Operating Systems 2023 Spring Term
Week 10

Dr. Emrah İnan (emrahinan@iyte.edu.tr)

CPU Scheduling

May 11, 2023

Week 8: Sample Glossary

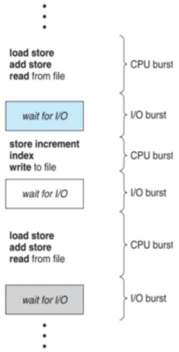
- **preemptive:** A form of scheduling in which processes or threads are involuntarily moved from the running state (e.g., by a timer signaling the kernel to allow the next thread to run). (on Page 1263)
- **nonpreemptive:** Scheduling in which, once a core has been allocated to a thread, the thread keeps the core until it releases the core either by terminating or by switching to the waiting state. (on Page 1260)

CPU Scheduling Objectives

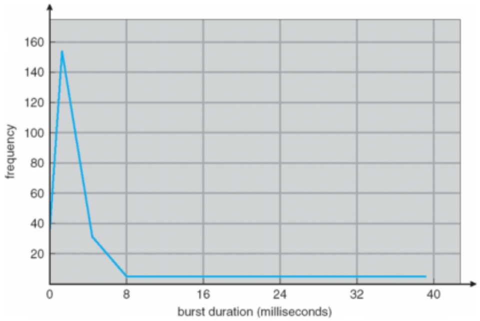
- CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU's core.
- The execution time is a resource that provided by a processor
  The resource is allocated by means of a schedule
- The aim of processor scheduling is to assign processes to be executed by the processor over time, in a way that meets system objectives, such as response time, throughput, and processor efficiency.
- The scheduling function should
  Share time fairly among processes
  Prevent starvation of a process
  Use the processor efficiently
  Have low overhead
  Prioritise processes when necessary (e.g. real time deadlines)

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 followed by I/O burst
- CPU burst distribution is of main concern



Histogram of CPU-burst Times



An I/O-bound program typically has many short CPU bursts.
A CPU-bound program might have a few long CPU bursts.
This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

CPU Scheduler

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them
  Queue may be ordered in various ways
- 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
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  Consider access to shared data
  Consider preemption while in kernel mode
  Consider interrupts occurring during crucial OS activities

Preemptive and Nonpreemptive Scheduling

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

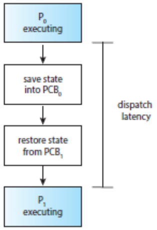
1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When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

For situations 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, for situations 2 and 3. Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes. Consider the case of two processes that share data.

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  switching context
  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

The role of the dispatcher



The dispatcher should be as fast as possible, since it is invoked during every context switch
How often do context switches occur? "vmstat" command that is available on Linux systems.

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system)
- Throughput – of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process (how long it takes to execute that process). The interval from the time of submission of a process to the time of completion
- 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 time-sharing environment)

Scheduling Algorithm Optimization Criteria

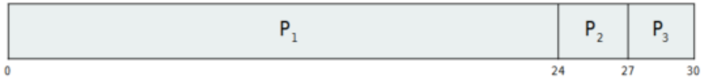
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

In most cases, we optimize the average measure. However, under some circumstances, we prefer 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.
An accurate illustration should involve many processes, each a sequence of several hundred CPU bursts and I/O bursts. For simplicity, though, we consider only one CPU burst (in milliseconds) per process in our examples. Our measure of comparison is the average waiting time.

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1 , P_2 , P_3
The Gantt Chart for the schedule is:



Waiting time for P_1 = 0; P_2 = 24; P_3 = 27

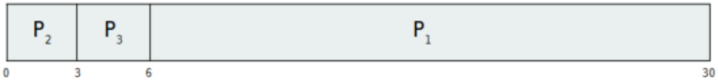
Average waiting time: (0 + 24 + 27)/3 = 17

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

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

- Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

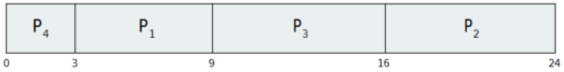
- Associate with each process the length of its next CPU burst
  Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  The difficulty is knowing the length of the next CPU request
  Could ask the user

Note that 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 thus particularly troublesome for interactive systems, where it is important that each process 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.

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

SJF scheduling chart



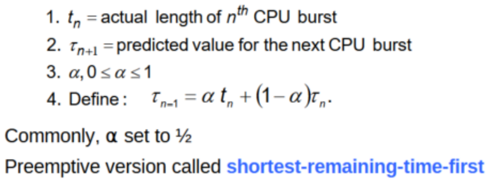
Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

By comparison, if we were using the FCFS scheduling scheme, 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.

