Operating Systems 2230

Computer Science & Software Engineering

Lecture 7: Uniprocessor scheduling

In a single processor multi-programming system, multiple processes are contained within memory (or its swapping space).

Processes alternate between executing (**Running**), being able to be executed (**Ready**), waiting for some event to occur (**Blocked**), and swapped-out (**Suspend**).

A significant goal is to keep the processor busy, by "feeding" it processes to execute, and always having at least one process able to execute.

The key to keeping the processor busy is the activity of *process scheduling*, of which we can categorise three main types:

Long-term scheduling: the decisions to introduce new processes for execution, or re-execution.

Medium-term scheduling: the decision to add to (grow) the processes that are fully or partially in memory.

Short-term scheduling: the decisions as to which (**Ready**) process to execute next.

The Types Of Scheduling

The primary goal of process scheduling is to determine the execution order of processes whilst attempting to maximise some objective functions measuring:

- processor efficiency,
- job turn-around time, and
- perceived response time.

The scheduling activities can be considered as state transitions in our process diagrams (Figure 9.1; all figures are taken from Stallings' web-site).

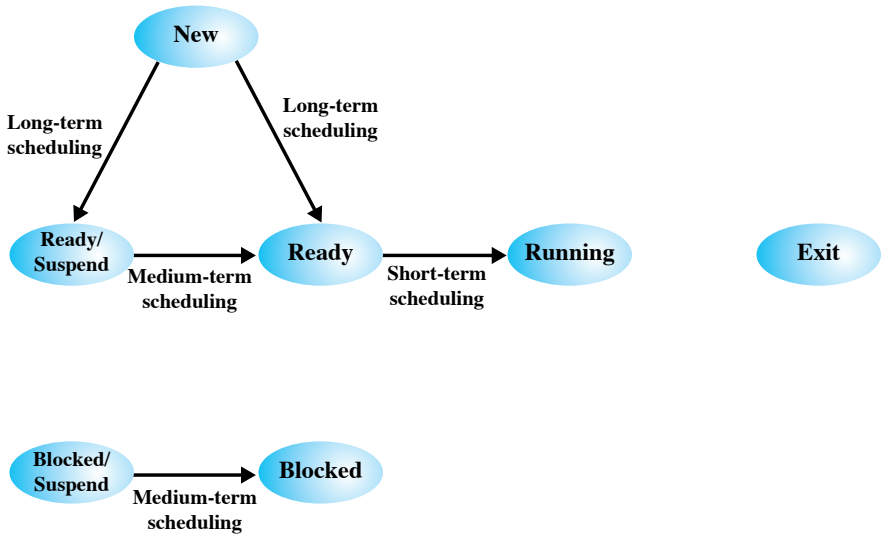

Figure 9.1 Scheduling and Process State Transitions

Scheduling Queues

Scheduling affects the performance of a system by determining (indirectly) how long processes have to wait until they may be executed.

Because there will typically be more than one process in any of the process states at any one time, the operating system must maintain queues of waiting processes (Figure 9.3).

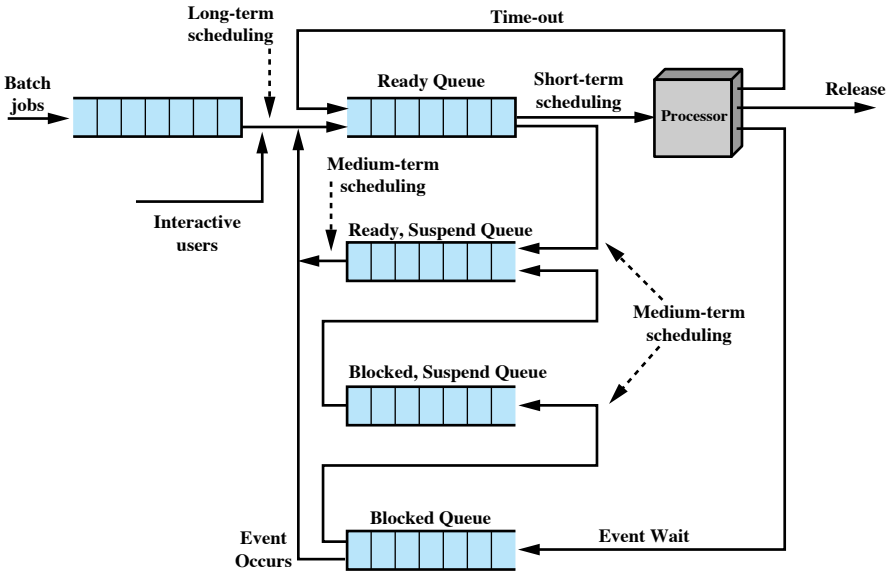

Figure 9.3 Queuing Diagram for Scheduling

Good scheduling is the "art" of managing the process state queues so as to minimise delays and optimise performance.

