

Scheduling

Introduction

In multiprogramming systems, when there is more than one runnable process (i.e., ready), the operating system must decide which one to activate. The decision is made by the part of the operating system called the *scheduler*, using a *scheduling algorithm*.

- **In the beginning**—there was no need for scheduling, since the users of computers lined up in front of the computer room or gave their job to an operator.
- **Batch processing**—the jobs were executed in first come first served manner.
- **Multiprogramming**—life became complicated!

The scheduler is concerned with deciding *policy*, not providing a *mechanism*.

The basics

Scheduling refers to a set of policies and mechanisms to control the order of work to be performed by a computer system. Of all the resources in a computer system that are scheduled before use, the CPU is by far the most important.

Multiprogramming is the (efficient) scheduling of the CPU. The basic idea is to keep the CPU busy as much as possible by executing a (user) process until it must wait for an event, and then switch to another process.

Processes alternate between consuming CPU cycles (*CPU-burst*) and performing I/O (*I/O-burst*).

Types of scheduling

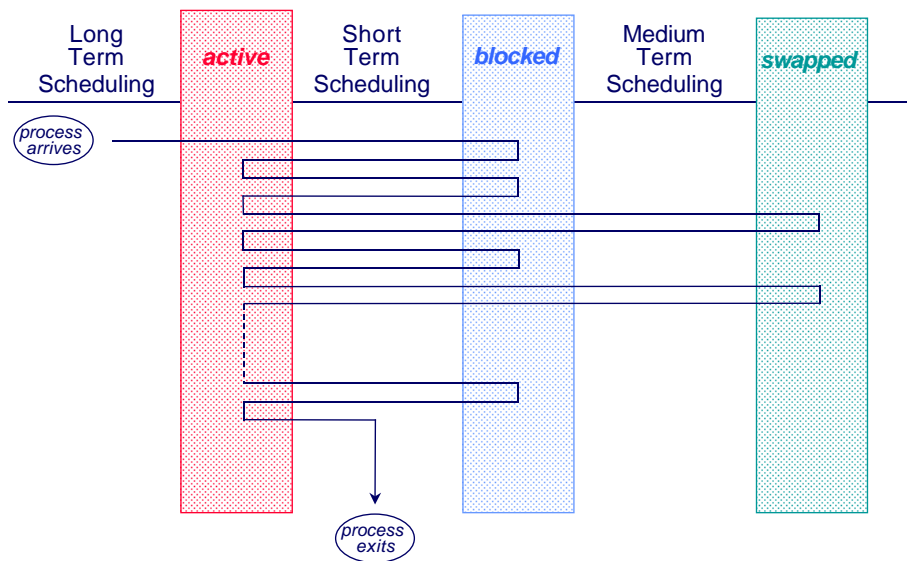
In general, (job) scheduling is performed in three stages: short-, medium-, and long-term. The activity frequency of these stages are implied by their names.

Long-term (job) scheduling is done when a new process is created. It initiates processes and so controls the *degree of multi-programming* (number of processes in memory).

Medium-term scheduling involves suspending or resuming processes by swapping (rolling) them out of or into memory.

Short-term (process or CPU) scheduling occurs most frequently and decides which process to execute next.

Life cycle of a typical process



Long- & medium-term scheduling

Acting as the primary resource allocator, the long-term scheduler admits more jobs when the resource utilization is low, and blocks the incoming jobs from entering the ready queue when utilization is too high.

When the main memory becomes over-committed, the medium-term scheduler releases the memory of a suspended (blocked or stopped) process by swapping (rolling) it out.

In summary, both schedulers control the level of multiprogramming and avoid (as much as possible) overloading the system by many processes and cause “thrashing” (more on this later).

Short-term scheduling

Short-term scheduler, also known as the process or CPU scheduler, controls the CPU sharing among the “ready” processes. The selection of a process to execute next is done by the short-term scheduler. Usually, a new process is selected under the following circumstances:

- When a process must wait for an event.
- When an event occurs (e.g., I/O completed, quantum expired).
- When a process terminates.

Short-term scheduling criteria

The goal of short-term scheduling is to optimize the system performance, and yet provide responsive service. In order to achieve this goal, the following set of criteria is used:

- CPU utilization
- I/O device throughput
- Total service time
- Responsiveness
- Fairness
- Deadlines

Scheduler design

A typical scheduler is designed to select one or more primary performance criteria and rank them in order of importance. One problem in selecting a set of performance criteria is that they often conflict with each other. For example, increased processor utilization is usually achieved by increasing the number of active processes, but then response time deteriorates. So, the design of a scheduler usually involves a careful balance of all requirements and constraints.

