**Chapter 7: Scheduling: Introduction**

In this chapter, we will learn about a series of **scheduling policies**, or **disciplines** of the OS.

**7.1 Workload Assumptions**

The **workload** assumptions we make here are the most unrealistic, but we will relax them as we go, and eventually refers them as a **fully-operational scheduling discipline**. We will make the following assumptions about the processes, sometimes called jobs, that are running in the system:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (they perform no I/O)
5. The run-time of each job is known.

Many of the above assumptions are unrealistic.

**7.2 Scheduling Metrics:**

The **turnaround time** of a job is defined as the time at which the job completes minus the time at which the job arrived.

*Tturnarround = Tcompletion - Tarrival*

Since we are assuming every job arrives at the same time, *Tarrival* = 0 and thus, *Tturnarround = Tcompletion*. This metric is a **performance** metric.

Another metric is **fairness**. Performance and fairness are often at odds in scheduling; a scheduler, for example, may optimize performance but at the cost of preventing a few jobs from running, thus decreasing fairness.

**7.3 First In, First Out (FIFO):**

Whatever that comes first gets processed first.

To assess this scheduling rule, we consider **average turnaround time**. Consider a combination of three jobs, A, B and C, arrives at the system, each run for 10 seconds. Let’s assume that A, B and C are processed in that order.

Chart, scatter chart

Description automatically generated

The turnaround time of A, B and C will be 10, 20 and 30, respectively. This gives the average turnaround time to be 20.

However, if we relax the first assumption so that the processing time of every job is different, things will get worse. For example, let A runs in 100 seconds, then in the same order as before, the average turnaround time would be 110. This problem is generally referred to as the **convoy effect**, where a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer.

Chart

Description automatically generated

**7.4** **Shortest Job First (SJF):**

For the previous example (figure 7.2), the schedule would now be

Chart

Description automatically generated

The average turnaround time would now be 50.

Virtually all modern schedulers are **preemptive**, and quite willing to stop one process from running in order to run another. In the old day, a number of non-preemptive schedulers were developed where they would run each job to completion before considering whether to run a new job.

This algorithm is optimal scheduling algorithm. However, it is still unrealistic. If we relax assumption 2 and jobs can arrive at any time, then if A arrives first (at 0), and then B (at 10) and C (at 10), then the average turnaround time would be 103.33.

**7.5 Shortest Time-to-Completion First (STCF)**:

To solve that previous problem, we need to relax assumption 3, so that jobs do not have to run to completion. Now, the scheduler can preempt job A and decide to run other job and continue job A later.

Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one. The schedule is now depicted in figure 7.5.

Chart

Description automatically generated

The average turnaround time will now be 50.

**7.6 A New Metric: Response Time.**

We define **response time** as the time from when the job arrives in a system to the first time it is scheduled. More formally:

*Tresponse = Tfirstrun - Tarrival*

For example, if we had the schedule from Figure 7.5 (with A arriving at time 0, and B and C at time 10), the response time of each job is as follows: 0 for job A, 0 for B, and 10 for C (average: 3.33).

STCF and related disciplines are not particularly good for response time. If three jobs arrive at the same time, for example, the third job has to wait for the previous two jobs to run in their entirety before being scheduled just once.

How can we build a scheduler that is sensitive to response time?

**7.7 Round Robin (RR):**

The basic idea is simple: instead of running jobs to completion, RR runs a job for a **time slice** (sometimes called a **scheduling quantum**) and then switches to the next job in the run queue. It repeatedly does so until the jobs are finished.

RR is sometimes called **time-slicing**. Note that the length of a time slice must be a multiple of the timer-interrupt period. Note that the cost of context switching does not arise solely from the OS actions of saving and restoring a few registers.

**Amortization** is a technique to increase the time slice so that the percentage of time spent on context switching is small.

Chart, bar chart

Description automatically generated

In figure 7.7, the average response time of RR is 1, while for SJF, it is 5.

RR, with a reasonable time slice, is thus an excellent scheduler if response time is our only metric. However, the turnaround time for RR is quite bad (avg of 14 in figure 7.7).

More generally, any policy (such as RR) that is **fair**, i.e., that evenly divides the CPU among active processes on a small time scale, will perform poorly on metrics such as turnaround time. This type of trade-off is common in systems.

**7.8 Incorporating I/O:**

Now, we will relax assumption 4.

A scheduler clearly has a decision to make when a job initiates an I/O request, because the currently-running job won’t be using the CPU during the I/O; it is **blocked** waiting for I/O completion. If the I/O is sent to a hard disk drive, the process might be blocked for a few milliseconds or longer, depending on the current I/O load of the drive. Thus, the scheduler should probably schedule another job on the CPU at that time.

The scheduler also has to make a decision when the I/O completes. When that occurs, an interrupt is raised, and the OS runs and moves the process that issued the I/O from blocked back to the ready state.

Chart

Description automatically generated

To understand this issue better, let us assume we have two jobs, A and B, which each need 50 ms of CPU time. However, there is one obvious difference: A runs for 10 ms and then issues an I/O request (assume here that I/Os each take 10 ms), whereas B simply uses the CPU for 50 ms and performs no I/O. Figure 7.8 obviously shows a poor use of resources, while figure 7.9 allows **overlap**, with the CPU being used by one process while waiting for the I/O of another process to complete.

**7.9 No More Oracle**

Remove the final assumption. Now the problem is way harder than removing other assumptions.

We have also seen how we might incorporate I/O into the picture, but have still not solved the problem of the fundamental inability of the OS to see into the future. Shortly, we will see how to overcome this problem, by building a scheduler that uses the recent past to predict the future. This scheduler is known as the **multi-level feedback queue**, and it is the topic of the next chapter.