Scheduling

The art and science of allocating processors and other resources to processes & threads

CS-3013 Operating Systems
Hugh C. Lauer

(Slides include materials from Modern Operating Systems, 3rd ed., by Andrew Tanenbaum and from Operating System Concepts, 7th ed., by Silbershatz, Galvin, & Gagne)
Why Scheduling?

• We know how to switch processors among processes or threads, but …

• How do we decide which to choose next?

• Reading Assignment – §2.4 of Tanenbaum
Bursts of processor usage alternate with periods of I/O wait
- a compute-bound process (a)
- an I/O bound process (b)

Which process/thread should have preferred access to CPU?
Which process/thread should have preferred access to I/O or disk?
Why?
Bursts of CPU usage alternate with periods of I/O wait
- a CPU-bound process (a)
- an I/O bound process (b)

Which process/thread should have preferred access to CPU?
Which process/thread should have preferred access to I/O or disk?
Why?

**Definition:**– *I/O-bound*
- A process or thread that spends

**Definition:**– *compute-bound*
- A process or thread that depends mostly on the processor
- Very little waiting for I/O or external activities
Alternating Sequence of CPU And I/O Bursts

- CPU bound = long CPU bursts & short I/O waits
- I/O bound = short bursts of processing & long I/O waits
Histogram of CPU-burst Times
Implementation of Scheduling

- Scheduler
  - Policy
- Dispatcher
  - Mechanism
• Selects from among the tasks in memory that are ready to execute, and allocates a processor to one of them
• Processor scheduling decisions may take place when a task:
  1. Switches from *running* to *waiting* state
  2. Switches from *running* to *ready* state
  3. Switches from *waiting* to *ready*
  4. Terminates
• Scheduling under 1 and 4 is *non-preemptive*
• Scheduling under 2 and 3 is *preemptive*
Dispatcher

- Dispatcher module gives control of a processor to the task selected by the scheduler:—
  - switching context (registers, etc.)
  - Loading the PSW to switch to user mode and restart the selected program

- Dispatch latency – time it takes for the dispatcher to stop one task and start another one running
  - Non-trivial in some systems
Potential Scheduling Criteria

- **Processor utilization** – keep the processor(s) as busy as possible
- **Throughput** – # of tasks that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular task
- **Waiting time** – amount of time task has been waiting in the ready queue
- **Response time** – amount of time from request submission until first response is produced

...
Considerations in Scheduling Policies

• Issues
  – *Fairness* – don’t starve some tasks in favor of others
  – *Priorities* – most important first
  – *Deadlines* – task (or burst) X must be done by time $t$
  – *Optimization* – e.g. throughput, response time

• Reality — No universal scheduling policy
  – Many models
  – Determine what to optimize (define *metrics*)
  – Select an appropriate one and adjust based on experience
Scheduling – Metrics

- **Simplicity** – easy to implement
- **Job latency** – time from start to completion
- **Interactive latency** – time from action start to expected system response
- **Throughput** – number of jobs completed
- **Utilization** – keep processor and/or subset of I/O devices busy
- **Determinism** – insure that jobs get done before some time or event
- **Fairness** – every job makes progress
Some Task Scheduling Strategies

- First-Come, First-Served (FCFS)
- Round Robin (RR)
- Shortest Job First (SJF)
  - *Variation*: Shortest Completion Time First (SCTF)
- Priority
- Real-Time
Scheduling Policies

First Come, First Served (FCFS)

• Easy to implement
• Non-preemptive
  – I.e., no task is moved from running to ready state in favor of another one
• Minimizes context switch overhead
**Example: FCFS Scheduling**

<table>
<thead>
<tr>
<th>Task</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

- Suppose that tasks arrive in the order: $P_1, P_2, P_3$
- The time line for the schedule is:

```
0  24  27  30
P_1  P_2  P_3
```

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- **Average waiting time:** $(0 + 24 + 27)/3 = 17$
Example: FCFS Scheduling
(continued)