The following is only a small subset of possible characteristics:

I/O throughput, CPU utilization, response time (batch or interactive), urgency of fast response, priority, maximum time allowed, total time required.

Jan/01

Scheduling policies

In general, scheduling policies may be *preemptive* or *non-preemptive*.

In a non-preemptive pure multiprogramming system, the short-term scheduler lets the current process run until it blocks, waiting for an event or a resource, or it terminates.

Preemptive policies, on the other hand, force the currently active process to release the CPU on certain events, such as a clock interrupt, some I/O interrupts, or a system call.

Jan/01

Scheduling algorithms

The following are some common scheduling algorithms:

Non-preemptive

- First-Come-First-Served (FCFS)
- Shortest Job first (SJF)

Good for “background” batch jobs.

Preemptive

- Round-Robin (RR)
- Priority

Good for “foreground” interactive jobs.

Jan/01

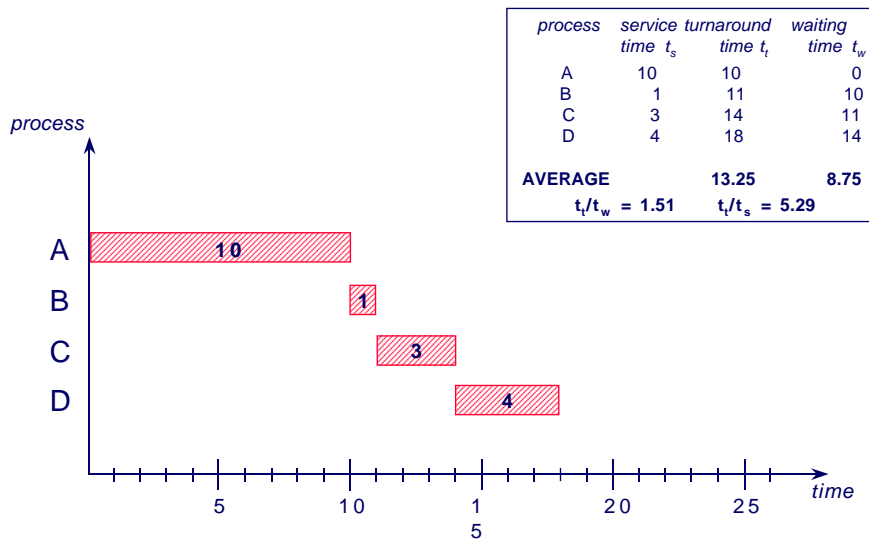
First-Come-First-Served (FCFS)

FCFS, also known as First-In-First-Out (FIFO), is the simplest scheduling policy. Arriving jobs are inserted into the tail (rear) of the ready queue and the process to be executed next is removed from the head (front) of the queue.

FCFS performs better for long jobs. Relative importance of jobs measured only by arrival time (poor choice). A long CPU-bound job may hog the CPU and may force shorter (or I/O-bound) jobs to wait prolonged periods. This in turn may lead to a lengthy queue of ready jobs, and thence to the “convoy effect.”

Jan/01

An example—FCFS



Jan/01

Copyright © 1998-2001 by Eskicioglu & Marsland

Scheduling 12

Shortest Job First (SJF)

SJF policy selects the job with the shortest (expected) processing time first. Shorter jobs are always executed before long jobs.

One major difficulty with SJF is the need to know or estimate the processing time of each job (can only predict the future!)

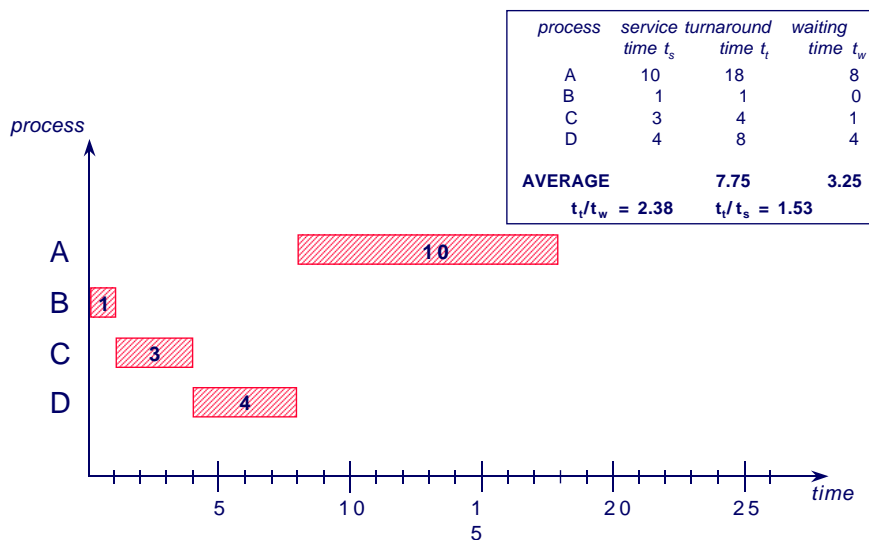
Also, long running jobs may starve, because the CPU has a steady supply of short jobs.