Long-term Scheduling

Long-term scheduling determines which programs, if any, may be admitted to the system as new processes.

We know that this is supported in different fashions in different operating systems: Linux permits new processes to be created only from old (via *fork()*),

whereas Windows-NT creates a process directly from a program image (via *CreateProcess()*).

Once a new process is accepted (and partially created) it may enter the scheduling queues in one of two places:

- If all resources (such as memory requirements) are initially fully available to the new process, it may be admitted to the tail of the **Ready** queue.
- If not all resources are immediately available, the new process may be instantiated in the **Blocked-Suspended** queue until those resources are provided.

The long-term scheduler is executed relatively infrequently. If the required degree of multiprogramming has been reached (i.e. a steady state of just the right number of currently-executing processes), the long-term scheduler will be invoked only when a running process terminates.

Alternatively, when the CPU becomes sufficiently idle, the long-term scheduler may be permitted to introduce a new process.

Interactive versus Batch Scheduling

These conditions of a steady degree of multiprogramming are now rarely seen on single-user, interactive, workstations: nearly all requests for new processes are satisfied immediately.

Unsatisfied requests in an interactive environment simply fail: *fork()* or *CreateProcess()* return an indication of failure. The requests are not queued.

But on some long running *batch scheduling environments*, the long-term scheduling choice may either be:

- simply first-come, first-served (FCFS), in which short running processes are "held up" by longer running ones, or

- more judiciously based on the expected resource requirements (CPU time and memory) of the new processes.

Supercomputing environments still employ batch scheduling in which long-running tasks will be scheduled so as to minimise their impact (late at night).

Some operating systems may also admit new processes based on their expected requirements: resource requirement histories are maintained.

Medium-term Scheduling

Medium-term scheduling is really part of the *swapping* function of an operating system.

The success of the medium-term scheduler is based on the degree of multiprogramming that it can maintain, by keeping as many processes "runnable" as possible.

As we have seen, more processes can remain executable if we reduce the *resident set size* of all processes.

The medium-term scheduler makes decisions as to which pages of which processes need stay resident, and which pages must be swapped out to make room for other processes.

The sharing of some pages of memory, either explicitly or through the use of shared or dynamic link libraries, complicates the task of the medium-term scheduler, which now must maintain reference counts on each page.

The responsibilities of the medium-term scheduler may be further complicated in some operating systems, in which some processes may request (demand?) that their pages remain locked in physical memory: see *man mlock* and *man munlock*.

Short-term Scheduling

The long-term scheduler runs relatively infrequently, when a decision must be made as to the admission of new processes: maybe on average every ten seconds.

The medium-term scheduler runs more frequently, deciding which process's pages to swap to and from the swapping device: typically once a second.

The short-term scheduler, often termed the *dispatcher*, executes most frequently (every few hundredths of a second) making *fine-grained* decisions as to which process to move to **Running** next.

The short-term scheduler is invoked whenever an event occurs which provides the opportunity, or requires, the interruption of the current process and the new (or continued) execution of another process.

Such opportunities include:

- clock interrupts, provide the opportunity to reschedule every few milliseconds,
- expected I/O interrupts, when previous I/O requests are finally satisfied,
- operating system calls, when the running process asks the operating system to perform an activity on its behalf, and
- unexpected, asynchronous, events, such as unexpected input, user-interrupt, or a fault condition in the running program.

Short-term Scheduling Criteria

The success of the short-term scheduler can be characterised by its success against

user-oriented criteria under which a single user (selfishly) evaluates their perceived response, or

system-oriented criteria where the focus is on efficient global use of resources such as the processor and memory. A common measure of the system-oriented criteria is *throughput*, the rate at which tasks are completed.

On a single-user, interactive operating system, the user-oriented criteria take precedence: it is unlikely that an individual will exhaust (or even tax) resource consumption, but responsiveness remains all important.

On a multi-user, multi-tasking system, the global system-oriented criteria are more important as they attempt to provide fair scheduling for all, subject to priorities and available resources.

User-oriented Scheduling Criteria

Response time

In an interactive system this measures the time between submission of a new process request and the commencement of its execution. Alternatively, it can measure the time between a user issuing a request to interactive input (such as a prompt) and the time to echo the user's input or accept the carriage return.