Suppose instead that the tasks arrive in the order $P_2, P_3, P_1$

- The time line for the schedule becomes:

```
0  3  6   30
P2  P3  P1
```

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- Previous case exhibits the convoy effect
  - short tasks stuck behind long task
**FCFS Scheduling (summary)**

- Favors *compute bound* jobs or tasks
- Short tasks penalized
  - I.e., once a longer task gets the CPU, it stays in the way of a bunch of shorter task
- Appearance of random or erratic behavior to users
- Does not help in real situations
Scheduling Policies – Round Robin

• Round Robin (RR)
  – *FCFS with preemption* based on time limits
  – Ready tasks given a *quantum* of time when scheduled
  – Task runs until quantum expires or until it blocks (whichever comes first)
  – Suitable for interactive (timesharing) systems
  – Setting quantum is critical for efficiency
Round Robin (continued)

- Each task gets small unit of CPU time (quantum) — for example, 10-100 milliseconds.
  - After quantum has elapsed, task is preempted and added to end of ready queue.
- If $n$ tasks in ready queue and quantum $= q$, then each task gets $1/n$ of CPU time in chunks of $\leq q$ time units.
  - No task waits more than $(n-1) \times q$ time units.
- Performance
  - $q$ large $\Rightarrow$ equivalent to FCFS
  - $q$ small $\Rightarrow$ may be overwhelmed by context switches
Example of RR with Time Quantum = 20

<table>
<thead>
<tr>
<th>Task</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>53</td>
</tr>
<tr>
<td>$P_2$</td>
<td>17</td>
</tr>
<tr>
<td>$P_3$</td>
<td>68</td>
</tr>
<tr>
<td>$P_4$</td>
<td>24</td>
</tr>
</tbody>
</table>

- The time line is:

<table>
<thead>
<tr>
<th>0</th>
<th>20</th>
<th>37</th>
<th>57</th>
<th>77</th>
<th>97</th>
<th>117</th>
<th>121</th>
<th>134</th>
<th>154</th>
<th>162</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td></td>
<td>$P_3$</td>
<td></td>
<td></td>
<td></td>
<td>$P_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_3$</td>
<td>$P_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_1$</td>
<td>$P_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_3$</td>
<td>$P_4$</td>
<td>$P_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$P_4$</td>
<td>$P_1$</td>
<td>$P_3$</td>
<td>$P_3$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Typically, higher average turnaround than SJF, but better response
Comparison of RR and FCFS

Assume: 10 jobs each take 100 seconds – look at when jobs complete

- **FCFS**
  - Job 1: 100s, job 2: 200s, … job 10:1000s

- **RR**
  - 1 sec quantum
  - Job 1: 991s, job 2: 992s, … job 10:1000s

- RR good for short jobs – worse for long jobs
Application of Round Robin

- Time-sharing systems
- *Fair* sharing of limited resource
  - Each user gets $1/n$ of CPU
- Useful where each user has one process to schedule
  - Very popular in 1970s, 1980s, and 1990s
- Not appropriate for desktop systems!
  - *One* user, many processes and threads with very different characteristics
Shortest-Job-First (SJF) Scheduling

- For each task, identify duration (i.e., length) of its next CPU burst.
- Use these lengths to schedule task with shortest burst

Two schemes:–
- Non-preemptive – once CPU given to the task, it is not preempted until it completes its CPU burst
- Preemptive – if a new task arrives with CPU burst length less than remaining time of current executing task, preempt.
  - This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- ...
Shortest-Job-First Scheduling (continued)

- ... 

