Unit 4 CPU Scheduling Algorithms

Structure:

4.1 Introduction

Dear student, in the last unit you have studied about process management. In this unit, let's explore various CPU scheduling algorithms. The CPU scheduler selects a process from among the ready processes to execute on the CPU. CPU scheduling is the basis for multi-programmed operating systems. CPU utilization increases by switching the CPU among ready processes instead of waiting for each process to terminate before executing the next. In this unit we will discuss various CPU scheduling algorithms such as First-Come-First-Served, Shortest-Job-First, and Round-Robin etc.

Objectives:

After studying this unit, you should be able to:

- explain basic scheduling concepts
- describe different scheduling algorithms
- discuss the evaluation of scheduling algorithms

4.2 Basic Concepts of Scheduling

Let's start with basic concepts of scheduling. Scheduling is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

CPU- I/O Burst Cycle

Process execution consists of alternate CPU (Central Processing Unit) execution and I/O (Input / Output) wait. A cycle of these two events repeats

till the process completes execution (Figure 4.1). Process execution begins with a CPU burst followed by an I/O burst and then another CPU burst and so on. Eventually, a CPU burst will terminate the execution. An I/O bound job will have short CPU bursts and a CPU bound job will have long CPU bursts.

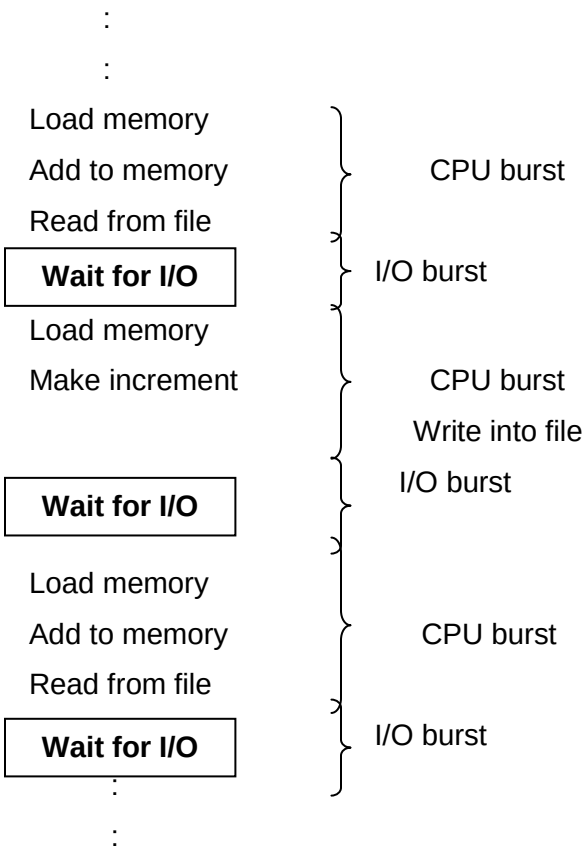


Fig. 4.1: CPU and I/O bursts

CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The short-term scheduler (or CPU scheduler) carries out the selection process. The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue may be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queue are generally PCBs of the processes.

Preemptive / Non-preemptive Scheduling

CPU scheduling decisions may take place under the following four circumstances. When a process:

1. switches from running state to waiting (an I/O request)
2. switches from running state to ready state (expiry of a time slice)
3. switches from waiting to ready state (completion of an I/O)
4. terminates

Scheduling under condition (1) or (4) is said to be non-preemptive. In non-preemptive scheduling, a process once allotted the CPU keeps executing until the CPU is released either by a switch to a waiting state or by termination. Preemptive scheduling occurs under condition (2) or (3). In preemptive scheduling, an executing process is stopped executing and returned to the ready queue to make the CPU available for another ready process. Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt. It is to be noted that pre-emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures.

Preemption also has an effect on the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an active on behalf *of* a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in t1: middle of these changes, and the kernel (or the device driver) needs to read (modify the same structure). Chaos ensues. Some operating systems, including most versions of UNIX, deal with this problem by waiting either for a system call to complete, or for an I/O block to take place, before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while

the kernel data structures are in an inconsistent state. Unfortunately, this kernel execution model is a poor one for supporting real-time computing and multiprocessing.

