i.e., the sum of the periods spent waiting in the ready queue, executing on the CPU and doing I/O.

- **Waiting time** – Amount of time a process has to wait in the ready queue, which varies with different scheduling algorithms.
- **Response time** – Amount of time it takes from when a request was submitted until the first response (not output) is produced (in time-sharing environments).

The goal of the CPU scheduling algorithms is to:

- **Maximize** CPU utilization
- **Maximize** throughput
- **Minimize** turnaround time
- **Minimize** waiting time
- **Minimize** response time

5.3 Scheduling Algorithms

CPU scheduling algorithms decide which of the processes in the ready queue is to select to be executed in the CPU core. In this section we describe several of them. These algorithms are described in the context of only one processing core, which can be extended for multiprocessing systems.

5.3.1 First-Come, First-Served (FCFS) Scheduling

It is the simplest CPU scheduling algorithm, where the process that requests the CPU first is allocated to CPU first, which can be easily implemented by a FIFO queue.

Let's consider the following set of three processes that arrive at time 0, with the length of their CPU bursts given in milliseconds.

Process	Burst Time(ms)
P1	24
P2	3
P3	3

Suppose that the processes arrive in the order: **P1**, **P2**, and **P3**. The **Gantt chart** for the schedule is:

- **Waiting time for P1 = 0; P2 = 24; P3 = 27 (milliseconds)**
- **Average waiting time = (0 + 24 + 27)/3 = 17ms**

Suppose that the processes arrive in the order: **P2**, **P3**, and **P1**. In this case, the **Gantt chart** is:

- **Waiting time for P1 = 6; P2 = 0; P3 = 3 (milliseconds)**
- **Average waiting time = (6 + 0 + 3)/3 = 3ms**

We can see that the average waiting time under the FCFS policy is often quite long, especially when a short process is behind a long process.

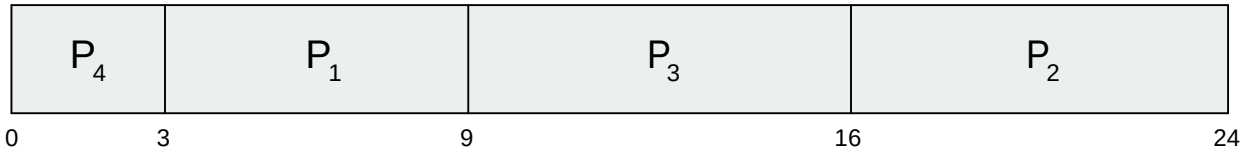
5.3.2 Shortest Job First (SJF) Scheduling

This algorithm associates with each process the length of its **next** CPU burst. When the CPU is available, it is assigned to the process that has the **smallest next CPU burst**. If the next CPU bursts of two processes are the same, then FCFS is used to break the tie.

Let's consider again a set of four processes that arrive at time 0, with the length of their CPU bursts given in milliseconds.

Process	Burst Time (ms)
P1	6
P2	8
P3	7
P4	3

Using **SJF** scheduling, these processes are scheduled according to the following **Gantt chart**:



- **Waiting time for P1 = 3; P2 = 16; P3 = 9 and P4 = 0 (milliseconds)**
- **Average waiting time = (3 + 16 + 9 + 0) / 4 = 7ms**

SJF is optimal as it gives minimum average waiting time for a given set of processes, but it cannot be implemented at the level of CPU scheduling as there is no way to know the length of the next CPU burst in advance. One way of solving this problem is by approximating this length. This can be done by using the length of previous CPU bursts, in an approach called exponential averaging.

5.3.3 Round-Robin (RR) Scheduling

This is similar to FCFS, but with **preemption**, so that the system can switch between processes when a process has a CPU burst that is too long. A small unit of time (is generally from 10 to 100 milliseconds) is defined as a **quantum** or **time slice**.

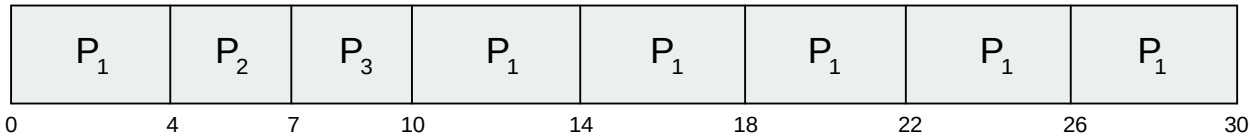
In Round-Robin scheduling, each process gets a small unit of CPU time (**time quantum, *q***) to execute. After this time elapses, the process is preempted and added to the end of the ready queue, which is implemented as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval.

If there are ***n*** processes in the ready queue and the time quantum is ***q***, then each process gets **1/*n*** of the CPU time in chunks of at most ***q*** time units at once. No process has to wait for more than **(*n*-1)*q*** time units. The timer interrupts every quantum to schedule the next process.

Let's consider again the set of three processes we saw earlier, all arrive at time 0.