- SJF is provably optimal – gives *minimum average waiting time* for a given set of task bursts
  - Moving a short burst ahead of a long one reduces wait time of short task more than it lengthens wait time of long one.
Example of Non-Preemptive SJF

<table>
<thead>
<tr>
<th>Task</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (non-preemptive)

- Average waiting time $= (0 + 6 + 3 + 7)/4 = 4$
Example of Preemptive SJF

<table>
<thead>
<tr>
<th>Task</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1)</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>(P_2)</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>(P_3)</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>(P_4)</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (preemptive)

- Average waiting time = \((9 + 1 + 0 + 2)/4 = 3\)
Determining Length of Next CPU Burst

- **Predict** from previous bursts
  - exponential averaging
- Let
  - $t_n =$ actual length of $n^{th}$ CPU burst
  - $\tau_n =$ predicted length of $n^{th}$ CPU burst
  - $\alpha$ in range $0 \leq \alpha \leq 1$
- Then define
  \[ \tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n. \]
  - i.e., the weighted average of $t_n$ and $\tau_n$
This is called *exponential averaging* because

\[ \tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \ldots + (1 - \alpha)^j \alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0 \]

- \( \alpha = 0 \implies \) history has no effect
- \( \alpha = 1 \implies \) only most recent burst counts
- Typically, \( \alpha = 0.5 \) and \( \tau_0 \) is system average
Predicted Length of the Next CPU Burst

- Notice how predicted burst length lags reality
  - $\alpha$ defines how much it lags!
Applications of SJF Scheduling

- Multiple desktop windows active at once
  - Document editing
  - Background computation (e.g., Photoshop)
  - Print spooling & background printing
  - Sending & fetching e-mail
  - Calendar and appointment tracking

- Desktop word processing (at thread level)
  - Keystroke input
  - Display output
  - Pagination
  - Spell checker
Some Task Scheduling Strategies

- First-Come, First-Served (FCFS)
- Round Robin (RR)
- Shortest Job First (SJF)
  - Variation: Shortest Completion Time First (SCTF)
- Priority
- Real-Time
Priority Scheduling

- A priority number (integer) is associated with each task
- CPU is allocated to the task with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Non-preemptive
Priority Scheduling

• (Usually) preemptive
• Tasks are given priorities and ranked
  – Highest priority runs next
  – May be done with multiple queues – multilevel
• SJF ≡ priority scheduling where priority is next predicted CPU burst time
• Recalculate priority – many algorithms
  – E.g. increase priority of I/O intensive jobs
  – E.g. favor tasks in memory
  – Must still meet system goals – e.g. response time
Priority Scheduling Issue #1

• Problem: Starvation
  – I.e., low priority tasks may never execute

• Solution: Aging
  – As time progresses, increase priority of waiting tasks
Priority Scheduling Issue #2

- **Priority inversion**
  - A has high priority, B has medium priority, C has lowest priority
  - C acquires a resource that A needs to progress
  - A attempts to get resource, fails and busy waits
    - C never runs to release resource!
  - or
  - A attempts to get resource, fails and blocks
    - B (medium priority) enters system & hogs CPU
    - C never runs!

- Priority scheduling can’t be naive
Solution

• Some systems increase the priority of a process/task/job to
  • Match level of resource  
    or
  • Match level of waiting task

• Some variation of this is implemented in almost all real-time operating systems
Priority Scheduling (conclusion)

• Very useful if different kinds of tasks can be identified by level of importance

• Very irritating if used to create different classes of citizens
Multilevel Queue
A variation on Priority Scheduling

• Ready queue is partitioned into separate queues — e.g.,
  – foreground (interactive)
  – background (non-interactive)
• Each queue has its own scheduling algorithm
  – foreground – RR
  – background – FCFS
• Scheduling must be done between the queues
  – Fixed priority scheduling: (i.e., serve all from foreground then from background). Possibility of starvation.
  – Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its tasks; i.e., 80% to foreground in RR
  – 20% to background in FCFS
Multilevel Queue Scheduling

- Highest priority: system processes
- Interactive processes
- Interactive editing processes
- Batch processes
- Student processes