Jan/01

Copyright © 1998-2001 by Eskicioglu & Marsland

Scheduling 13

An example—SJF



Jan/01

Copyright © 1998-2001 by Eskicioglu & Marsland

Scheduling 14

Round-Robin (RR)

RR reduces the penalty that short jobs suffer with FCFS by preempting running jobs periodically. The CPU suspends the current job when the reserved *quantum* (*time-slice*) is exhausted. The job is then put at the end of the ready queue if not yet completed.

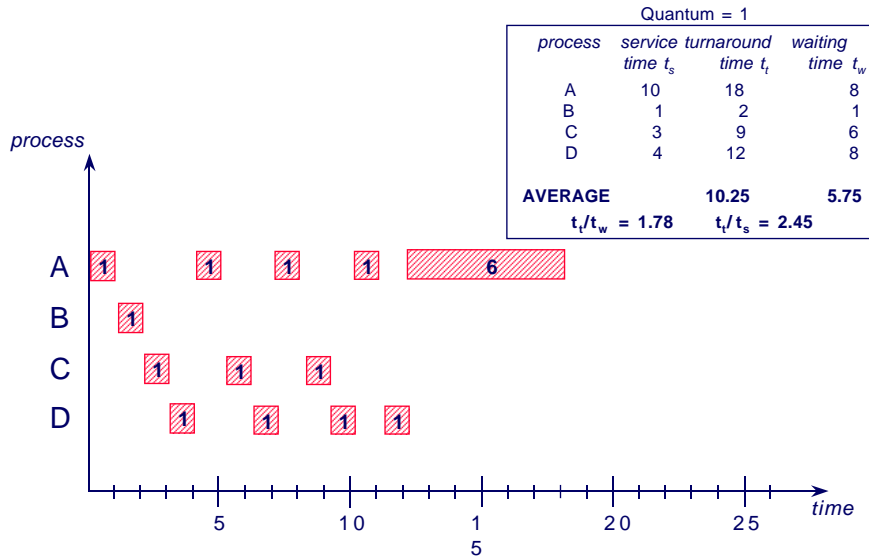
The critical issue with the RR policy is the length of the quantum. If it is too short, then the CPU will be spending more time on context switching. Otherwise, interactive processes will suffer.

Jan/01

Copyright © 1998-2001 by Eskicioglu & Marsland

Scheduling 15

An example—RR



Jan/01

Copyright © 1998-2001 by Eskicioglu & Marsland

Scheduling 16

Priority-based

Each process is assigned a priority. The ready list contains an entry for each process ordered by its priority. The process at the beginning of the list (highest priority) is picked first.

A variation of this scheme allows preemption of the current process when a higher priority process arrives.

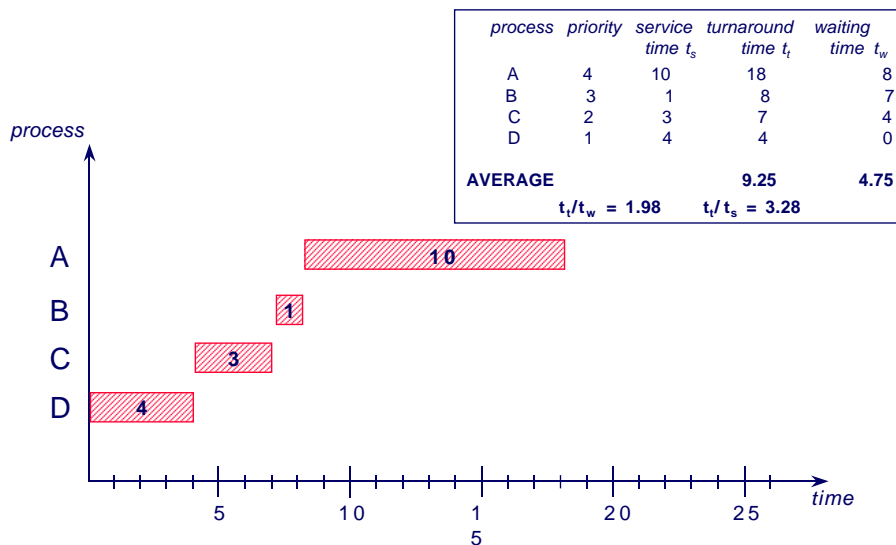
Another variation of the policy adds an aging scheme, where the priority of a process increases as it remains in the ready queue; hence, will eventually execute to completion.

