CPU Scheduling

SUBJECT: OPERATING SYSTEMS

SAVIYA VARGHESE BCA 2020-21

Operating System Concepts

CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling
- Algorithm Evaluation

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.

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle Process execution consists of a *cycle* of CPU execution and I/O wait.
- CPU burst distribution

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

Alternating Sequence of CPU And I/O Bursts



Histogram of CPU-burst Times



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. 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 queues are generally process control blocks (PCBs) of the processes.

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
 - 1.Switches from running to waiting state.
 - 2.Switches from running to ready state.
 - 3.Switches from waiting to ready.
 - 4.Terminates.

- Preemptive: allows a process to be interrupted in the midst of its CPU execution, taking the CPU away to another process
- Non- Preemptive: ensures that a process relinquishes control of CPU when it finishes with its current CPU burst
- Scheduling under 1 and 4 is *non preemptive*.
- All other scheduling is *preemptive*

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - o switching context
 - o switching to user mode
 - jumping to the proper location in the user program to restart that program
- *Dispatch latency* time it takes for the dispatcher to stop one process and start another running.

Scheduling Criteria

- CPU utilization keep the CPU as busy as possible
- Throughput # of processes that complete their execution per time unit
- Turnaround time amount of time to execute a particular process (finishing time arrival time)
- Waiting time amount of time a process has been waiting in the ready queue
- Response time amount of time it takes from when a request was submitted until the first response is produced, **not** output (for timesharing environment)

Scheduling Criteria

• **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.

• **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.

• **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.

Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come**, **first-served** (FCFS) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.



24

27

30

- Waiting time for $P_1 = 0; P_2 = 24; P_3 = 27$
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2, P_3, P_1$$
.

• The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case.
- Convoy effect short process behind long process

FCFS Scheduling (Cont.)

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get the CPU and hold it. During this time, all the other processes will finish their I/O and move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have very short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a convoy effect, as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

FCFS Scheduling (Cont.)

The FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is particularly troublesome for time-sharing systems, where each user needs to get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

Shortest-Job-First (SJF) Scheduling

• Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

Two schemes:

- nonpreemptive once CPU given to the process it cannot be preempted until completes its CPU burst.
- preemptive if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal gives minimum average waiting time for a given set of processes.

SJF

A different approach to CPU scheduling is the **shortest-job-first (SJF) scheduling algorithm**. This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie. Note that a more appropriate term would be the *shortest next CPU burst*, because the scheduling is done by examining the length of the next CPU burst of a process, rather than its total length. As an example, consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time		
P_1	6		
P_2	8		
P_3	7		
P_4	3		



The waiting time is 3 milliseconds for process P_1 , 16 milliseconds for process P_2 , 9 milliseconds for process P_3 , and 0 milliseconds for process P_4 . Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. If we were using the FCFS scheduling scheme, then the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably *optimal*, in that it gives the minimum average waiting time for a given set of processes. By moving a short process before a long one, the waiting time of the short process decreases more than it increases the waiting time of the long process. Consequently, the *average* waiting time decreases.

The real difficulty with the SJF algorithm is knowing the length of the next CPU request. For long-term (or job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job. Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response. (Too low a value will cause a time-limitexceeded error and require resubmission.) SJF scheduling is used frequently in long-term scheduling.

moccondo

The SJF algorithm may be either *preemptive* or *nonpreemptive*. The choice arises when a new process arrives at the ready queue while a previous process is executing. The new process may have a shorter next CPU burst than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

EXAMPLE

Process	Arrival Time	Burst Time		
P_1	0	8		
P_2	1	4		
P_3	2	9		
P_4	3	5		

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 is scheduled. The average waiting time for this example is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5 milliseconds. A nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

Example of Non-Preemptive SJF Process Arrival Time Burst Time P_1 0.0 P_{2} 2.0 P_3 4.0 P_4 5.0 4 SJF (non-preemptive)



Operating System Concepts

Average waiting time = (0 + 6 + 3 + 7)/4 - 4







Operating System Concepts

Average waiting time = (9 + 1 + 0 + 2)/4 = 3

Priority Scheduling

The SJF algorithm is a special case of the general **priority-scheduling algorithm**. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order P_1 , P_2 , ..., P_5 , with the length of the CPU-burst time given in milliseconds:



Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria that are external to the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority-scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority-scheduling algorithm will algorithm will simply put the new process at the head of the ready queue.

