Uniprocessor Scheduling

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

Processes alternate between their execution (`Running`), being able to be executed (`Ready`), and waiting for some event to occur – typically I/O, a timer, or an asynchronous event.

We have discussed the use of *process states* to describe the activities of processes.

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 categorize three main types:

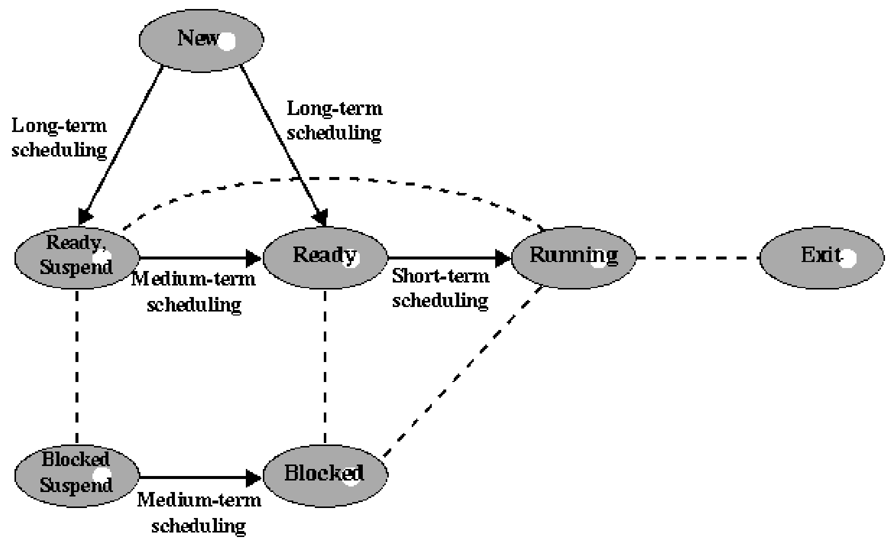
- Long-term scheduling – the decisions to introduce new processes for execution, or re-execution.
- Medium-term scheduling – the decision to add to (grow) the processes that are fully or partially in memory.
- Short-term scheduling – the decisions as to which (`Ready`) process to execute next.

The Types Of Scheduling

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

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

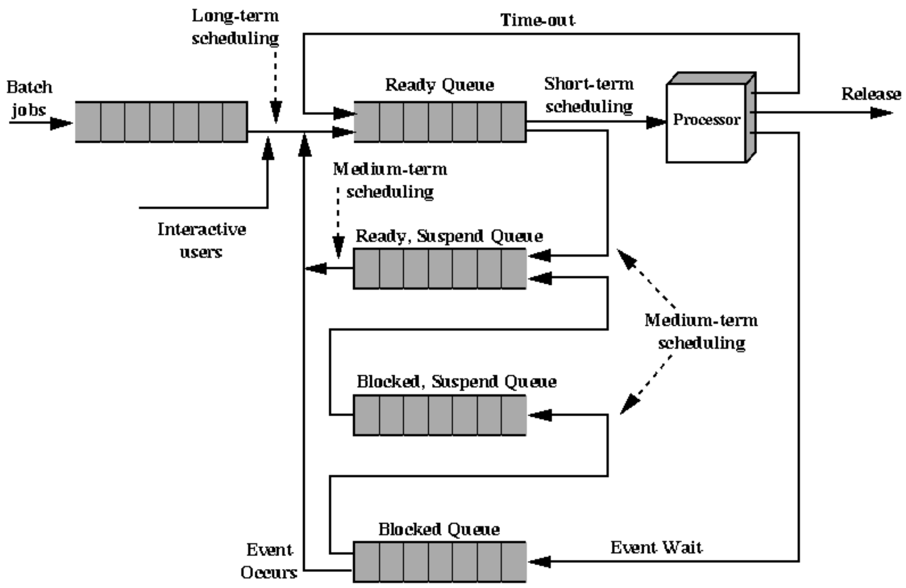
The scheduling activities can easily be considered as state transitions in our process diagrams.



Scheduling Queues

Scheduling affects the performance of a system by (indirectly) determining 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.