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. This function involves:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the *dispatch latency.*

Scheduling Criteria

Many algorithms exist for CPU scheduling. Various criteria have been suggested for comparing these CPU scheduling algorithms. Common criteria include:

1) **CPU Utilization:** We want to keep the CPU as busy as possible. CPU utilization may range from 0% to 100% ideally. In real systems it ranges, from 40% for a lightly loaded system to 90% for heavily loaded systems.
2) **Throughput:** Number of processes completed per time unit is throughput. For long processes may be of the order of one process per hour whereas in case of short processes, throughput may be 10 or 12 processes per second.
3) **Turnaround Time:** The interval of time between submission and completion of a process is called turnaround time. It includes execution time and waiting time.
4) **Waiting Time:** Sum of all the times spent by a process at different instances waiting in the ready queue is called waiting time.
5) **Response Time:** In an interactive process the user is using some output generated while the process continues to generate new results. Instead of using the turnaround time that gives the difference between time of submission and time of completion, response time is sometimes used.

Response time is thus the difference between time of submission and the time the first response occurs.

Desirable features include maximum CPU utilization, throughput and minimum turnaround time, waiting time and response time.

Self Assessment Questions

1. Scheduling is a fundamental operating-system function. (True / False)
2. The _____ selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
3. In real systems CPU utilization ranges from _____ for a lightly loaded system to _____ for heavily loaded systems.
   a) 40%, 90%
   b) 50%, 50%
   c) 90%, 40%
   d) 25%, 75%

4.3 Scheduling Algorithms

Scheduling algorithms differ in the manner in which the CPU selects a process in the ready queue for execution. In this section, we shall describe several of these algorithms.

First-Come-First-Served Scheduling Algorithm

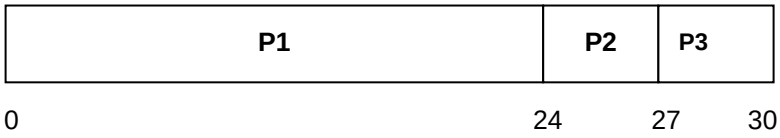
This is one of the brute force algorithms. A process that requests for the CPU first is allocated the CPU first. Hence, the name first come first serve. The FCFS algorithm is implemented by using a first-in-first-out (FIFO) queue structure for the ready queue. This queue has a head and a tail. When a process joins the ready queue its PCB is linked to the tail of the FIFO queue. When the CPU is idle, the process at the head of the FIFO queue is allocated the CPU and deleted from the queue.

Even though the algorithm is simple, the average waiting is often quite long and varies substantially if the CPU burst times vary greatly, as seen in the following example.

Consider a set of three processes P1, P2 and P3 arriving at time instant 0 and having CPU burst times as shown below:

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

The Gantt chart below shows the result.



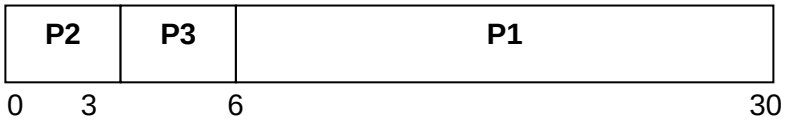
Average waiting time and average turnaround time are calculated as follows:

The waiting time for process P1 = 0 ms
P2 = 24 ms
P3 = 27 ms

Average waiting time = (0 + 24 + 27) / 3 = 51 / 3 = 17 ms

P1 completes at the end of 24 ms, P2 at the end of 27 ms and P3 at the end of 30 ms. Average turnaround time = (24 + 27 + 30) / 3 = 81 / 3 = 27 ms.

If the processes arrive in the order P2, P3 and P3, then the result will be as follows:



Average waiting time = (0 + 3 + 6) / 3 = 9 / 3 = 3 ms.

Average turnaround time = (3 + 6 + 30) / 3 = 39 / 3 = 13 ms.

Thus, if processes with smaller CPU burst times arrive earlier, then average waiting and average turnaround times are lesser. The algorithm also suffers from what is known as a convoy effect. Consider the following scenario. Let there be a mix of one CPU bound process and many I/O bound processes in the ready queue. The CPU bound process gets the CPU and executes (long I/O burst). In the meanwhile, I/O bound processes finish I/O and wait for CPU, thus leaving the I/O devices idle. The CPU bound process releases the CPU as it goes for an I/O. I/O bound processes have short CPU bursts

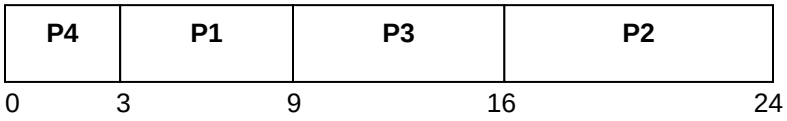
and they execute and go for I/O quickly. The CPU is idle till the CPU bound process finishes the I/O and gets hold of the CPU. The above cycle repeats. This is called the convoy effect. Here small processes wait for one big process to release the CPU. Since the algorithm is non-preemptive in nature, it is not suited for time sharing systems.

