Module 6
Process Scheduling Methods

Updated: September 25, 2014
In This Module…

- Batch and interactive workloads
- Scheduling basics
- The goals of scheduling
- Selected scheduling algorithms
- Policy and mechanism
Batch Processing

- **Batch processing** was so named because multiple jobs were presented to a system for processing.
- Today, batch processing is more likely called *background processing*, since each job is usually submitted individually, and not as part of a group of jobs.
- Batch, or background jobs usually don’t perform I/O on terminals or other interactive devices. They don’t have the timing restraints normally associated with those devices.
Interactive Processing

- Interactive processes do normally 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, it is important that the terminal quickly report 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.

The scheduling with which we are concerned in this module is that associated with deciding when a process is allowed to use the CPU.
Execution Sequences are Finite

Actions of a single process are conceptually piecewise sequential, with the sequence in which the pieces are executed determined by input. 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 executing the process.
Job Scheduling

- 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.
Non-Multitasking Job Scheduling

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

- Process scheduling is sometimes called “low-level” scheduling, since these scheduling decisions are made after a process has been admitted to the system.

- 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.
Goals of Scheduling

- **Fairness**: each process should get its “fair share” of the CPU time.
- **Efficiency**: the CPU 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). [modern versions of Windows, Mac OS X, UNIX, mainframe OS]

- Non-preemptive scheduling (also known as run-to-completion scheduling) allows a process to continue using the CPU as long as it wishes. Such scheduling still allows a process to yield the CPU or become blocked. [Windows 3.1]
Clocks

To allow preemptive scheduling, virtually all systems include clocks that cause a periodic interrupt. [In many systems, clocks have programmable interrupt periods.]

Each time a clock interrupt occurs (perhaps once each millisecond), the system updates the time of day, schedules any actions scheduled for that time, and preempts processes that have had enough CPU time.
Round-Robin Scheduling

- Processes in the ready state are ordered by their position in the ready queue.
- Each process is allowed to run for at most a specified period 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 from the blocked state, it is placed at the end of the ready queue and 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 Gedanken Experiment

How long should a quantum be?

- If it is too short, most of the system time is spent doing context switches – not good.
- If it is too long, response time is sacrificed. (In the limit, this turns into non-preemptive scheduling.) Again, not good.

A compromise must be made between efficiency and responsiveness.
Priority Scheduling

- Each process is assigned a priority, which is usually just an integer value in a given range. Larger values mean higher priority.
- At any instant, the ready process with the highest priority is allowed to run.
- Priorities can be static (they never change) or dynamic (they’re adjusted during the process’ lifetime depending on process behavior).
- Priorities may be altered by the system to cause more efficient use of resources or obtain other goals.
- Processes with identical priorities can be run in a round-robin fashion.
Scheduling Variations

- Rather than have just a single queue that holds all processes, some systems use multiple queues.
- Each queue holds 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.
Example: UNIX

- In UNIX systems, processes have associated priorities.
- When a process uses all its quantum, it is assumed to be compute bound, and its priority is reduced.
- When a process blocks for an I/O operation it is assumed to be I/O bound and its priority is increased.
- There is an upper limit to the priority a process may ever receive.
The UNIX “nice” Command

- The priority of a process can be adjusted by the user by prefixing the command that starts it with “nice –p ” where p is a numeric priority adjustment.
- The priority adjustment p is subtracted from the default priority associated with the process.
- The superuser can use negative values of p to increase process priorities.
Example: Inverse Remainder of Quantum

- Another system attempts to reward processes that block before using their entire quantum.
- After unblocking, these processes are placed in the ready queue in a position related to the fractional part of their quantum that went unused during their last execution.
- For example, if a process used only 25% of its quantum before blocking, when it next became ready it would be placed behind 25% of the processes in the ready queue, and not at the end (as usual).
**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 the 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).
Guaranteed Scheduling

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 $\frac{1}{n^{th}}$ of the CPU time.

To achieve this promise, the system keeps track of how much CPU 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

- Real-time systems are those in which processes have time constraints.
- For example, when playing digital audio it is important that the data is read (from the web, or disk, or wherever) in time so that a gap will not appear in the audio output.
- There are numerous applications for real-time systems, including systems that perform monitoring functions (e.g. nuclear reactors, oil refineries, and hospitals).
Hard and Soft Real Time Systems

- Hard real time systems have absolute deadlines. Failing to meet a deadline can result in total system failure or death.
- Soft real time systems allow deadlines to be missed without such catastrophic failures. For example, a delay of one hour in getting grade reports printed doesn’t necessarily suggest the end of the world...
**Real Time Events**

- 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.

© 2008 Stanley A. Wileman, Jr.  Operating Systems  Slide 25
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

\[
\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1
\]
Other Real-Time Scheduling Approaches

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.

Finally, the least laxity algorithm first runs the process that has the greatest time before it must begin running in order to meet its deadline.
Two-Level Scheduling

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 of them will need to be moved to secondary storage.
Swapping Processes

- 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.
Possible Swapping Criteria

- How long has it been since the process was swapped in or out?
- How much CPU time has the process had recently?
- How big is the process?
- How high 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.
Another Example

- Suppose a system requires a password to be entered by each user during a logon attempt. This is a security mechanism.
- The formation of passwords, and the frequency with which they must be changed, is a policy decision, independent of the password mechanism.
- There are numerous situations in systems where policies and mechanisms are encountered.
Next…

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.