Jan/01

Copyright © 1998-2001 by Eskicioglu & Marsland

Scheduling 17

An example—Priority



Jan/01

Copyright © 1998-2001 by Eskicioglu & Marsland

Scheduling 18

Comparison of scheduling policies

Unfortunately, the performance of scheduling policies vary substantially depending on the characteristics of the jobs entering the system (job mix), thus it is not practical to make definitive comparisons.

For example, FCFS performs better for “long” processes and tends to favor CPU-bound jobs. Whereas SJF is risky, since long processes may suffer from CPU starvation. Furthermore, FCFS is not suitable for “interactive” jobs, and similarly, RR is not suitable for long “batch” jobs.

The (processing) overhead of FCFS is negligible, but it is moderate in RR and can be high for SJF.

Jan/01

Copyright © 1998-2001 by Eskicioglu & Marsland

Scheduling 19

Other policies

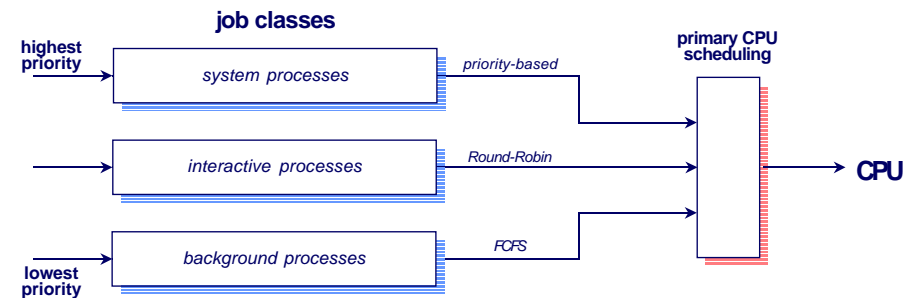
As discussed earlier, the previous policies cannot efficiently handle a mixed collection of jobs (e.g., batch, interactive, and CPU-bound). So, other schemes were developed:

- Multi-level queue scheduling
- Multi-level feedback queue scheduling

Jan/01

Multi-level queue

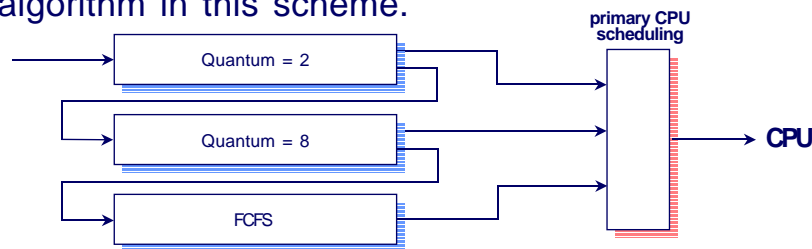
Multi-Level Queue (MLQ) scheme solves the mix job problem by maintaining separate “ready” queues for each type of job class and apply different scheduling algorithms to each.



Jan/01

Multi-level feedback queue

This is a variation of MLQ, where processes (jobs) are not permanently assigned to a queue when they enter the system. In this approach, if a process exhausts its time quantum (i.e., it is CPU-bound), it is moved to another queue with a longer time quantum and a lower priority. The last level usually uses FCFS algorithm in this scheme.



Jan/01

Real-time computing

A real-time (R-T) system controls or monitors external events that have their own timing requirements, thus a R-T operating system should be tailored to respond to these activities. Examples of R-T applications include control of laboratory experiments, process control, robotics, video games, and telecommunications.

An OS designed to support batch, interactive, or time-sharing is different from that for a R-T. One basic difference is the way external events are handled.

Jan/01

Real-time tasks

A process (usually referred to as a *task* in R-T systems) that controls or reacts to an event in a R-T system is associated with a *deadline* specifying either a start time or a completion time. Depending on its deadline, a process can be classified as one of the following:

- Hard real-time
 - must meet the deadlines (time-critical).
- Soft real-time
 - meeting a deadline is desirable (better performance).

Multiprocessor scheduling

- thread scheduling
- load sharing/balancing
- gang scheduling
- dedicated processor assignment
- dynamic scheduling