CI583: Data Structures and Operating Systems
Scheduling

Scheduling

We have said that the main purpose of an OS is to *manage resources*, and one of the ways of doing this is to ensure that processes and threads have access to a "fair" amount of processor time.

Managing the access to processors is called scheduling. This is a dynamic process – the scheduler has to respond immediately to demands for processor time, as threads move in and out of the runnable state.

Scheduling

At the highest level, we can consider that we have a list of runnable threads which we want to sort so that the most urgent ones are at the front of the list. But what do we mean by "urgent"? That could be a question of:

- giving priority to interactive threads,
- giving priority to system-level threads,
- maximising the number of threads processed per unit of time,
- a combination of the above.

Scheduling strategies

The strategy we use will depend on the type of system we are designing:

1. Simple batch systems (SBS): Each process runs to completion and the job of the scheduler is to pick the next task to be run.

   The two main considerations are *system throughput* and *average wait time*.

Scheduling strategies

2. Multiprogrammed batch systems: Same as SBS but several jobs are run concurrently.

   The considerations from the SBS are supplemented with the questions of *how many jobs should be running at any one time* and *how the processor time is apportioned amongst running jobs*.

Scheduling strategies

3. Time-sharing systems: here the main question becomes apportioning time to the jobs that are ready to execute – the runnable threads.

   The main concern is *response time* – the time from when a command is given to when it is completed. *Short requests should be completed quickly*.

Scheduling strategies

4. Shared servers: A single computer being used by many clients.

   The question arises here of *giving each client a reasonable apportionment of time*, instead of focusing only on runnable threads.

Scheduling strategies

5. Real-time systems: Can be *soft* or *hard*.

   An example of soft real-time would be *video processing* – data must be processed in a strictly synchronised fashion and as efficiently as possible, but some lag is not a disaster.

   An example of hard real-time would be *autonomous vehicle software* that alters the direction of a car – certain commands *must* be handled in a timely way.

Simple Batch Systems

On an SBS, jobs are run one at a time and, when a job ends, we can choose between several queued jobs to decide which to run next.

There are two approaches we could take:

1. First-in-first-out (FIFO): jobs are executed in the order they were submitted to the system.
2. Shortest-job-first (SJF): some (relatively crude) measure is used to decide how long a job is going to take, and the shortest will be executed first.

Simple Batch Systems

FIFO might seem fairest, but if we take *average throughput* as the measure of effectiveness for our scheduler, then SJF will perform best.

However, without further measures, SJF could mean that a very long job is queued indefinitely, so we need to make sure that doesn't happen.

Multiprogrammed Batch Systems

An MBS is essentially the same as SBS except that several jobs are held in memory at one time and time-slicing is used to switch focus between the currently active job.

To do this we need to decide how many jobs to hold in memory at any one time and to decide on a time quantum – the amount of time to allocate to a job before it is preempted in favour of the next job.

A solution which gets round some of the problem with SJF is to hold multiple queues of relatively short and relatively long jobs, so a short and a long job will be held in memory at any one time.

Time-Sharing Systems

A TSS is one which has several users logged on at any one time.

The main concern of the scheduler is that the system should *feel* responsive. A job which ought to be short should be short.

Time-Sharing Systems

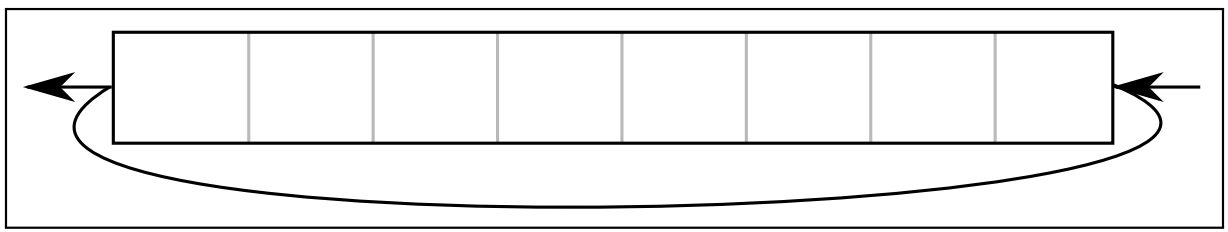
If a job that normally takes 3 or 4 minutes, such as compiling some code, takes 5 minutes, users probably won't mind.

But if a job that should be very quick, such as opening a file in a word processor, takes 1 minute then the users will think the system is slow.

