Scheduling

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).

Scheduling

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.

Scheduling

Long-Term and Medium-Term Scheduling

Acting as the primary resource allocator, the longterm 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’’.

Scheduling

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.

Scheduling

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

Scheduling

Algorithms

The following are some common scheduling algorithms:

Non-preemptive	Preemptive
First-Come-First-Served (FCFS)	Round-Robin (RR)
Shortest Job First (SJF)	Priority
Background "batch Good for jobs"	**Foreground "Good for" interactive jobs**

In general, scheduling policies may be 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

First Come First Serve (FCFS)

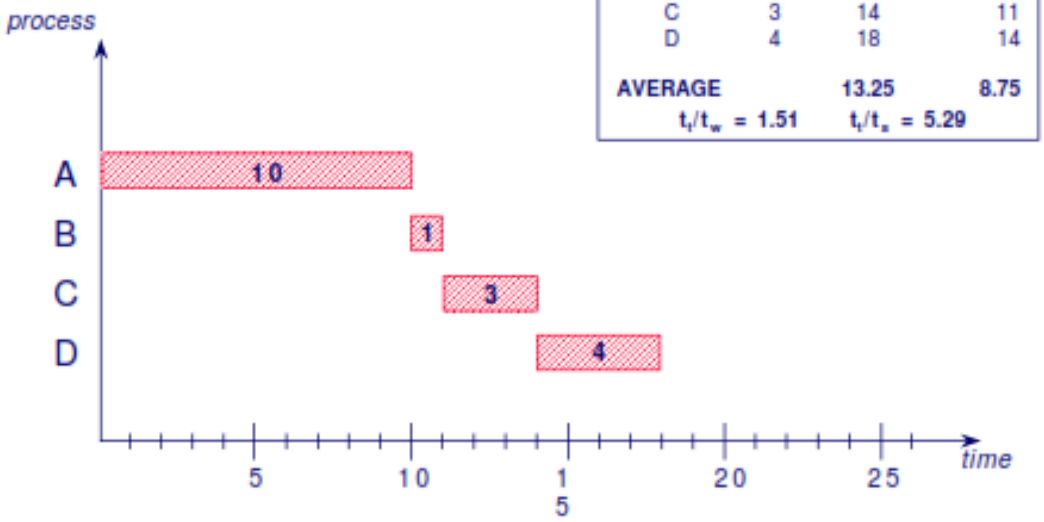
FCFS, also known as First-In-First-Out (FIFO), is the simplest scheduling policy.

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	

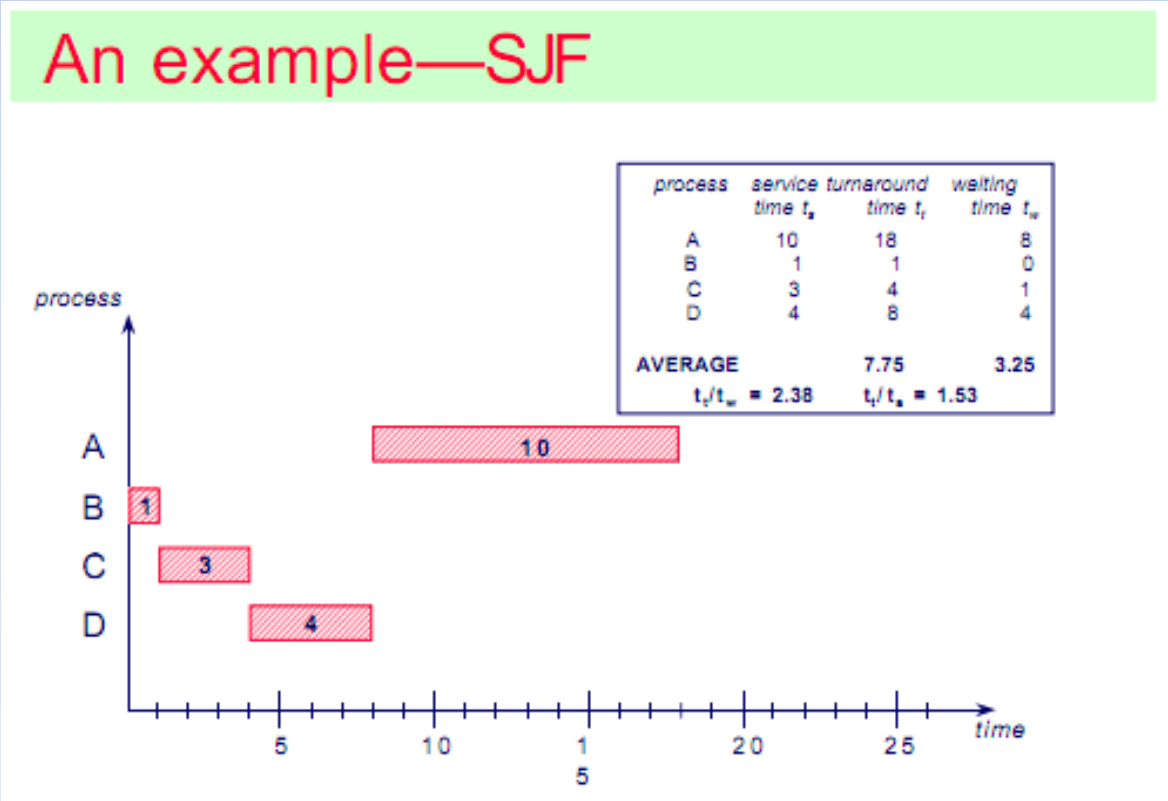


Scheduling

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.