Shortest-Job-First Scheduling

Another approach to CPU scheduling is the shortest job first algorithm. In this algorithm, the length of the CPU burst is considered. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. Hence this is named shortest job first. In case there is a tie, FCFS scheduling is used to break the tie. As an example, consider the following set of processes P1, P2, P3, P4 and their CPU burst times:

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

Using SJF algorithm, the processes would be scheduled as shown below.



Average waiting time = (0 + 3 + 9 + 16) / 4 = 28 / 4 = 7 ms.

Average turnaround time = (3 + 9 + 16 + 24) / 4 = 52 / 4 = 13 ms.

If the above processes were scheduled using FCFS algorithm, then

Average waiting time = (0 + 6 + 14 + 21) / 4 = 41 / 4 = 10.25 ms.

Average turnaround time = (6 + 14 + 21 + 24) / 4 = 65 / 4 = 16.25 ms.

The SJF algorithm produces the most optimal scheduling scheme. For a given set of processes, the algorithm gives the minimum average waiting and turnaround times. This is because, shorter processes are scheduled earlier than longer ones and hence waiting time for shorter processes decreases more than it increases the waiting time of long processes.

The main disadvantage with the SJF algorithm lies in knowing the length of the next CPU burst. In case of long-term or job scheduling in a batch system,

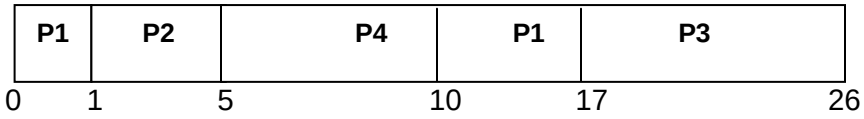
the time required to complete a job as given by the user can be used to schedule. SJF algorithm is therefore applicable in long-term scheduling. The algorithm cannot be implemented for CPU scheduling as there is no way to accurately know in advance the length of the next CPU burst. Only an approximation of the length can be used to implement the algorithm.

But the SJF scheduling algorithm is provably optimal and thus serves as a benchmark to compare other CPU scheduling algorithms. SJF algorithm could be either preemptive or non-preemptive. If a new process joins the ready queue with a shorter next CPU burst then what is remaining of the current executing process, then the CPU is allocated to the new process. In case of non-preemptive scheduling, the current executing process is not preempted and the new process gets the next chance, it being the process with the shortest next CPU burst.

Given below are the arrival and burst times of four processes P1, P2, P3 and P4.

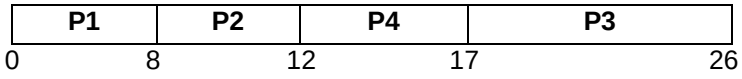
Process	Arrival time (ms)	Burst time (ms)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

If SJF preemptive scheduling is used, the following Gantt chart shows the result.



Average waiting time = ((10 – 1) + 0 + (17 – 2) + (15 – 3)) / 4 = 26 / 4 = 6.5 ms.

If non-preemptive SJF scheduling is used, the result is as follows:



Average waiting time = ((0 + (8 – 1) + (12 – 3) + (17 – 2)) / 4 = 31 / 4 = 7.75 ms.

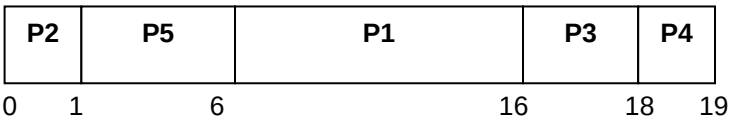
Priority Scheduling

In priority scheduling each process can be associated with a priority. CPU is allocated to the process having the highest priority. Hence this is named priority scheduling. Equal priority processes are scheduled according to FCFS algorithm.

The SJF algorithm is a particular case of the general priority algorithm. In this case priority is the inverse of the next CPU burst time. Larger the next CPU burst, lower is the priority and vice versa. In the following example, we will assume lower numbers to represent higher priority.

Process	Priority	Burst time (ms)
P1	3	10
P2	1	1
P3	3	2
P4	4	1
P5	2	5

Using priority scheduling, the processes are scheduled as shown in the Gantt chart:



Average waiting time = (6 + 0 + 16 + 18 + 1) / 5 = 41 / 5 = 8.2 ms.