Turnaround time

The time between submission of a new process and its completion. Depending on the mixture of current tasks, two submissions of identical processes will likely have different turnaround times.

Turnaround time is the sum of execution and waiting times.

Deadlines

In a genuine real-time operating system, hard deadlines may be requested by processes. These either demand that the process is completed with a guaranteed

upper-bound on its turnaround time, or provide a guarantee that the process will receive the processor in a guaranteed maximum time in the event of an interrupt.

A real-time long-term scheduler should only accept a new process if it can guarantee required deadlines. In combination, the short-term scheduler must also meet these deadlines.

Predictability

With lower importance, users expect similar tasks to take similar times. Wild variations in response and turnaround times are distracting.

System-oriented Scheduling Criteria

Throughput

The short-term scheduler attempts to maximise the number of completed jobs per unit time.

While this is constrained by the mixture of jobs, and their execution profiles, the policy affects utilisation and thus completion.

Processor utilisation

The percentage of time that the processor may be "fed" with work from **Ready**. In a single-user, interactive system, processor utilisation is very unlikely to exceed a few percent.

Fairness

Subject to priorities, all processes should be treated fairly, and none should suffer processor starvation. This simply implies, in most cases, that all processes

are moved to the ends of their respective state queues, and may not "jump the queue".

Priorities

Conversely, when processes are assigned priorities, the scheduling policy should favour higher priorities.

Priority-Based Scheduling

In most operating systems, processes are assigned priorities, and the short-term scheduler will always choose to execute a higher priority process over a lower priority process.

Instead of a single queue of **Ready** processes, the operating systems maintains a queue for each priority level: RQ0, RQ1, ..., RQn (Figure 9.4).

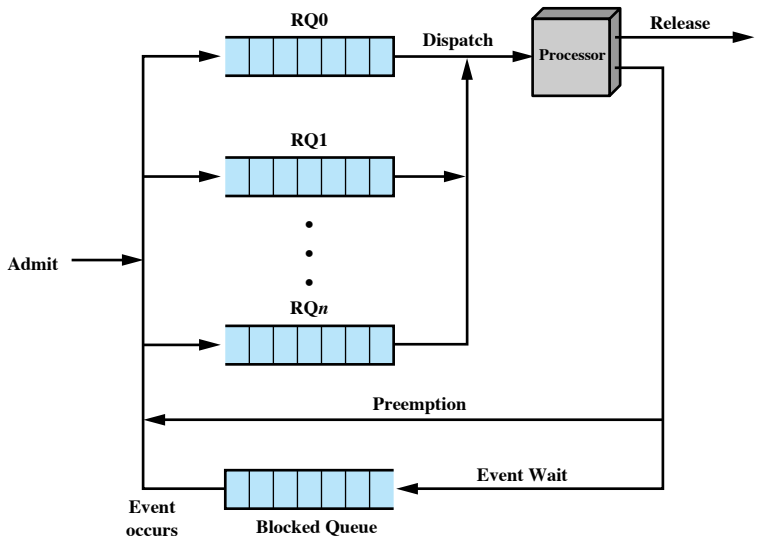

Figure 9.4 Priority Queuing

When a short-term scheduling decision is to be made, the short-term scheduler will examine the (head of the) highest priority queue, RQ0, and continue to schedule and execute processes from there until that queue is exhausted.

There is now a danger of *process starvation*: a low-priority process may always be overlooked due to the continual arrival of new, higher-priority processes.

Scheduling Selection Functions

When all processes are of the same priority, or processes at a given priority level must be decided amongst, the *selection function* determines which process to execute next.

The selection function typically makes its decision based on resource requirements, or on the past execution profile of all processes.

In addition, the *decision mode* decides the moments in time at which the decision function is invoked:

non-pre-emptive: once a process is executing, it will continue to execute until

- it terminates, or
- it makes an I/O request which would block the process, or
- it makes an operating system call.

pre-emptive: the same three conditions as above apply, and in addition the process may be pre-empted by the operating system when

- a new process arrives (perhaps at a higher priority), or
- an interrupt or signal occurs, or
- a (frequent) clock interrupt occurs.

First-Come-First-Served (FCFS) Scheduling

The simplest selection function is the *First-Come-First-Served* (FCFS) scheduling policy:

1. the operating system kernel maintains all **Ready** processes in a single queue,
2. the process at the head of the queue is always selected to execute next,
3. the **Running** process runs to completion, unless it requests blocking I/O,
4. if the **Running** process blocks, it is placed at the end of the **Ready** queue.