Time-Sharing Systems

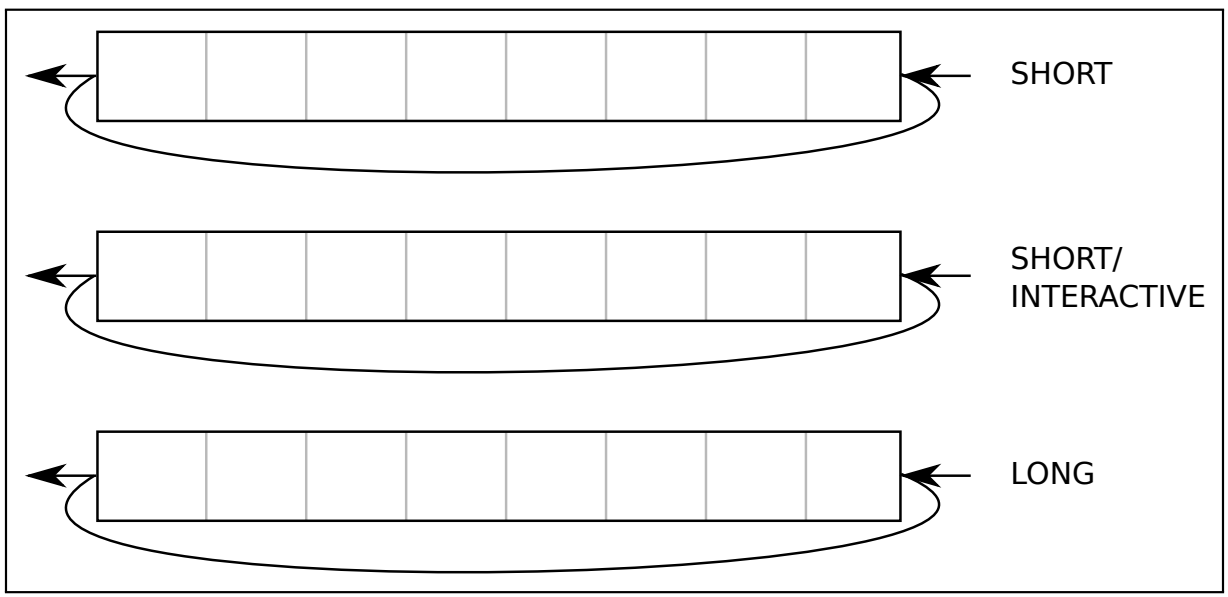
A scheduling strategy for a TSS should favour short and interactive operations at the possible expense of longer ones.

As a first attempt, consider a time-sliced scheduler that uses a round robin strategy. When a thread uses up its time quantum, it is moved to the back of the queue and the next thread gets to run.



Time-Sharing Systems

As we want to prioritise short, interactive operations, we can modify this strategy to assign a priority to each thread. We can use this notion of high/low priority to move short, interactive threads to the top of the list or to maintain multiple queues.



Time-Sharing Systems

Determining the level of interactivity of a process will necessarily involve some guesswork.

A more effective algorithm is to reduce the priority of a thread each time it uses a time quantum.

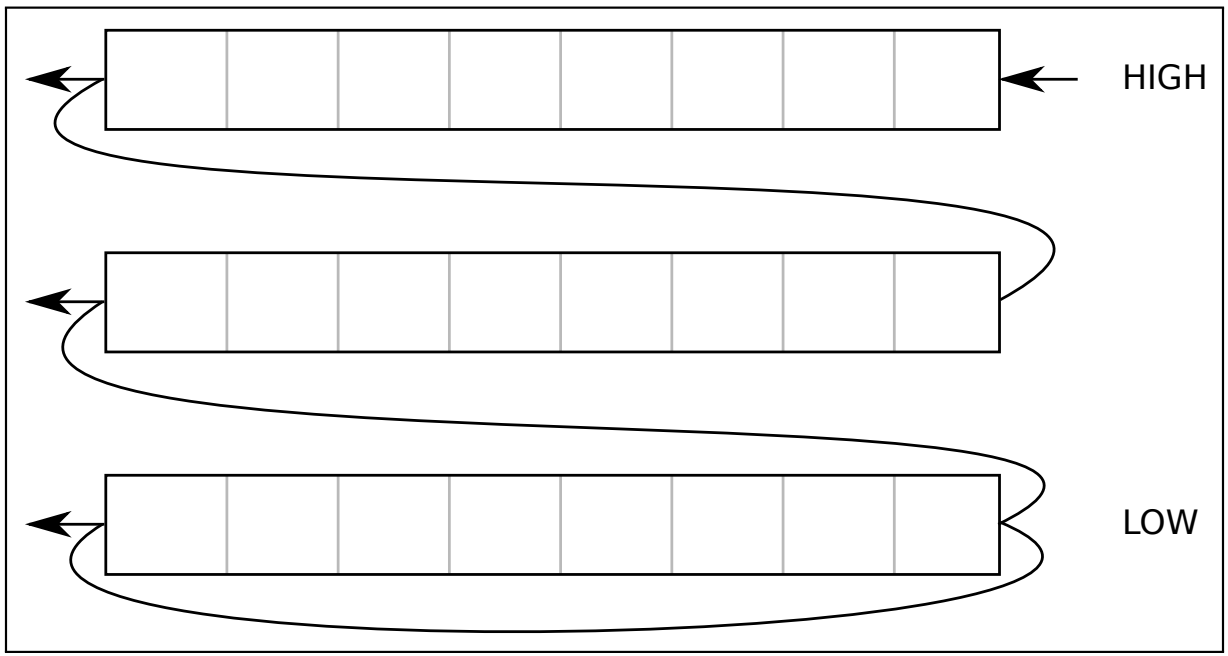
Time-Sharing Systems

The first time a thread runs, it does so at the highest priority level and, presuming it isn't completed, its priority is reduced.

The next time it runs, it's priority is reduced again, and so on. This is called a multilevel feedback queue.

In this system, threads in low priority queue can only run if there are no threads in a higher queue.

Time-Sharing Systems



Time-Sharing Systems

However, we can improve on this by taking other factors than the number of time quanta consumed into account.

When we run a thread, we can modify its priority based on how long it has been waiting since it last ran (e.g. it may have been in the *paused* state for a long time, waiting for an IO operation to end).

Time-Sharing Systems

We can sum this up by saying *a thread's priority gets worse while it is running, and better while it is waiting*.

This is the strategy used by the schedulers in Windows and Linux today.

At the same time, each OS allows the user some way of stating how important they consider a process and its threads to be (on Linux, this is the `nice` command).