Priorities can be defined either internally or externally. Internal definition of priority is based on some measurable factors like memory requirements, number of open files and so on. External priorities are defined by criteria such as importance of the user depending on the user's department and other influencing factors.

Priority based algorithms can be either preemptive or non-preemptive. In case of preemptive scheduling, if a new process joins the ready queue with a priority higher than the process that is executing, then the current process is preempted and CPU allocated to the new process. But in case of non-preemptive algorithm, the new process having highest priority from among the ready processes is allocated the CPU only after the current process gives up the CPU.

Starvation or indefinite blocking is one of the major disadvantages of priority scheduling. Every process is associated with a priority. In a heavily loaded system, low priority processes in the ready queue are starved or never get a chance to execute. This is because there is always a higher priority process ahead of them in the ready queue.

A solution to starvation is aging. Aging is a concept where the priority of a process waiting in the ready queue is increased gradually. Eventually the lowest priority process ages to attain the highest priority, by which it gets a chance to execute on the CPU.

Round-Robin Scheduling

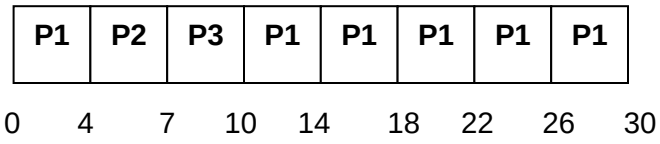
The round-robin CPU scheduling algorithm is basically a preemptive scheduling algorithm designed for time-sharing systems. One unit of time is called a time slice (Quantum). Duration of a time slice may range between 10 ms and about 100 ms. The CPU scheduler allocates to each process in the ready queue one time slice at a time in a round-robin fashion. Hence this is named round-robin.

The ready queue in this case is a FIFO queue with new processes joining the tail of the queue. The CPU scheduler picks processes from the head of the queue for allocating the CPU. The first process at the head of the queue gets to execute on the CPU at the start of the current time slice and is deleted from the ready queue. The process allocated the CPU may have the current CPU burst either equal to the time slice or smaller than the time slice or greater than the time slice. In the first two cases, the current process will release the CPU on its own and thereby the next process in the ready queue will be allocated the CPU for the next time slice. In the third case, the current process is preempted, stops executing, goes back and joins the ready queue at the tail thereby making way for the next process.

Consider the same example explained under FCFS algorithm.

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

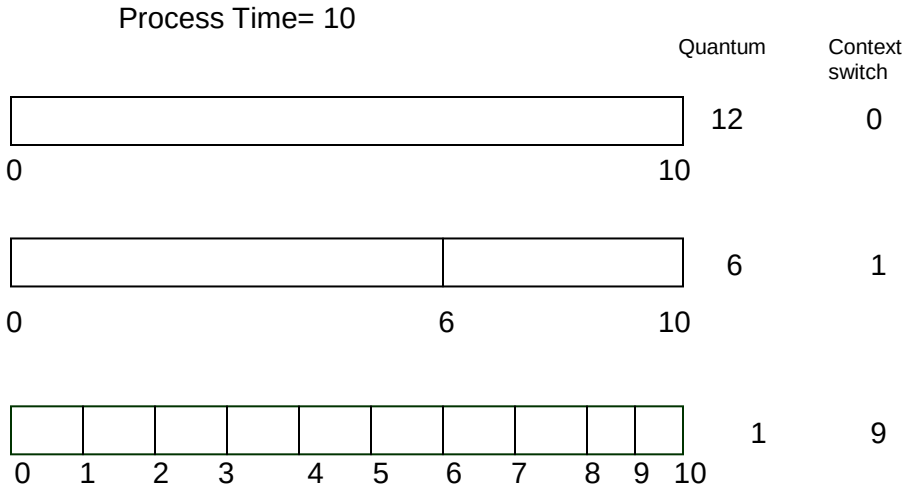
Let the duration of a time slice be 4 ms, which is to say CPU switches between processes every 4 ms in a round-robin fashion. The Gantt chart below shows the scheduling of processes.



Average waiting time = (4 + 7 + (10 – 4)) / 3 = 17/ 3 = 5.66 ms.

If there are 5 processes in the ready queue that is n = 5, and one time slice is defined to be 20 ms that is q = 20, then each process will get 20 ms or one time slice every 100 ms. Each process will never wait for more than (n – 1) x q time units.

