CPU SCHEDULING

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

6.1 Basic Concepts

The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. In a uniprocessor system, only one process may run at a time; any other processes must wait until the CPU is free and can be rescheduled.

The idea of multiprogramming is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU would then sit idle; all this waiting time is wasted. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues.

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.

6.1.1 CPU-I/O Burst Cycle

The success of CPU scheduling depends on the following observed property of processes: Process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst.** That is followed by an ***I/O* burst,** then another CPU burst, then another I/O burst, and so on. Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst

(Figure 6.1). The durations of these CPU bursts have been extensively measured. Although they vary greatly by process and by computer. This distribution can help us select an appropriate CPU-scheduling algorithm.

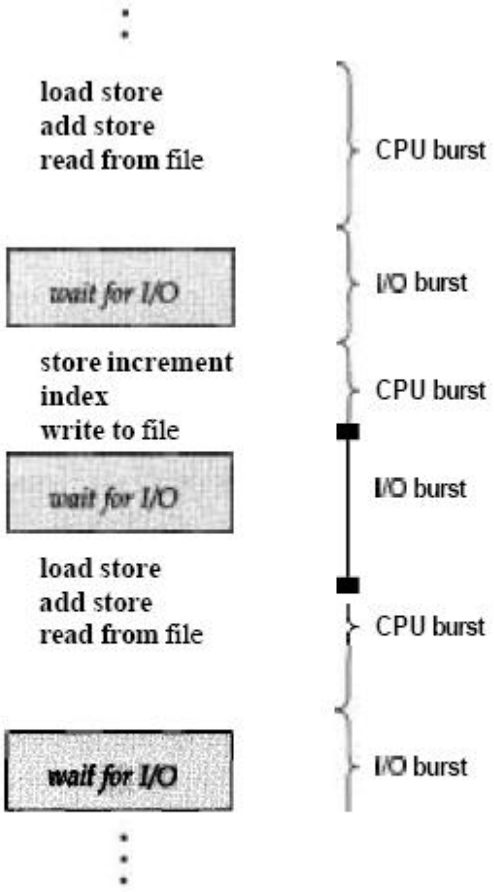

Figure 6.1 Alternating sequence of CPU and I/O bursts.

6.1.2 CPU Scheduler

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

The ready queue is not necessarily a first-in, first-out (FIFO) queue. 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 queues are generally process control blocks (PCBs) of the processes.

6.1.3 Preemptive Scheduling

CPU scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, I/O request, or invocation of wait for the termination of one of the child processes)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, completion of I/O)
4. When a process terminates

In circumstances 1 and 4, there is no choice in terms of scheduling. **A** new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, in circumstances 2 and **3.**

When scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is **nonpreemptive;** otherwise, the scheduling scheme is **preemptive.** Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems. It is the only method that can be used on certain hardware platforms,because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

Preemptive scheduling incurs a cost. Consider the case of two processes sharing data. One may be in the midst of updating the data when it is preempted and the

second process is run. The second process may try to read the data, which are currently in an inconsistent state. New mechanisms thus are needed to coordinate access to shared data.

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 activity 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 the middle of these changes, and the kernel (or the device driver) needs to read or modify the same structure? Chaos could ensue. 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.

6.1.4 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**.

6.2 . Scheduling Criteria

Different CPU-scheduling algorithms have different properties and may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. The characteristics used for comparison can make a substantial difference in the determination of the best algorithm. The criteria include the following:

CPU utilization: We want to keep the CPU as busy as possible. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

Throughput: If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes completed per time unit, called throughput. For long processes, this rate may be 1 process per hour; for short transactions, throughput might be 10 processes per second.

a Turnaround time: From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

a Response time: In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early, and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the amount

of time it takes to start responding, but not the time that it takes to output that response. The turnaround time is generally limited by the speed of the output device.

We want to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, in some circumstances we want to optimize the minimum or maximum values, rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

For interactive systems (such as time-sharing systems), some analysts suggest that minimizing the variance in the response time is more important than minimizing the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average, but is highly variable. However, little work has been done on CPU-scheduling algorithms to minimize variance.