Clearly, once a process commences execution, it will run as fast as possible (having 100% of the CPU, and being non-pre-emptive), but there are some obvious problems:

- processes of short duration (typically ones invoked interactively) suffer when "stuck" behind very long-running processes,
- *compute-bound* processes are favoured over *I/O-bound* processes.

We can measure the effect of FCFS by examining:

- the average *turnaround time* of each task (the sum of its waiting and running times), or
- the *normalised turnaround time* (the ratio of running to waiting times).

Round-Robin (RR) Scheduling

We can address some of the problems with FCFS scheduling by introducing *pre-emption*. The **Running** process runs until:

- it terminates, or

- it requests blocking I/O, or
- its time-slice (or time-quantum) expires.

Of critical importance with round-robin scheduling is getting the time-quantum correct: too short, and time is wasted on scheduling housekeeping; too long, and it degenerates to FCFS.

An enhancement to round-robin scheduling tries to further compensate for *I/O-bound* processes. If blocking I/O stalls the **Running** process, it is returned to an auxiliary queue to consume the remainder of its unused time-quantum (Figure 9.7).

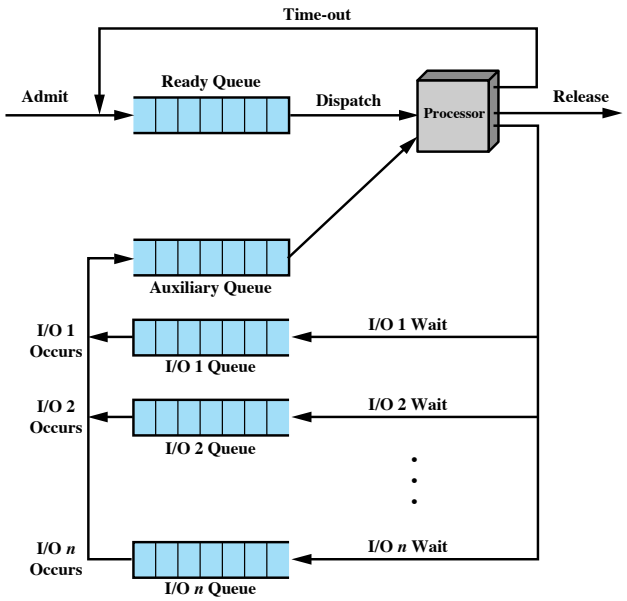

Figure 9.7 Queuing Diagram for Virtual Round-Robin Scheduler

Multi-Level Feedback Scheduling

For most systems, we have no accurate estimate of how long a new process will require to complete. Moreover, in most interactive workstation environments, it is both impractical to specify running times, and we wish short, interactive tasks to complete quickly.

Multi-level feedback scheduling employs both pre-emption and *dynamic priorities* (Figure 9.10):

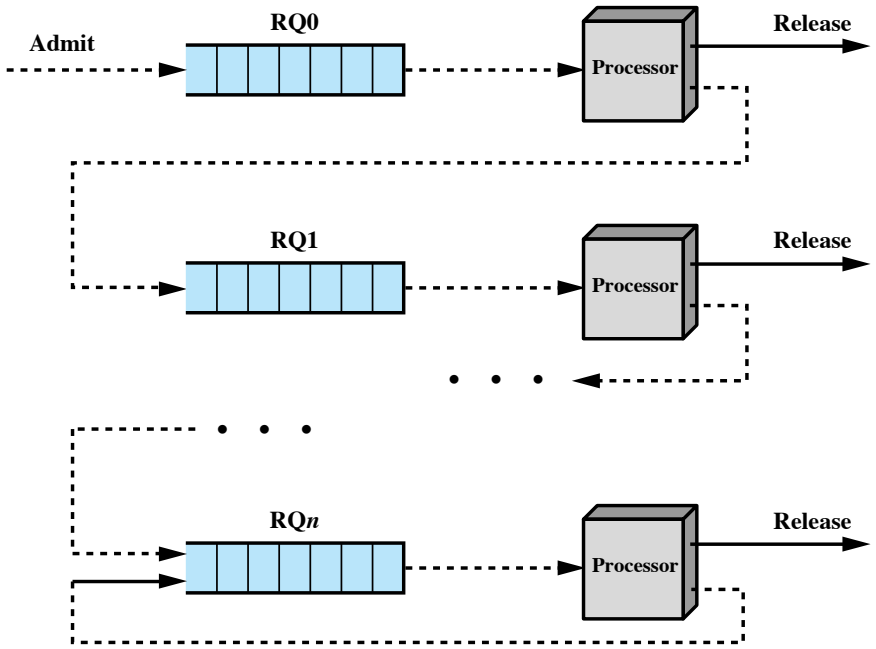

Figure 9.10 Feedback Scheduling

1. the highest priority **Ready** process is always selected next,
2. the process executes until terminated or pre-empted,
3. if pre-empted, its priority is lowered,
4. ... until it reaches, and stays at, the lowest priority.