The performance of the RR algorithm is very much dependent on the length of the time slice. If the duration of the time slice is indefinitely large then the RR algorithm is the same as FCFS algorithm. If the time slice is too small, then the performance of the algorithm deteriorates because of the effect of frequent context switching. A comparison of time slices of varying duration and the context switches they generate on only one process of 10 time units is shown below.



The above example shows that the time slice should be large with respect to the context switch time, else, if RR scheduling is used the CPU will spend more time in context switching.

Multi-level Queue Scheduling

Processes can be classified into groups. For example, interactive processes, system processes, batch processes, student processes and so on. Processes belonging to a group have a specified priority. This algorithm partitions the ready queue into as many separate queues as there are groups. Hence this is named multilevel queue scheduling. Based on certain properties, process is assigned to one of the ready queues. Each queue can have its own scheduling algorithm like FCFS or RR. For example, queue for interactive processes could be scheduled using RR algorithm where queue for batch processes may use FCFS algorithm. An illustration of multilevel queues is shown below (Figure 4.2).

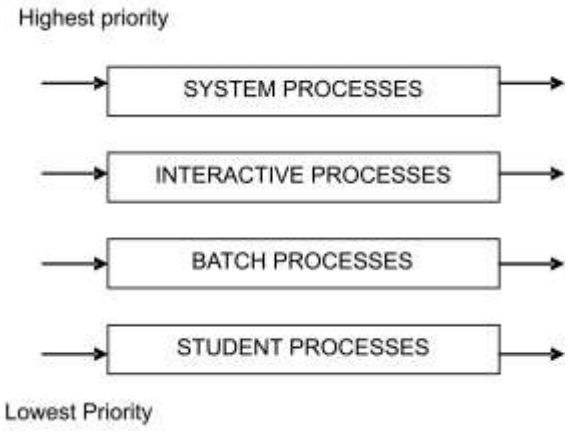


Fig. 4.2: Multilevel Queue scheduling.

Queues themselves have priorities. Each queue has absolute priority over low priority queues that are a process in a queue with lower priority will not be executed until all processes in a queue with higher priority have finished executing. If a process in a lower priority queue is executing (higher priority queues are empty) and a process joins a higher priority queue, then the executing process is preempted to make way for a process in the higher priority queue.

This priority on the queues themselves may lead to starvation. To overcome this problem, time slices may be assigned to queues when each queue gets

some amount of CPU time. The duration of the time slices may be different for queues depending on the priority of the queues.

Multi-level Feedback Queue Scheduling

In the previous multilevel queue scheduling algorithm, processes once assigned to queues do not move or change queues. Multilevel feedback queues allow a process to move between queues. The idea is to separate out processes with different CPU burst lengths. All processes could initially join the highest priority queue. Processes requiring longer CPU bursts are pushed to lower priority queues. I/O bound and interactive processes remain in higher priority queues. Aging could be considered to move processes from lower priority queues to higher priority to avoid starvation. An illustration of multilevel feedback queues is shown in Figure 4.3.

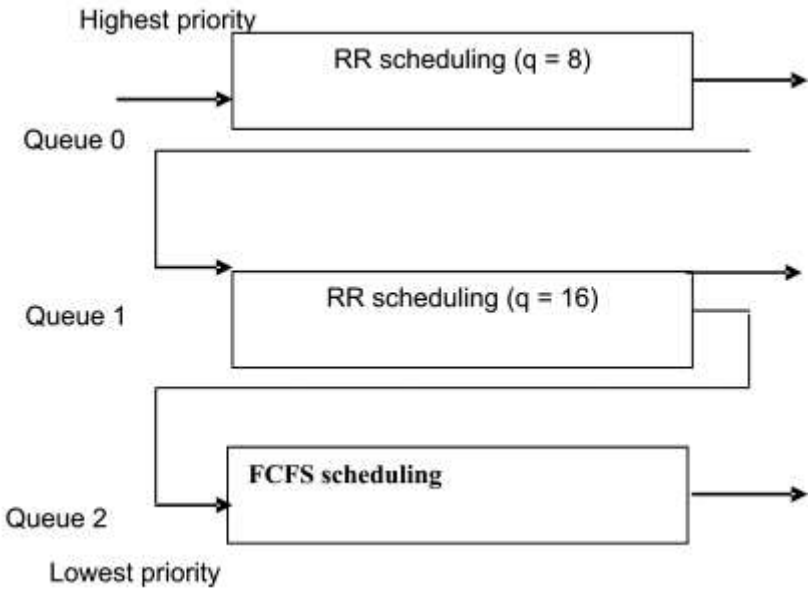


Fig. 4.3: Multilevel Feedback Queue Scheduling

As shown in Figure 4.3, a process entering the ready queue joins queue 0. RR scheduling algorithm with q = 8 is used to schedule processes in queue 0. If the CPU burst of a process exceeds 8 ms, then the process preempted, deleted from queue 0 and joins the tail of queue 1. When queue 0 becomes empty, then processes in queue 1 will be scheduled. Here also RR scheduling algorithm is used to schedule processes but q = 16. This will

