Scheduling

(Chapter 2.5, Tanenbaum)

Introduction

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

- **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 dispatcher is the mechanism.

*Policy versus Mechanism

Separate what is allowed to be done with how it is done

- a process knows which of its children threads are important and need priority

Scheduling algorithm parameterized

- mechanism in the kernel

Parameters filled in by user processes

- policy set by user process

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 of a computer system that are scheduled before use, the CPU is the far most important.

But, other criteria may be important too (e.g., memory).

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

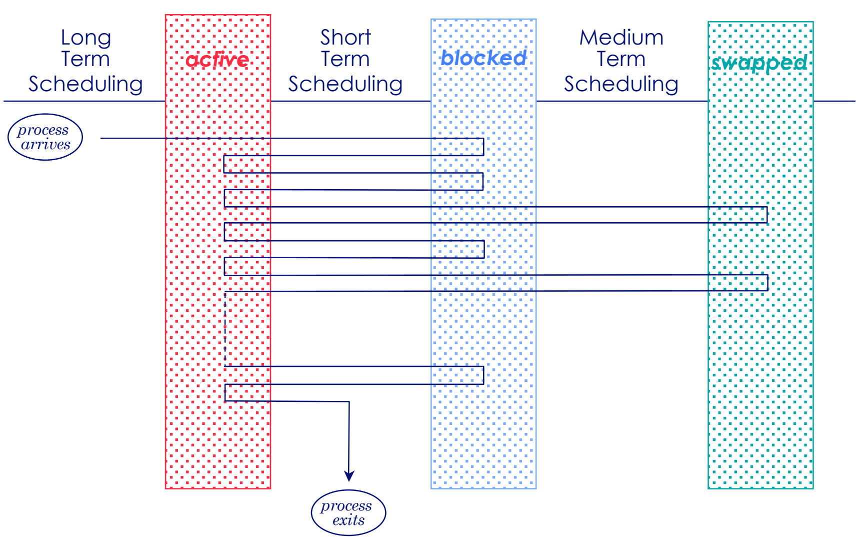
In general, (job) scheduling is performed in three stages: short-, medium-, and long-term. The activity frequency of these stages are implied by their names.

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).
e.g., job admission control.

Medium-term scheduling involves suspending or resuming processes by swapping (rolling) them out of *or* in to 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 or blocks the incoming (batch) jobs from entering the ready queue otherwise.

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.

The short-term scheduler must be fast!

Short-term scheduling criteria

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

- CPU utilization
- I/O device throughput
- Total service time (e.g., turnaround time)
- Responsiveness (e.g., for keystroke)
- Fairness
- Deadlines (e.g., for real-time systems)

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 (i.e., trade-offs)

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.

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.

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.

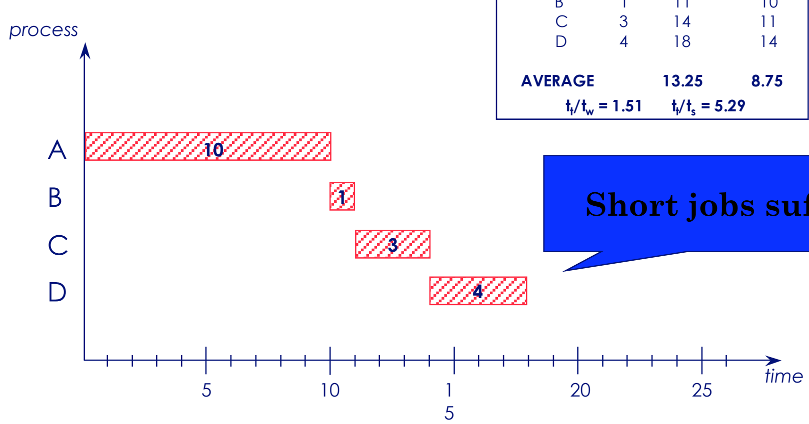
First-Come-First-Served (FCFS)

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."

An example—FCFS

process	*service time t_s*	*turnaround time t_t*	*waiting time t_w*
A	10	10	0
B	1	11	10
C	3	14	11
D	4	18	14
AVERAGE		**13.25**	**8.75**
	t_t/t_w = 1.51	**t_t/t_s = 5.29**	



Short jobs suffer!

Shortest Job First (SJF)

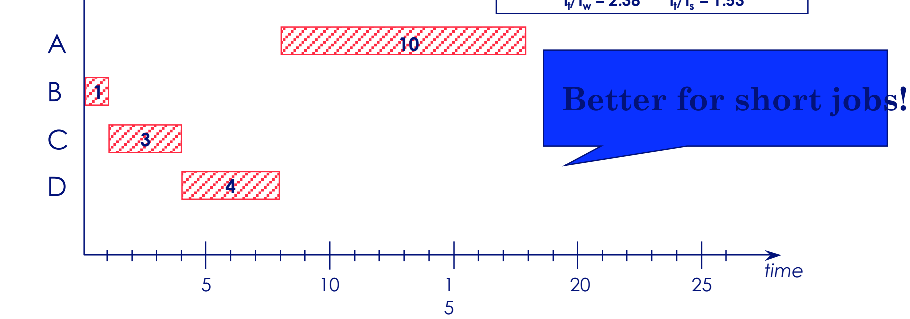
SJF policy selects the job with the shortest (expected) processing time first. Shorter jobs are always executed before long jobs.

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 for the CPU when there is a steady supply of short jobs.