A major problem with priority-scheduling algorithms is **indefinite blocking** (or **starvation**). A process that is ready to run but lacking the CPU can be considered blocked—waiting for the CPU. A priority-scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU. In a heavily loaded computer system, a steady stream of higher-priority processes

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority).
 - Preemptive
 - o nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem = Starvation low priority processes may never execute.
- Solution = Aging as time progresses increase the priority of the process.

Round Robin (RR)

The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a **time quantum** (or **time slice**), is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

---- - more quanting and approxice the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the **tail** of the ready queue. The CPU scheduler will then select the next process in the ready queue. The average waiting time under the RR policy, however, is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time		
P_1	24		
P_2	3		
P_3	3		

If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2 . Since process P_2 does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P_3 . Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is



The average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a process' CPU burst exceeds 1 time quantum, that process is *preempted* and is put back in the ready queue. The RR scheduling algorithm is preemptive.

The performance of the RR algorithm depends neavily on the size of the time quantum. At one extreme, if the time quantum is very large (infinite), the RR policy is the same as the FCFS policy. If the time quantum is very small (say 1 microsecond), the RR approach is called processor sharing, and appears (in theory) to the users as though each of n processes has its own processor running at 1/n the speed of the real processor. This approach was used in Control Data Corporation (CDC) hardware to implement 10 peripheral processors with only one set of hardware and 10 sets of registers. The hardware executes one instruction for one set of registers, then goes on to the next. This cycle continues, resulting in 10 slow processors rather than one fast processor. (Actually, since the processor was much faster than memory and each instruction referenced memory, the processors were not much slower than 10 real processors would have been.)

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- 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 waits more than (*n*-1)*q* time units.

Performance

 $\circ q \text{ large} \Rightarrow \text{FIFO}$

• $q \text{ small} \Rightarrow q \text{ must}$ be large with respect to context switch, otherwise overhead is too high.



- Ready queue is partitioned into separate queues: foreground (interactive) background (batch)
- Each queue has its own scheduling algorithm, foreground – RR background – FCFS

• Scheduling must be done between the queues.

- Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
- Time slice each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
- o 20% to background in FCFS

A multilevel queue-scheduling algorithm partitions the ready queue into several separate queues (Figure 6.6). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

ouchoround processes.



Operating System Concepts

Let us look at an example of a multilevel queue-scheduling algorithm with five queues:

- 1. System processes
- 2. Interactive processes
- 3. Interactive editing processes
- 4. Batch processes
- 5. Student processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted. Solaris 2 uses a form of this algorithm.

Another possibility is to time slice between the queues. Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes in a FCFS manner.

Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - o number of queues
 - o scheduling algorithms for each queue
 - o method used to determine when to upgrade a process
 - o method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service

Normally, in a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.

Multilevel feedback queue scheduling, however, allows a process to move between queues. The idea is to separate processes with different CPU-burst characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. Similarly, a process that waits too long in a lowerpriority queue may be moved to a higher-priority queue. This form of aging prevents starvation.



Operating System Concepts

Example of Multilevel Feedback Queue

• Three queues:

- $\circ Q_{o}$ time quantum 8 milliseconds
- $\circ Q_1$ time quantum 16 milliseconds
- $\circ Q_2 FCFS$

Scheduling

- A new job enters queue Q_o which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1 .
- At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 .

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
- *Homogeneous processors* within a multiprocessor.
- Load sharing
- Asymmetric multiprocessing only one processor accesses the system data structures, alleviating the need for data sharing.

Real-Time Scheduling

- *Hard real-time* systems required to complete a critical task within a guaranteed amount of time.
- *Soft real-time* computing requires that critical processes receive priority over less fortunate ones.



Operating System Concepts

Algorithm Evaluation

- Deterministic modeling takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Queueing models
- Implementation



For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 6.7). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process that arrives for queue 0 will, in turn, preempt a process in queue 1.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis, only when queues 0 and 1 are empty.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8, but less than 24, milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higherpriority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it also requires some means of selecting values for all the parameters to define the best scheduler. Although a multilevel feedback queue is the most general scheme, it is also the most complex. **Process Scheduling Models**

Linux Process Scheduling

2 separate process-scheduling algorithms
time-sharing: a prioritized credit-based
Soft-real time: FCFS and RR

only allows processes in a user mode to be preempted.

Solaris 2 Scheduling



Windows 2000 Priorities

	real- time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1