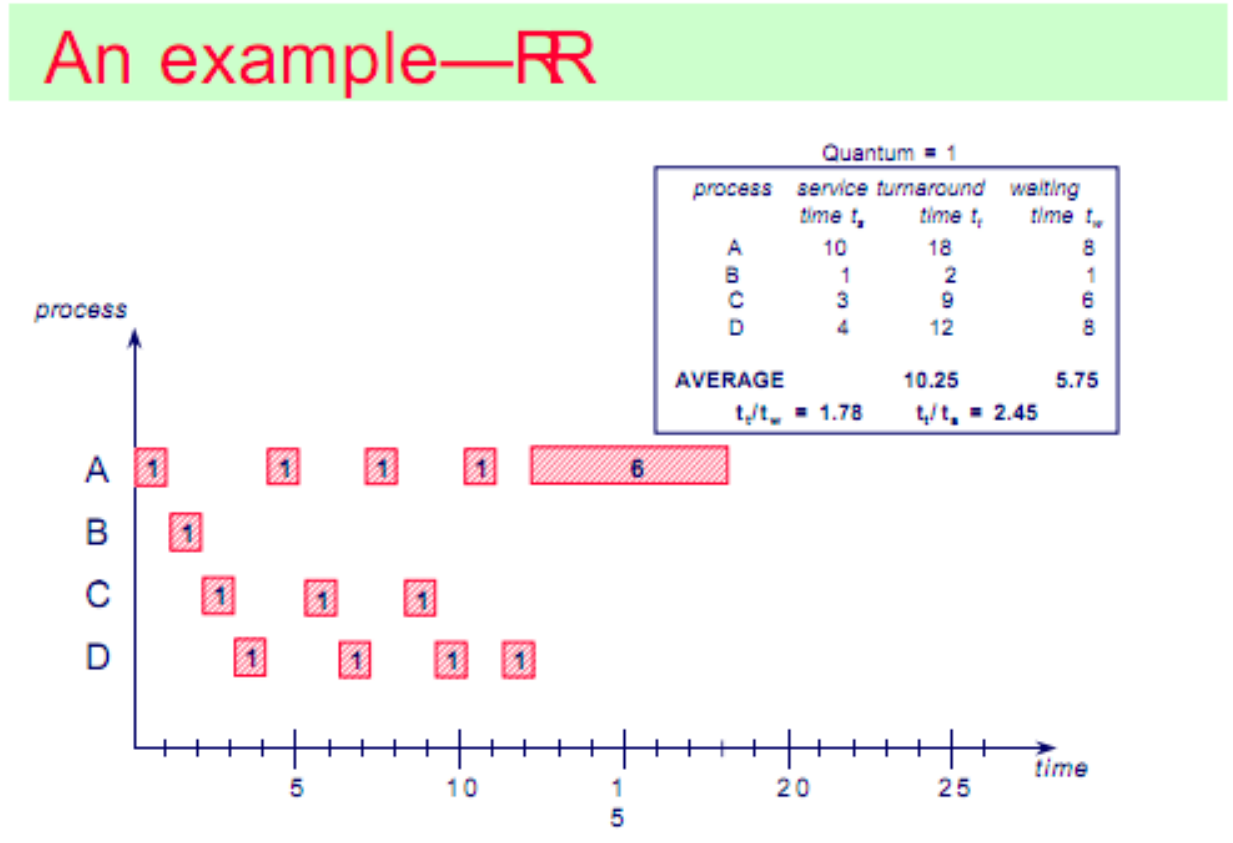


Scheduling

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) is exhausted. The job is then time-slice (quantum 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.