Process	Burst Time(ms)
P1	24
P2	3
P3	3

The **Gantt chart** for **Round-Robin** scheduling with **time quantum = 4** is:



- **Waiting time for P1 = (10 - 4) = 6; P2 = 4 and P3 = 7 (milliseconds)**
- **Average waiting time = (6 + 4 + 7) / 3 = 5.66ms**

Typically **RR** has higher average turnaround than **SJF**, but with better response time.

The duration of **q** needs to be selected carefully, because when **q** is large, **RR** acts as the **FCFS** and if **q** is too small, then a lot context switching occurs and the overhead becomes high. In general, 80% of CPU bursts should be shorter than **q**.

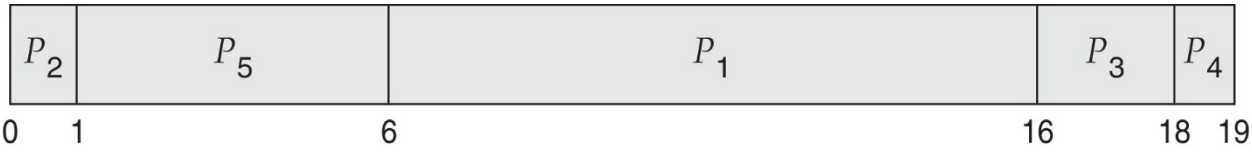
5.3.4 Priority Scheduling

In in approach, a **priority** (**integer**) is associated with each process. The CPU is allocated to the process with the **highest priority** (smallest integer has the highest priority). Processes with equal priority are scheduled in FCFS order.

Let's consider the set of following five processes that arrive at time 0, with the length of their CPU bursts given in milliseconds.
Each process has a priority associated, 1 means highest priority, 2 is next highest and so on.

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

The **Gantt chart** for the **Priority Scheduling** of these processes is:



- **Waiting time for P1 = 6; P2 = 0; P3 = 16, P4 = 18 and p5 = 1 (milliseconds)**
- **Average waiting time = (6 + 0 + 16 + 18 + 1) / 5 = 8.2ms**

Priority Scheduling can be **Preemptive** or **Non-preemptive**. One drawback of Priority Scheduling is **starvation** – that is, low priority processes may never get the chance to execute. This can be solved using the concept of a**ging** – that is, as time progresses, increasing the priority of the process.

5.3.5 Multilevel Queue Scheduling

This is Priority Scheduling, with **separate** queues for each priority. The processes in the higher priority queue are scheduled first.

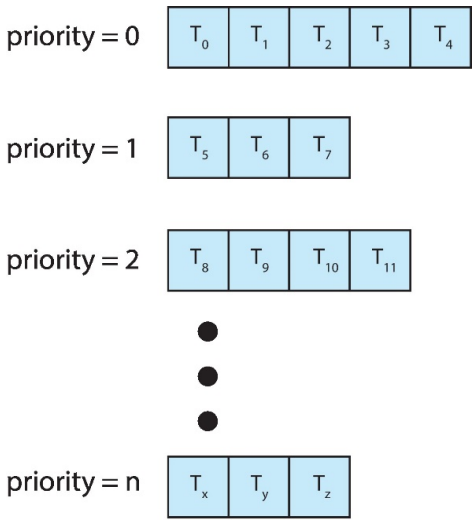


Figure 4 *Separate queue for each priority*

5.4 Thread Scheduling

In the systems where multithreading is implemented, scheduling is done on threads, not on processes.

5.5 Multiprocessor Scheduling

CPU scheduling is more complex when multiple CPUs are available. In this case,

- All threads/processes may be put in a common ready queue, or
- Each processor may have its own private queue of threads

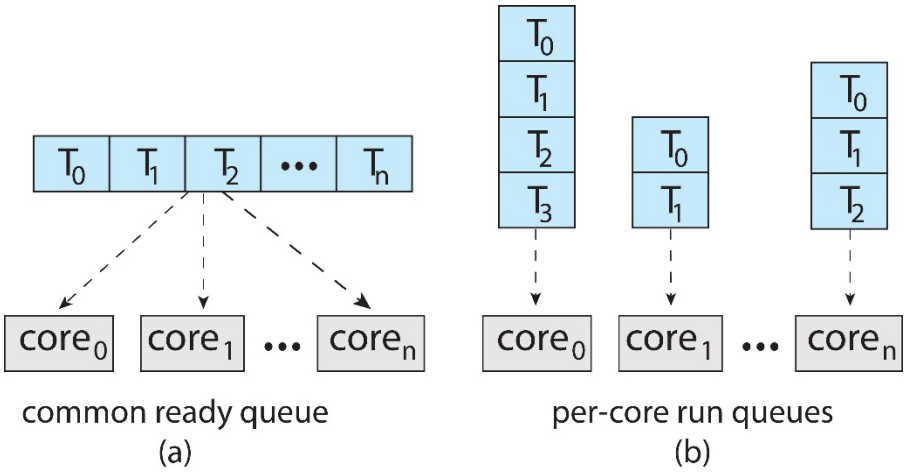


Figure 5 *Organization of ready queues*

5.6 Real-Time CPU Scheduling

In real-time CPU scheduling, two types of latencies affect performance, which should be considered:

- **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt.
- **Dispatch latency** – time for the scheduler to take current process off the CPU and switch to another.