CS 571
Operating Systems

CPU Scheduling
CPU Scheduling

- Basic Concepts

- Scheduling Criteria

- Scheduling Algorithms
  - First-Come-First-Served
  - Shortest-Job-First, Shortest-remaining-Time-First
  - Priority Scheduling
  - Round Robin
  - Multi-level Queue
  - Multi-level Feedback Queue
Basic Concepts

- During its lifetime, a process goes through a sequence of CPU and I/O bursts.
  - In a multi-programmed computer system, multiple processes compete for the CPU at a given time, to complete their current CPU bursts.

- The CPU scheduler (a.k.a. short-term scheduler) will select one of the processes in the ready queue for execution.

- The CPU scheduler algorithm may have tremendous effects on the system performance
  - Interactive systems
  - Real-time systems
When to Schedule?

Under the simple process state transition model we discussed in Lecture 2, CPU scheduler can be invoked at five different points:

1. When a process switches from the new state to the ready state.
2. When a process switches from the running state to the waiting state.
3. When a process switches from the running state to the ready state.
4. When a process switches from the waiting state to the ready state.
5. When a process terminates.
Non-preemptive vs. Preemptive Scheduling

- Under *non-preemptive scheduling*, each running process keeps the CPU until it completes or it switches to the waiting (blocked) state.

- Under *preemptive scheduling*, a running process may be also forced to release the CPU even though it is neither completed nor blocked.
  - In time-sharing systems, when the running process reaches the end of its time quantum *(slice)*
  - In general, whenever there is a change in the ready queue.
Non-preemptive vs. preemptive kernels

- Non-preemptive kernels do not allow preemption of a process running in kernel mode
  - Serious drawback for real-time applications

- Preemptive kernels allow preemption even in kernel mode
  - Insert safe preemption points in long-duration system calls
  - Or, use synchronization mechanisms (e.g. mutex locks) to protect the kernel data structures against race conditions
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- We’ll discuss scheduling algorithms in two contexts
  - A preemptive scheduler can interrupt a running job
  - A non-preemptive scheduler waits for running job to block
Several criteria can be used to compare the performance of scheduling algorithms:

- **CPU utilization**: keep the CPU as busy as possible
- **Throughput**: number of processes that complete their execution per time unit
- **Turnaround time**: amount of time to execute a particular process
- **Waiting time**: amount of time a process has been waiting in the ready queue
- **Response time**: amount of time it takes from when a request was submitted until the first response is produced, not the complete output.
- **Meeting the deadlines**: (real-time systems)
Optimization Criteria

- Maximize the CPU utilization
- Maximize the throughput
- Minimize the (average) turnaround time
- Minimize the (average) waiting time
- Minimize the (average) response time

In the examples, we will assume
- average waiting time is the performance measure
- only one CPU burst (in milliseconds) per process
First-Come, First-Served (FCFS) Scheduling

<table>
<thead>
<tr>
<th>Process</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 the processes arrive in the order: \( P_1, P_2, P_3 \)

The Gantt Chart for the schedule:

- Waiting time for \( P_1 = 0; \ P_2 = 24; \ P_3 = 27 \)
- Average waiting time: 17
Suppose that the processes arrive in the order $P_2, P_3, P_1$.
The Gantt chart for the schedule:

<table>
<thead>
<tr>
<th></th>
<th>P_2</th>
<th>P_3</th>
<th>P_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- **Problems:**
  - *Convoy effect* (short processes behind long processes)
  - Non-preemptive -- not suitable for time-sharing systems
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. The CPU is assigned to the process with the smallest (next) CPU burst.

- Two schemes:
  - Non-preemptive
  - Preemptive – Also known as the Shortest-Remaining-Time-First (SRTF).

- Non-preemptive SJF is optimal if all the processes are ready simultaneously– gives minimum average waiting time for a given set of processes.

- SRTF is optimal if the processes may arrive at different times
Example for Non-Preemptive SJF

<table>
<thead>
<tr>
<th>Process</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 for Preemptive SJF (SRTF)

<table>
<thead>
<tr>
<th>Process</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)**

![Timeline Diagram]

- **Average waiting time** = \(\frac{9 + 1 + 0 + 2}{4} = 3\)
Problem with SJF: It is very difficult to know exactly the length of the next CPU burst.