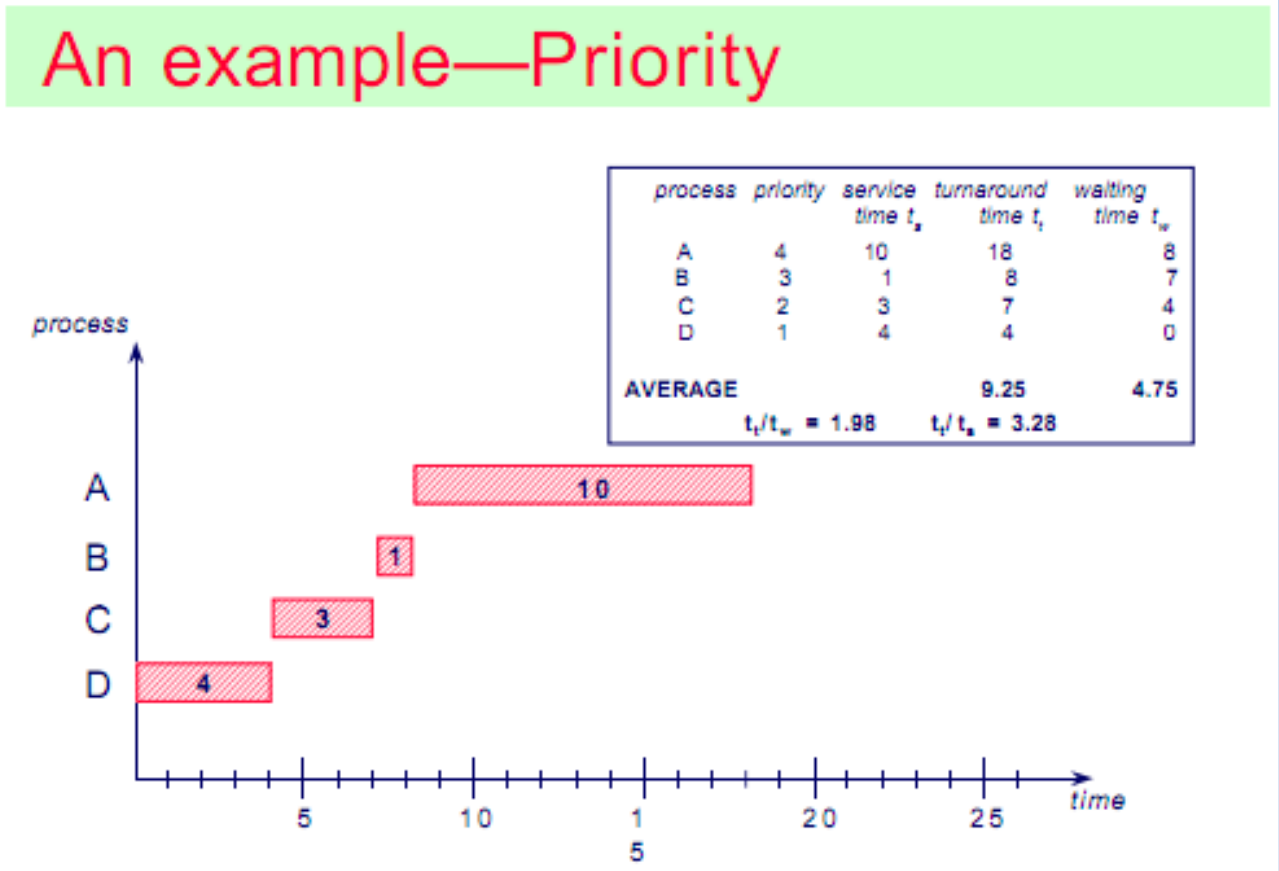


Scheduling

Priority

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.



Scheduling

Comparisons

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 “interactive” jobs, and similarly, RR is not suitable for long “batch” jobs.

Deadlocks

Deadlock - Occurs when resources needed by one process are held by some other waiting process.

Deadlock not only occurs in OS.
Kansas state legislature in the early 20th century passed the following legislation:

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone. "

Assume we have the following operating system:

1. Finite number of resources to be distributed among some number of competing processes

2. Resources may be of several types and there may be several instances of each

Deadlocks

3. When a process requests a resource any instance of that resource will satisfy the process

4. Processes can

   a. request a resource
   b. use the resource
   c. release the resource

5. A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

   1. Same resource type - three tape drives, three processes request a tape drive then they each request another. Dining philosophers request chopsticks held by another.

   2. Different resource type - process A has a printer process B has a file, Each requests the other's resource.

Deadlocks

Four Necessary Conditions for Deadlock

1. Mutual exclusion: At least one resource is not sharable, i.e. can only be used by one process at a time
2. Hold and wait: A process holds at least one resource and requests resources held by other processes
3. No preemption: resource cannot be preempted, it must be voluntarily released by the process.
4. Circular wait: Given a set of processes { P1, P2, P3, …Pn} P1 has a resource needed by P2, P2 has a resource needed by P3, …, Pn has a resource needed by P1.

Deadlocks

Deadlock Avoidance

Avoidance

- Provide information in advance about what resources will be needed by processes to guarantee that deadlock will not happen
- System only grants resource requests if it knows that the process can obtain all resources it needs in future requests
- Avoids circularities (wait dependencies)

Tough

- Hard to determine all resources needed in advance
- Good theoretical problem, not as practical to use