To avoid possible *process starvation*, enhancements give long-running, lowest priority processes a "boost", by raising their priority again.

The Effect of Scheduling Strategies

Processes P1 ... P5 commence respectively at t = 0s, 2s, 4s, 6s, and 8s. The required running time of each process is known in advance.

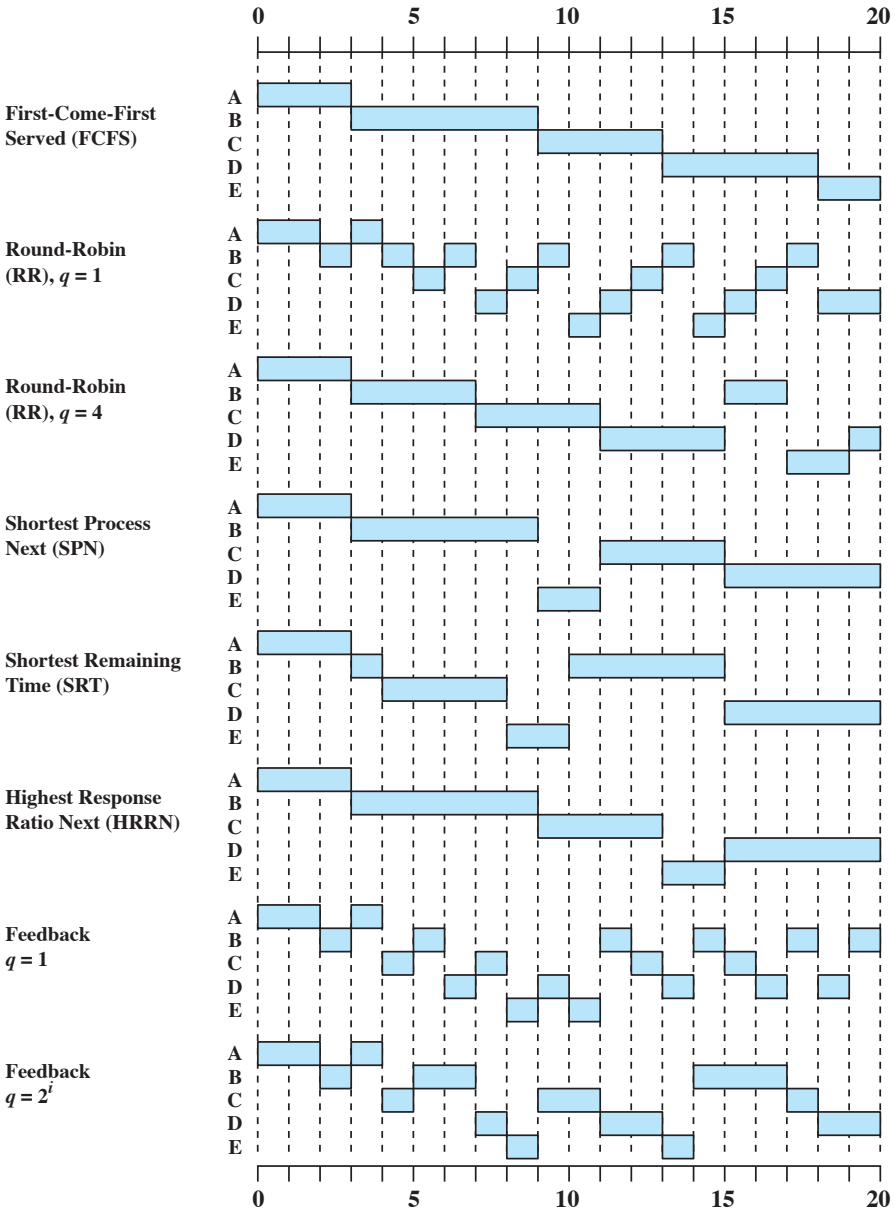

Figure 9.5 A Comparison of Scheduling Policies