give processes a longer time with the CPU. If a process has a CPU burst still longer, then it joins queue 3 on being preempted. Hence highest priority processes (processes having small CPU bursts, that is I/O bound processes) remain in queue 1 and lowest priority processes (those having long CPU bursts) will eventually sink down. The lowest priority queue could be scheduled using FCFS algorithm to allow processes to complete execution.

Multilevel feedback scheduler will have to consider parameters such as number of queues, scheduling algorithm for each queue, criteria for upgrading a process to a higher priority queue, criteria for downgrading a process to a lower priority queue and also the queue to which a process initially enters.

Multiple-processor Scheduling

When multiple processors are available, then the scheduling gets more complicated, because now there is more than one CPU which must be kept busy and in effective use at all times. Multi-processor systems may be **heterogeneous,** (different kinds of CPUs), or **homogenous,** (all the same kind of CPU). This book will restrict its discussion to homogenous systems. In homogenous systems any available processor can then be used to run any processes in the queue. Even within homogenous multiprocessor, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes wishing to use that device must be scheduled to run on that processor; otherwise the device would not be available.

If several identical processors are available, then load sharing can occur. It would be possible to provide a separate queue for each processor. In this case, however one processor could be idle, with an empty queue, while another processor was very busy. To prevent this situation, we use a common ready queue. All processes go into one queue and are scheduled onto any available processor.

In such scheme, one of two scheduling approaches may be used. In one approach, each processor is self–scheduling. Each processor examines the common ready queue and selects a process to execute. We must ensure that two processors do not choose the same process, and that processes are not lost from the queue. The other approach avoids this problem by

appointing one processor as scheduler for other processors, thus creating a master-slave structure.

Some systems carry this structure one step further, by having all scheduling decisions, I/O processing, and other system activities handled by one single processor- the master server. The other processors only execute user code. This asymmetric multiprocessing is far simpler than symmetric multiprocessing, because only one processor accesses the system data structures, alleviating the need for data sharing.

Real-time Scheduling

Real-time computing is divided into two types: hard real-time systems and soft real-time systems. Hard real time systems are required to complete a critical task within a guaranteed amount of time. A process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O. The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as resource reservation. Such a guarantee requires that the scheduler knows exactly how long each type of operating system function takes to perform, and therefore each operation must be guaranteed to take a maximum amount of time. Such a guarantee is impossible in a system with secondary storage or virtual memory, because these subsystems cause unavoidable and unforeseeable variation in the amount of time to execute a particular process. Therefore, hard real-time systems are composed of special purpose software running on hardware dedicated to their critical process, and lack the functionality of modern computers and operating systems.

Soft real-time computing is less restrictive. It requires that critical processes receive priority over less fortunate ones. Implementing soft real-time functionality requires careful design of the scheduler and related aspects of the operating system. First, the system must have priority scheduling, and real-time processes must have the highest priority. The priority of real-time processes must not degrade over time, even though the priority of non-real time processes may. Secondly the dispatch latency must be small. The smaller the latency, the faster a real-time process can start executing once it is runnable. To keep dispatch latency low, we need to allow system calls to be pre-emptible.

Figure 4.4 shows a make-up of dispatch latency. The conflict phase of dispatch latency has three components:

1) preemption of any process running in the kernel
2) low-priority processes releasing resources needed by the high-priority process
3) context switching from the current process to the high-priority process

