CS 4500 - Operating Systems
Module 6: Process Scheduling Methods

Stanley Wileman

Department of Computer Science
University of Nebraska at Omaha
Omaha, NE 68182-0500, USA

June 4, 2017
In This Module...

- Batch and interactive workloads
- Scheduling basics
- The goals of scheduling
- Selected scheduling algorithms
- Policy and mechanism
Batch processing was so named because multiple jobs were presented as a “batch” to a system for processing.

Today, a related type of job from a batch might be called a “background” job, since it does not require any online or interactive processing.

Batch, or background jobs, don’t perform I/O on terminals or other interactive devices. They also don’t have the timing restraints normally associated with interactive devices.
Interactive processes do utilize on-line devices like terminals, mice, and other devices designed for human input and output (as well as disks and other devices also used by batch jobs).

As such, it is usually important for these processes to be able to respond to user input in a timely manner.

For example, when a user presses a key or moves a mouse (or performs any other recognizable action), it is important that there be swift recognition of that input.
Scheduling Basics

- Scheduling is the name given to all activities of a system associated with determining when various actions are to be performed, or various resources are to be usable by a process.

- The scheduling with which we are concerned in this module is that associated with deciding when a process is allowed to use a processor (CPU).

- Later we will consider scheduling operations on a traditional moving-head disk system.
Execution Sequences are Finite

- Actions of a single process can be considered as “piecewise sequential,” and the sequential pieces to be executed determined by input (or other events).

- If a program is run several times with exactly the same input, it should execute exactly the same sequence of instructions.

- Interactions with other processes may possibly alter the order in which the actions of a process occur, but we can still enumerate all possible sequences of execution without actually executing the process.
Job scheduling has to do with deciding when a job, or a sequence of sequential steps) is to be started.

This is done using information about the resources required by the entire job, and the currently available system resources.

Job scheduling is sometimes called “high-level” scheduling, since a job is a larger, or higher-level unit of work than an individual process.
Recall that multitasking requires that a system’s hardware be capable of performing input/output operations at the same time as the processor was executing a program.

Before multitasking, each job was run separately since it wasn’t possible to have two or more processes executing concurrently.

Each job had a time limit specified for its CPU usage, and this limit was used in making job scheduling decisions.
Process scheduling is sometimes called “low-level” scheduling, since these scheduling decisions are made after a process has been admitted to the system.

The term “scheduler” is usually associated with the system component that does this low-level process scheduling. When a processor becomes available, it selects the appropriate process from among those in the ready state to move to the running state.

You might imagine that there is an additional process state associated with processes that have not yet been admitted to the system; processes in this state cannot compete for system resources.
Some of the usually-quoted goals of scheduling include these:

- **Fairness**: each process should get its “fair share” of the CPU time.
- **Efficiency**: the processor(s) should be kept busy.
- **Response time**: interactive users should receive timely responses to their actions.
- **Turnaround**: batch users should get reasonable response.
- **Throughput**: maximize the number of jobs in a given time period.
Preemptive vs. Non-Preemptive Scheduling

- Preemptive scheduling allows processes that are logically runnable to be suspended (i.e. moved from the running state to the ready state). This is typical of the scheduler in modern versions of Windows, Mac OS X, textscunix, and mainframe operating systems.

- Non-preemptive scheduling (also known as run-to-completion scheduling) allows a process to continue using the processor as long as it wishes. Such scheduling still allows a process to yield the processor or become blocked. This is characteristic of the scheduler in Windows 3.1. In that system, a program that entered an infinite loop could only be terminated by rebooting the system!
To allow preemptive scheduling, virtually all modern systems include devices that cause a periodic interrupt. The names for these devices vary, but the terms “clock” or “timer” are common.

In many systems, timers have programmable interrupt periods but do not keep any record of accumulated time, and clocks have fixed intervals and do keep track of accumulated time.

Each time one of these timing interrupts occurs (perhaps once each millisecond), the system might update the time of day, schedule any actions scheduled for that time, and preempt processes that have had enough processor time in their current execution on the processor.