Good scheduling is the “art” of managing the process state queues so as to minimize delays and optimize performance.

Long-term Scheduling

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

We know that this is supported in different fashions in different operating systems – LINUX only permits new processes to be created from old (via `fork()`), whereas Windows-NT creates a process 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 made fully available to the new process, it may be admitted to the tail of the `Ready` queue.
- If not all resources can be made immediately available, the new process may be instantiated in the `Block-Suspended` queue until those resources are provided.

The long-term scheduler is executed relatively infrequently. If a 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 may be only invoked 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.

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 environments in which long-running tasks will be scheduled so as to minimize 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 keep as many processes “runnable” as possible.

As we have seen, by reducing the *resident set size* of all processes to be as small as possible (but not too small!), more processes may remain executable.

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 must now 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 page 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 with an average of every 10 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 seconds) making *fine-grained* decisions as to which process to move to the `Running` state next.

The short-term scheduler needs be 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 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 characterized by its success along two dimensions:

- against *user-oriented* criteria under which a single user (selfishly) evaluates their perceived response, or
- against *system-oriented* criteria where the focus is on efficient *global* use of resources such as the processor and memory. A 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 extremely 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, to its commencement of 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. Obviously, 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 demand that the process is either completed with a guaranteed upper-bound on its turnaround time, or provide a guarantee that the process will receive the processor in a guaranteed maximal 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 maximize the number of completed jobs per unit time.

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

Processor utilization – the percentage of time that the processor may be "fed" with work from the `Ready` state. In a single-user, interactive system, processor utilization is very unlikely to exceed a few percent.

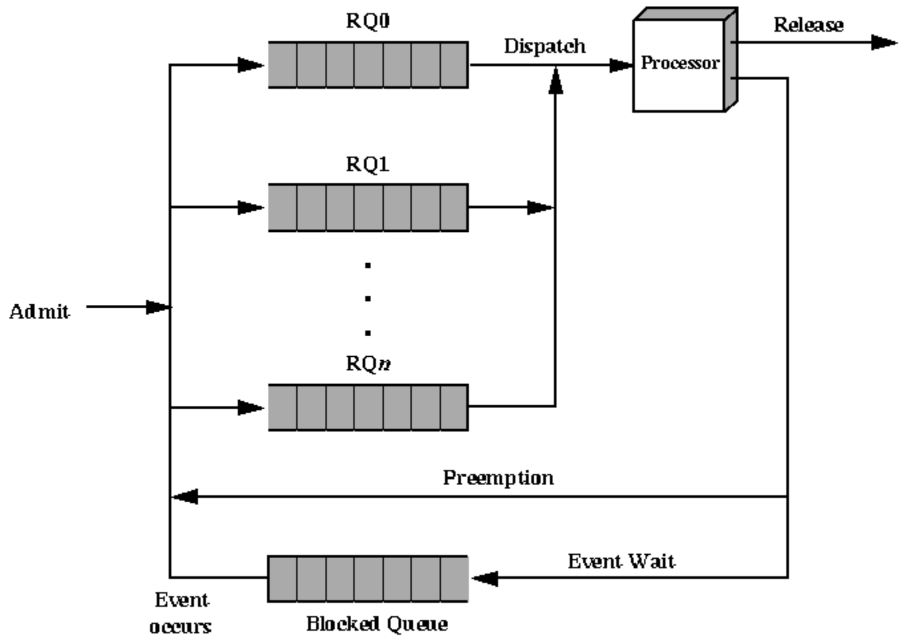
Fairness – in the absence of priority-based requests, all processes should be treated fairly, and none should suffer processor starvation. This simply implies, in most cases, that all processes are moved to the ends of their respective state queues, and may not "jump the queue".

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

Priority-Based Scheduling

In most operating systems, processes are ascribed *priorities* – 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.



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.

Scheduling Selection Functions

When all processes are of the same priority, or all 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-preemptive – once a process is executing, it will continue to execute until:

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

- preemptive – the same three conditions as above apply, and in addition the process may be preempted by the operating system when

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

The Effect of Scheduling Strategies

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