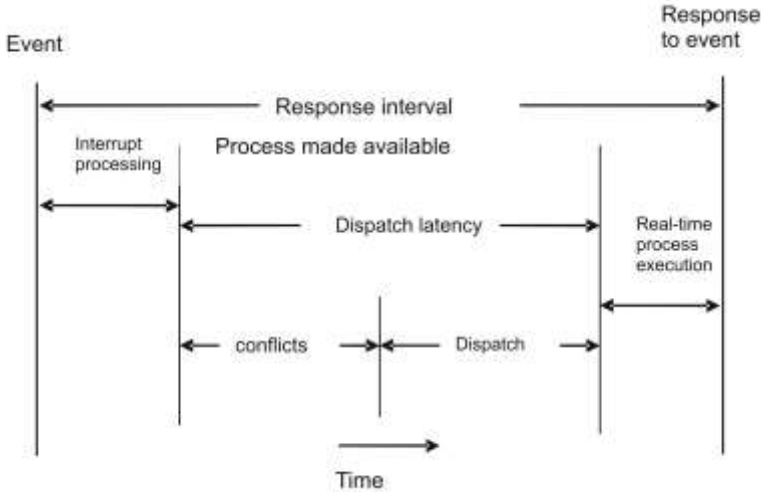


Fig. 4.4: Dispatch latency

As an example in Solaris 2, the dispatch latency with preemption disabled is over 100 milliseconds. However, the dispatch latency with preemption enabled is usually to 2 milliseconds.

Self Assessment Questions

4. The FCFS algorithm is implemented by using a Last-In-First-Out (LIFO) queue structure. (True / False)
5. In _____ algorithm, when the CPU is available, it is assigned to the process that has the smallest next CPU burst.
6. The _____ CPU scheduling algorithm is basically a preemptive scheduling algorithm designed for time-sharing systems.

4.4 Evaluation of CPU Scheduling Algorithms

We have many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult. To select an algorithm first we must define, the criteria on which we can select the best algorithm. These criteria may include several measures, such as:

- Maximize CPU utilization under the constraint that the maximum response time is 1 second.
- Maximize throughput such that turnaround time is (on average) linearly proportional to total execution time.

Once the selection criteria have been defined, we use one of the following different evaluation methods.

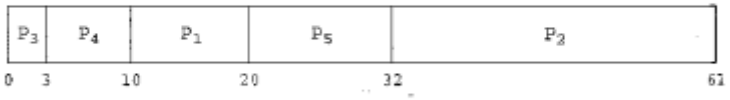
Deterministic Modeling

If a specific workload is known, then the exact values for major criteria can be fairly easily calculated, and the "best" determined. For example, consider the following workload (with all processes arriving at time 0), and the resulting schedules determined by three different algorithms:

Process	Burst Time
P1	10
P2	29
P3	3
P4	7
P5	12

FCFS:

Non-preemptive SJF:



Round Robin:

The average waiting times for FCFS, SJF, and RR are 28ms, 13ms, and 23ms respectively. Deterministic modeling is fast and easy, but it requires specific known input, and the results only apply for that particular set of input. However, by examining multiple similar cases, certain trends can be observed. (Like the fact that for processes arriving at the same time, SJF will always yield the shortest average wait time.)

Queuing Models

Specific process data is often not available, particularly for future times. However, a study of historical performance can often produce statistical descriptions of certain important parameters, such as the rate at which new processes arrive, the ratio of CPU bursts to I/O times, the distribution of CPU burst times and I/O burst times, etc.

Armed with those probability distributions and some mathematical formulas, it is possible to calculate certain performance characteristics of individual waiting queues. For example, **Little's Formula** says that for an average queue length of N, with an average waiting time in the queue of W, and an average arrival of new jobs in the queue of Lambda, the these three terms can be related by:

N = Lambda x W

Queuing models treat the computer as a network of interconnected queues, each of which is described by its probability distribution statistics and formulas such as Little's formula. Unfortunately real systems and modern scheduling algorithms are so complex as to make the mathematics intractable in many cases with real systems.

Simulations

Another approach is to run computer simulations of the different proposed algorithms (and adjustment parameters) under different load conditions, and to analyze the results to determine the "best" choice of operation for a particular load pattern. Operating conditions for simulations are often randomly generated using distribution functions similar to those described above. A better alternative when possible is to generate ***trace tapes***, by monitoring and logging the performance of a real system under typical expected work loads. These are better because they provide a more accurate picture of system loads, and also because they allow multiple simulations to be run with the identical process load, and not just statistically equivalent loads. A compromise is to randomly determine system loads and then save the results into a file, so that all simulations can be run against identical randomly determined system loads.

Although trace tapes provide more accurate input information, they can be difficult and expensive to collect and store, and their use increases the

complexity of the simulations significantly. There are also some questions as to whether the future performance of the new system will really match the past performance of the old system. (If the system runs faster, users may take fewer coffee breaks, and submit more processes per hour than under the old system. Conversely if the turnaround time for jobs is longer, intelligent users may think more carefully about the jobs they submit rather than randomly submitting jobs and hoping that one of them works out.) Figure 4.5 shows the evaluation of CPU schedulers by simulation.