As an analogy, consider a batter in a baseball game. After three unsuccessful swings at the ball, the batter is “preempted” (that is, they’re out for the current inning). They may, however, get another “at bat” in the next inning.
The Round-Robin Scheduling Algorithm

- Processes in the ready state are ordered by their position in the ready queue.
- Each time a process is allowed to run (that is, use the processor) it is allowed to use at most a specified amount of time called its quantum.
- If a process blocks before using its entire quantum, the process at the head of the ready queue is moved to the running state.
- When a process awakens, it is removed from the blocked state, and placed at the end of the ready queue. The next time it executes, it is given a new quantum.
- If a process uses all of its quantum without blocking, it is moved to the end of the ready queue.
A Gedankenexperiment, or thought experiment, considers a hypothesis for the purpose of thinking through the result of the hypothesis.

Our hypothesis is that we can arbitrarily change the size of the quantum used in round-robin scheduling.

Imagine our computer system has a knob with a pointer on the front. The knob is labeled “Quantum Size”, and there is a pointer on the knob that can be adjusted to any value between $\epsilon$ and $\infty$.

If the quantum is too short ($\epsilon$), then most of the processor time is spent doing context switching between processes - definitely not a desirable situation.

If the quantum is too long ($\infty$), then process response time is compromised. A process can compute for essentially as long as it wishes. Interactive processes are inappropriately delayed from providing timely responses to the user. Again, this is not desirable.

It should be obvious that a compromise must be made between efficiency and responsiveness.
Priority-based Scheduling

- Each process is assigned a priority, which is usually just an integer value in a given range. Larger values indicate a higher, or more important, priority.
- At any instant, a ready process with the highest priority (or any process with the highest priority) is allowed to run.
- Priority assignment to processes can be static (never to change) or dynamic (the priority is adjusted during the life of the process depending on its behavior).
- Priorities may be altered by the system to cause more efficient use of resources or obtain other goals.
- Processes with the same priority can be (and typically are) run in a round-robin fashion.
Scheduling Variations

- A simplistic scheduling system might maintain a single queue that holds entries for all ready processes.
- Rather than have just a single queue that holds all processes, some systems use multiple queues.
- Each queue holds entries for processes in different categories (e.g. high or low priority processes, compute bound or I/O bound processes).
- Processes in each queue may be given different quantum sizes when they’re allowed to execute.
- Processes may be moved between queues if their characteristics change.
In UNIX systems, processes have associated priorities. When a process has used its entire quantum, it is assumed to be compute bound, and its priority is reduced. When a process blocks because it attempts an I/O operation that cannot be immediately completed, it is assumed to be I/O bound and its priority is increased. There is a limit to the range of priorities a process may receive.
The priority of a process can sometimes be adjusted by the user by prefixing the command that starts it with “nice -p” where $p$ is a numeric priority adjustment. The name was suggested because the user is being “nice” to the other processes and users.

The priority adjustment $p$ is subtracted from the default priority associated with the process.

The superuser (that is, the user with user ID 0) can use negative values of $p$ to increase process priorities.
This is an algorithm that attempts to give higher priorities to processes that block before using their entire quantum.

After becoming unblocked (that is, after the requested I/O has been completed), a process is placed in the ready queue in a position directly related to the fractional part of its quantum that went unused during its last execution.

For example, if a process used 25% of its quantum before blocking during its last execution, then when it next becomes ready it will be placed behind 25% of the processes in the ready queue, and not at the end of the queue, as would be typical.

The basic idea is to make “I/O bound” processes more likely to acquire the processor than “compute bound” processes.
Shortest Job First

- In shortest job first (SJF) scheduling, the first job to be run is the one with the smallest estimated execution time.
- SJF provably yields the lowest average turnaround time for jobs (when all jobs are available at the time the scheduling decisions are made).
- Turnaround time is the difference between the time a job was submitted and the time it completed execution.
Aging with Shortest Job First

- Since SJF depends on knowledge of the estimated CPU time, it isn’t directly usable for interactive processes. But we can adapt it.
- Compute an estimate $T'_e$ of the next run time for a process as a weighted sum of the last actual run time (say $T_l$) and the previous estimated run time (say $T_e$).
- Specifically, $T'_e = aT_l + (1 - a)T_e$, where $a$ is the weighting factor (between 0 and 1).
In guaranteed scheduling, the system provides a guarantee to users (processes) about scheduling characteristics.

For example, the system might guarantee that if there are $n$ processes, each one will receive $1/n$ of the processor time.

To achieve this promise, the system keeps track of how much processor time each process has received, and computes the ratio between this and the time it should have received. The process with the lowest ratio is selected for running.
Real-time systems are those in which processes have time constraints associated with their computations.