Lowest priority
Multilevel Feedback Queue

• A task can move between the various queues
  – *Aging* can be implemented this way
• Multilevel-feedback-queue scheduler defined by the following parameters:
  – number of queues
  – scheduling algorithms for each queue
  – method used to determine when to upgrade a task
  – method used to determine when to demote a task
  – method used to determine which queue a task will enter when that task needs service
Example of Multilevel Feedback Queue

• Three queues:
  – $Q_0$ – RR with time quantum 8 milliseconds
  – $Q_1$ – RR time quantum 16 milliseconds
  – $Q_2$ – FCFS

• Scheduling
  – New job enters queue $Q_0$ (FCFS). When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue $Q_1$.
  – At $Q_1$ job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue $Q_2$. 
Multilevel Feedback Queues

- quantum = 8
- quantum = 16
- FCFS
Scheduling – Examples

- Unix – multilevel - many policies and many policy changes over time
- Linux – multilevel with 3 major levels
  - Realtime FIFO
  - Realtime round robin
  - Timesharing
- Windows Vista – two-dimensional priority policy
  - Process class priorities
    - Real-time, high, above normal, normal, below normal, idle
  - Thread priorities relative to class priorities.
    - Time-critical, highest, …, idle
Reading Assignments

• Tanenbaum
  – §2.4: Scheduling
  – §11.4, Processes and Threads in Vista
  – (also) §10.3.4, Scheduling in Linux

• Love, Chapter 4, Process Scheduling
  – Esp. pp. 47-50

• Much overlap between the two
  – Tanenbaum tends to be broader overview
  – Love tend to be more practical about Linux
Instructive Example

- O(1) scheduling in Linux kernel
- Supports 140 priority levels
  - Derived from *nice* level and previous bursts
  - No queue searching
  - Next ready task identified in constant time
  - Depends upon hardware instruction to find first bit in bit array.
- See Love, p. 42-43

However, less appropriate for desktop systems in actual practice. Replaced with *Completely Fair Scheduler* in kernel 2.6.23.xx
Questions?
Some Task Scheduling Strategies

- First-Come, First-Served (FCFS)
- Round Robin (RR)
- Shortest Job First (SJF)
  - Variation: Shortest Completion Time First (SCTF)
- Priority
- Real-Time
Real-Time Scheduling

- When you need to meet deadlines in the physical world
  - According to the “real world” clock

- Audio or video player
  - to avoid “jerky” presentation, blips and bleeps, etc.

- Process control – to react to physical processes
  - Power plants, refineries, steel mills, nuclear reactors
  - Aircraft control, autopilots, etc.
  - Automatic braking systems
  - …
Two common methods

- Rate Monotonic Scheduling
- Earliest Deadline First

- Many variations
- Many analytic methods for proving QoS (Quality of Service)
Processor Scheduling for Real-Time Rate Monotonic Scheduling (RMS)

• Assume $m$ periodic processes
  – Process $i$ requires $t_i$ milliseconds of processing time every $p_i$ milliseconds.
  – Equal processing every interval — like clockwork!
Example

- Periodic process $i$ requires the CPU at specified intervals (periods)
- $p_i$ is the duration of the period
- $t_i$ is the processing time
- $d_i$ is the deadline by when the process must be serviced
  - Often same as end of period

![Diagram of periodic processes and deadlines]
Processor Scheduling for Real-Time Rate Monotonic Scheduling (RMS)

- Assume $m$ periodic processes
  - Process $i$ requires $t_i$ milliseconds of processing time every $p_i$ milliseconds.
  - Equal processing every interval — like clockwork!
- Assume
  $$\sum_{i=1}^{m} \frac{t_i}{p_i} \leq 1$$
- Assign priority of process $i$ to be $\frac{1}{p_i}$
  - Statically assigned
- Let priority of non-real-time processes be 0
Rate Monotonic Scheduling (continued)

• Scheduler simply runs highest priority process that is ready
  – May pre-empt other real-time processes
  – Real-time processes become ready in time for each frame or sound interval
  – Non-real-time processes run only when no real-time process needs CPU
Example