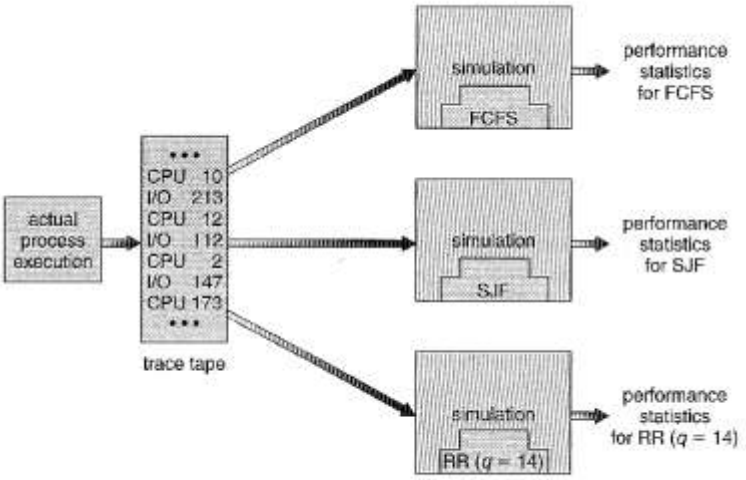


Fig. 4.5: Evaluation of CPU schedulers by simulation

Implementation

The only real way to determine how a proposed scheduling algorithm is going to operate is to implement it on a real system. For experimental algorithms and those under development, this can cause difficulties and resistances among users who don't care about developing OS's and are only trying to get their daily work done. Even in this case, the measured results may not be definitive, for at least two major reasons: (1) System workloads are not static, but change over time as new programs are installed, new users are added to the system, new hardware becomes available, new work projects get started, and even societal changes. (For example the explosion of the Internet has drastically changed the amount of network traffic that a system sees and the importance of handling it with rapid response times.) (2) As mentioned above, changing the scheduling

system may have an impact on the workload and the ways in which users use the system.

Most modern systems provide some capability for the system administrator to adjust scheduling parameters, either on the fly or as the result of a reboot or a kernel rebuild.

Self Assessment Questions

7. To select an algorithm first we must define, the criteria on which we can select the best algorithm. (True / False)
8. N = Lambda x W is _____ formula.

4.5 Summary

Let's summarize the key points covered in this unit:

- The long-term scheduler provides a proper mix of CPU-I/O bound jobs for execution. The short-term scheduler has to schedule these processes for execution.
- Scheduling can either be preemptive or non-preemptive. If preemptive, then an executing process can be stopped and returned to ready state to make the CPU available for another ready process. But if non-preemptive scheduling is used then a process once allotted the CPU keeps executing until either the process goes into wait state because of an I/O or it has completed execution.
- First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes.
- Shortest-Job-First (SJF) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult because predicting the length of the next CPU burst is difficult.
- Multilevel queue algorithms allow different algorithms to be used for various classes of processes.

4.6 Terminal Questions

1. List out various CPU scheduling criteria.
2. Discuss First-Come-First-Served scheduling algorithm.
3. What are the drawbacks of Shortest-Job-First scheduling algorithm?
4. How will you evaluate CPU schedulers by simulation?

4.7 Answers

Self Assessment Questions

1. True
2. CPU Scheduler
3. a) 40%, 90%
4. False
5. Shortest-Job-First Scheduling
6. Round-Robin
7. True
8. Little's

Terminal Questions

1. Many algorithms exist for CPU scheduling. Various criteria have been suggested for comparing these CPU scheduling algorithms. Common criteria include: CPU utilization, Throughput, Turnaround time, Waiting time, Response time etc. (Refer Section 4.2)
2. First Come First Served scheduling algorithm is one of the very brute force algorithms. A process that requests for the CPU first is allocated the CPU first. Hence, the name first come first serve. The FCFS algorithm is implemented by using a first-in-first-out (FIFO) queue structure for the ready queue. (Refer Section 4.3)
3. The main disadvantage with the SJF algorithm lies in knowing the length of the next CPU burst. In case of long-term or job scheduling in a batch system, the time required to complete a job as given by the user can be used to schedule. SJF algorithm is therefore applicable in long-term scheduling. (Refer Section 4.3)
4. Computer simulations of the different proposed algorithms (and adjustment parameters) are run under different load conditions, and the results are analyzed to determine the "best" choice of operation for a particular load pattern. Operating conditions for simulations are often randomly generated using distribution functions. (Refer Section 4.4)