An example—SJF

process	service time t_s	turnaround time t_t	waiting time t_w
A	10	18	8
B	1	1	0
C	3	4	1
D	4	8	4
AVERAGE		**7.75**	**3.25**
	t_t/t_w = 2.38	**t_t/t_s = 1.53**	



Round-Robin (RR)

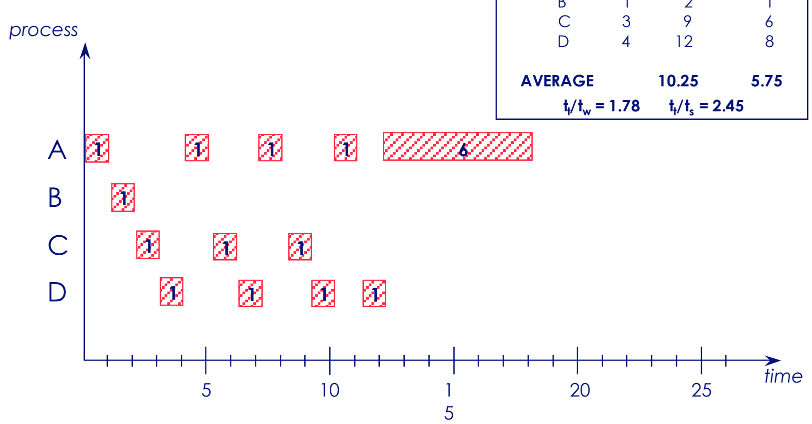
RR reduces the penalty that short jobs suffer with FCFS by preempting running jobs periodically. The CPU suspends the current job when the reserved *quantum* (*time-slice*) is exhausted. The job is then put at the end of the ready queue if not yet completed.

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.

An example—RR

Quantum = 1

process	service time t_s	turnaround time t_t	waiting time t_w
A	10	18	8
B	1	2	1
C	3	9	6
D	4	12	8
AVERAGE		**10.25**	**5.75**
	t_t/t_w = 1.78	**t_t/t_s = 2.45**	



Priority-based

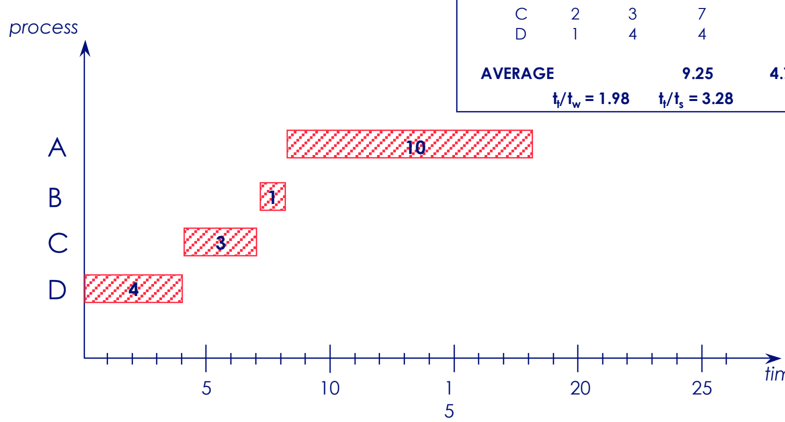
Each process is assigned a priority (e.g., a number). The ready list contains an entry for each process ordered by its priority. The process at the beginning of the list (highest priority) is picked first.

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.

An example—Priority

process	*priority*	*service time t_s*	*turnaround time t_t*	*waiting time t_w*
A	4	10	18	8
B	3	1	8	7
C	2	3	7	4
D	1	4	4	0
AVERAGE			**9.25**	**4.75**
		t_t/t_w = 1.98	**t_t/t_s = 3.28**	



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. The results depend on the specific workload.

For example, FCFS performs better for "long" processes and tends to favor CPU-bound jobs. Whereas SJF is risky as 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(er) for SJF.

Other policies

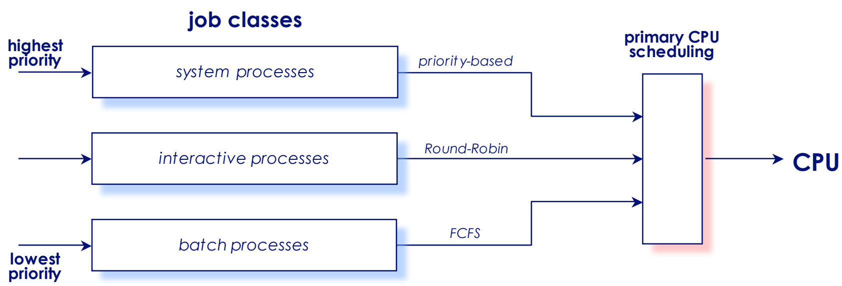
As discussed earlier, the previous policies cannot efficiently handle a mixed collection of jobs (e.g., batch, interactive, and CPU-bound). So, other schemes were developed:

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

Ken Thompson says: A good interactive system is the best compromise. Why?

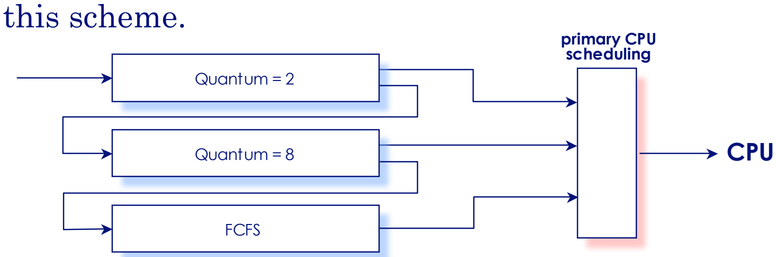
Multi-level queue

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.



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.



Real-time computing

The typical Unix system is a "best effort" system, not real time.

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 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.

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).
  - often, an external factor determines the deadlines
- Soft real-time
  - meeting a deadline is desirable (better performance).

Compare real-time tasks vs. scheduling an entire workload.

Multiprocessor scheduling

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