Determining Length of Next CPU Burst

Although the SJF algorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling

- Can only estimate the length – should be similar to the previous one
  Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

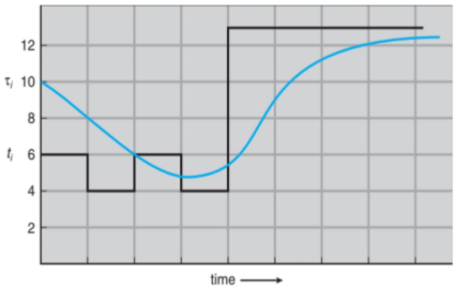


1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define: $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n .$

Commonly, α set to ½

Preemptive version called **shortest-remaining-time-first**

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

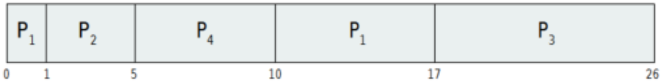
We may not know the length of the next CPU burst, but we may be able to predict its value.We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

Example of Shortest-remaining-time-first

Now we add the concepts of varying arrival times and preemption to the analysis

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

Preemptive SJF Gantt Chart



Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Priority Scheduling

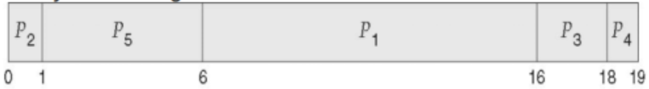
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer -> highest priority)
  Preemptive
  Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem -> Starvation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

Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could periodically (say, every second) increase the priority of a waiting process by 1.

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Priority scheduling Gantt Chart



Average waiting time = 8.2 msec

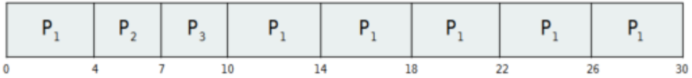
Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum q), 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.
- Timer interrupts every quantum to schedule next process
- Performance
  q large -> FIFO
  q small -> q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is:

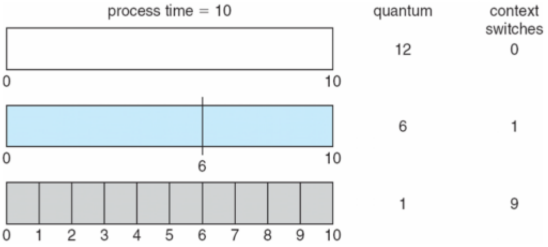


Typically, higher average turnaround than SJF, but better ***response***

q should be large compared to context switch time

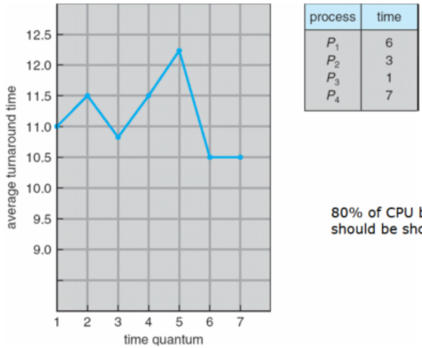
q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

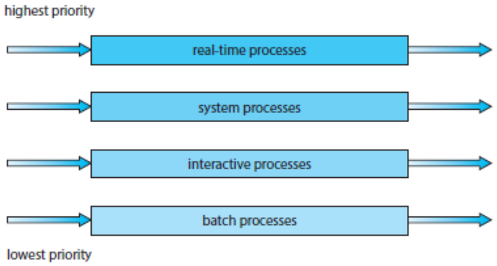
80% of CPU bursts
should be shorter than q

Turnaround time also depends on the size of the time quantum. As we can see from Figure, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  foreground (interactive)
  background (batch)
- Process permanently in a given queue
- 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
  20% to background in FCFS

Multilevel Queue Scheduling



Separate queues might be used for foreground and background processes, and each queue might have its own scheduling algorithm. The foreground queue might be scheduled by an RR algorithm, for example, while the background queue is scheduled by an FCFS algorithm.

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:
  number of queues
  scheduling algorithms for each queue
  method used to determine when to upgrade a process
  method used to determine when to demote a process
  method used to determine which queue a process will enter when that process needs service

The multilevel feedback queue scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. 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—which are typically characterized by short CPU bursts —in the higher-priority queues.

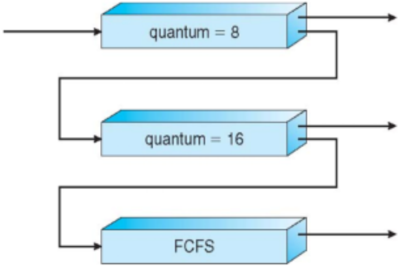
Example of Multilevel Feedback Queue

Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

Scheduling

- A new job enters queue Q_0 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


quantum = 8
quantum = 16
FCFS