- \( p_1 = 50 \text{ msec}; \ t_1 = 20 \text{ msec} \)
- \( p_2 = 100 \text{ msec}; \ t_2 = 35 \text{ msec} \)
- \( \text{Priority}(p_1) > \text{Priority}(p_2) \)
- Total compute load is 75 msec per every 100 msec.
- Both tasks complete within every period
  - 25 msec per 100 msec to spare
Example 2

- \( p_1 = 50 \text{ msec}; \; t_1 = 25 \text{ msec} \)
- \( p_2 = 80 \text{ msec}; \; t_2 = 35 \text{ msec} \)
- Priority(\( p_1 \)) > Priority(\( p_2 \))
- Total compute load is \( \approx \) 94% of CPU.
- Cannot complete both tasks within some periods
  - Even though there is still CPU capacity to spare!
Theorem 1: using these priorities, scheduler can guarantee the needed Quality of Service (QoS), provided that

\[ \sum_{i=1}^{m} \frac{t_i}{p_i} \leq m(2^{\frac{1}{m}} - 1) \]

- Asymptotically approaches \( \ln 2 \) as \( m \to \infty \)

\[ \ln 2 = 0.6931... \]

Theorem 2: If a set of processes can be scheduled by any method of static priorities, then it can be scheduled by Rate Monotonic method.
Example 2 again

\[
\left( \frac{t_1}{p_1} + \frac{t_2}{p_2} \right) = \left( \frac{25}{50} + \frac{35}{80} \right) = 0.9375 > 2\left(2^{\frac{1}{2}} - 1\right) = 0.828
\]

• Note that \( p_1 \) pre-empts \( p_2 \) in second interval, even though \( p_2 \) has the earlier deadline!
More on Rate Monotonic Scheduling

• Rate Monotonic assumes periodic processes

• MPEG-2 playback is *not* a periodic process!

Processor Scheduling for Real-Time Earliest Deadline First (EDF)

- When each process \( i \) become ready, it announces deadline \( D_i \) for its next task.

- Scheduler always assigns processor to process with earliest deadline.
  - May pre-empt other real-time processes
Earliest Deadline First Scheduling (continued)

- No assumption of periodicity
- No assumption of uniform processing times

**Theorem:** If any scheduling policy can satisfy QoS requirement for a sequence of real time tasks, then EDF can also satisfy it.

- *Proof:* If $i$ scheduled before $i+1$, but $D_{i+1} < D_i$, then $i$ and $i+1$ can be interchanged without affecting QoS guarantee to either one.
Earliest Deadline First Scheduling (continued)

• EDF is more complex scheduling algorithm
  • Priorities are dynamically calculated
  • Processes must know deadlines for tasks

• EDF can make higher use of processor than RMS
  • Up to 100%

• There is a large body of knowledge and theorems about EDF analysis
Example 2 (again)

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority.

```
+---+---+---+---+---+---+---+---+---+---+---+---+
| P1 | P2 | P1 | P2 | P1 | P1 | P2 | P2 | P1 | P2 |
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
| 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | 110 | 120 | 130 | 140 | 150 | 160 |
```
Some Task Scheduling Strategies

- First-Come, First-Served (FCFS)
- Round Robin (RR)
- Shortest Job First (SJF)
  - Variation: Shortest Completion Time First (SCTF)
- Priority
- Real-Time

Lots of other Scheduling Strategies for Different purposes
Scheduling – Summary

• General theme – what is the “best way” to run \( n \) tasks on \( k \) resources? (\( k < n \))

• Conflicting Objectives – no one “best way”
  – Latency vs. throughput
  – Speed vs. fairness

• Incomplete knowledge
  – E.g. – does user know how long a job will take

• Real world limitations
  – E.g. context switching takes CPU time
  – Job loads are unpredictable
Scheduling – Summary (continued)

- Bottom line – scheduling is hard!
  - Know the models
  - Adjust based upon system experience
  - Dynamically adjust based on execution patterns
Questions?