For example, when playing digital audio it is important that audio data read (from a network, or a disk, or wherever) is processed in a reasonable amount of time so that a gap or interruption will not appear in the audio output.

Real-time system applications are numerous, including monitoring and control functions in places like nuclear power plants, oil refineries, and hospitals.
The processes in most real-time systems repeatedly execute a loop that looks like this:

```plaintext
do forever {
    wait for an event (perhaps a timer expiration)
    perform the appropriate action (or actions)
}
```

Some systems try to handle all events in one process.

It’s usually easier to use one process for each event.

The term task is normally used to describe the processing performed when an event occurs.
Hard and Soft Real Time Systems

- Hard real time systems run tasks with absolute deadlines. Failing to meet a deadline may result in total system failure or death. Consider what happens if a driverless auto senses a stopped vehicle ahead.
- Soft real time systems allow deadlines to be missed—occasionally—without catastrophic failures. For example, a delay of one hour in getting grade reports generated doesn’t necessarily suggest the end of the world—or does it?
- It’s possible to handle both kinds of tasks in the same system.
Events that trigger actions in real-time systems can be periodic (that is, they occur at regular intervals) or aperiodic (unpredictable).

Each event will require a certain number of resources for its processing, including a certain amount (perhaps unpredictable) of CPU time.
Real Time Scheduling

If we have $m$ periodic events, and event $i$ occurs with period $P_i$ and requires $C_i$ seconds of CPU time, then for the real time system to be schedulable we must have

$$\sum_{i=1}^{i=m} \frac{C_i}{P_i} \leq 1$$

What this says, essentially, is that the total of the fractional parts of the processor time used by each task must be less than or equal to 100%.
Keep in mind that this test of schedulability is a simple, absolute requirement. We obviously cannot use more than 100% of the available processor time.

A good engineering “rule of thumb” would require that the limit on processor utilization - the left-hand-side sum - be considerably less than 100%.
In the rate-monotonic algorithm, each process gets a priority proportional to the occurrence of its triggering event, and the highest priority process is always run, preempting the currently running process if necessary.

The earliest deadline first algorithm first runs the process that is required to complete at the earliest time.

The least laxity algorithm first runs the process that has the greatest time before it must begin running in order to meet its deadline.
There are additional resources to be considered in making scheduling decisions. One of the most important of these is primary memory.

If insufficient memory is available to hold all processes, then some can be moved to secondary storage (disk); later they can be moved back to primary memory.

Although possibly misleading and often misused, the usual term used for such a two-level scheduler is **swapper**. The action it performs is called **swapping**.
The usual scheduler is concerned only with selecting from among those processes in the “in memory” ready state.

A higher-level scheduler makes decisions about moving processes between memory and secondary storage.

Swapping effectively introduces two additional process states associated with blocked and ready processes that are swapped out of memory.
Two Level Scheduling Process States

Running

Blocked

Ready

Blocked & Swapped

In Memory

On Disk

Ready & Swapped
Possible Swapping Criteria

Here are some questions that might be asked before swapping a process to disk, or bringing a swapped process back into memory:

- How long has it been since the process was swapped in or out?
- How much processor time has the process had recently?
- How big is the process? That is, how much memory does it require?
- What is the priority of the process?
A **mechanism** is just the algorithm used by the operating system to carry out a particular action. For example, a system may use a priority-based scheduling algorithm as its scheduling mechanism.

A **policy** is a set of rules that determine the parameters or means of application of the mechanism. For example, if users have the ability to set the priority of their processes, then they can affect their scheduling.

Per Brinch Hansen introduced the concept of separation of policy and mechanism in operating systems in the RC 4000 multiprogramming system.
Another Example

- Suppose a building has magnetic or RFID card readers, remote controlled locks, and connections to a centralized security server (like UNO). These provide the mechanism for a security system. By themselves, though, they do not impose any limitations on the entrance policy.
- The centralized security server, however, likely uses a database of room access rules to see who can enter which rooms, and when. This is the policy.
- Note that the policy for this system can be easily changed without modifying the mechanism.
- Contrast this system with one that uses physical keys as the mechanism.
In the next module we’ll discuss deadlocks.

In particular, we will

- define deadlock and look at some examples,
- examine some deadlock models, and
- study various algorithms for dealing with deadlock.