Idea: Based on the observations in the recent past, we can try to predict.

Techniques such as exponential averaging are based on combining the observations in the past and our predictions using different weights.
Priority-Based Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
  - Preemptive
  - Non-preemptive
- Example priority-based scheduling schemes
Priority-Based Scheduling (Cont.)

- Priority Assignment
  - Internal factors: timing constraints, memory requirements, the ratio of average I/O burst to average CPU burst....
  - External factors: Importance of the process, financial considerations, hierarchy among users...

- Problem: Indefinite Blocking (or Starvation) – low priority processes may never execute.

- One solution: Aging – as time progresses increase the priority of the processes that wait in the system for a long time.
Starvation occurs when a job cannot make progress because some other job has the resource it require
  - We’ve seen locks, Monitors, Semaphores, etc.
  - The same thing can happen with the CPU!

Starvation can be a side effect of synchronization
  - Constant supply of readers always blocks out writers
  - Well-written critical sections should ensure bounded waiting

Starvation is usually a side effect of the scheduling algorithm:
  - A high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock
Extension to Multiple CPU & I/O Bursts

- When the process arrives, it will try to execute its first CPU burst
  - It will join the ready queue
  - The priority will be determined according to the underlying scheduling algorithm and considering only that specific (i.e. first) burst

- When it completes its first CPU burst, it will try to perform its first I/O operation (burst)
  - It will join the device queue
  - When that device is available, it will use the device for a time period indicated by the length of the first I/O burst.

- Then, it will re-join the ready queue and try to execute its second CPU burst
  - Its new priority may now change (as defined by its second CPU burst)!
  - ..
Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum). After this time has elapsed, the process is preempted and added to the end of the ready queue.

- Newly-arriving processes (and processes that complete their I/O bursts) are added to the end of the ready queue.

- If there are \( n \) processes in the ready queue and the time quantum is \( q \), then no process waits more than \((n-1)q\) time units.

- Performance
  - \( q \) large \(\Rightarrow\) FCFS
  - \( q \) small \(\Rightarrow\) Processor Sharing (The system appears to the users as though each of the \( n \) processes has its own processor running at the \((1/n)^{th}\) of the speed of the real processor)
Example for Round-Robin

<table>
<thead>
<tr>
<th>Process</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 Gantt chart: (Time Quantum = 20)

- Typically, higher average turnaround time than SJF, but better response time.
Choosing a Time Quantum

- The effect of quantum size on context-switching time must be carefully considered.
- The time quantum must be large with respect to the context-switch time.
- Turnaround time also depends on the size of the time quantum.

<table>
<thead>
<tr>
<th>Process time = 10</th>
<th>Quantum</th>
<th>Context switches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>
Turnaround Time and the Time Quantum

<table>
<thead>
<tr>
<th>process</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
</tr>
</tbody>
</table>

average turnaround time

time quantum

GMU CS 571
Multilevel Queue

- Ready queue is partitioned into separate queues: Example, a queue for foreground (interactive) and another for background (batch) processes; or:

```
highest priority

system processes

interactive processes

interactive editing processes

batch processes

student processes

lowest priority
```
Multilevel Queue Scheduling

- Each queue may have its own scheduling algorithm: Round Robin, FCFS, SJF...

- In addition, (meta-)scheduling must be done between the queues.
  - Fixed priority scheduling (i.e. serve first the queue with highest priority). Problems?
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; for example, 50% of CPU time is used by the highest priority queue, 20% of CPU time to the second queue, and so on.
  - Also, need to specify which queue a process will be put to when it arrives to the system and/or when it starts a new CPU burst.
Multilevel Feedback Queue

- In a multi-level queue-scheduling algorithm, processes are permanently assigned to a queue.

- **Idea:** Allow processes to move among various queues.

- **Examples**
  - If a process in a queue dedicated to interactive processes consumes too much CPU time, it will be moved to a (lower-priority) queue.
  - A process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
Example of Multilevel Feedback Queue

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

$Q_i$ has higher priority than $Q_{i+1}$

Scheduling
- A new job enters the queue $Q_0$. When it gains CPU, the job receives 8 milliseconds. If it does not finish in 8 milliseconds, the job is moved to the queue $Q_1$.
- In queue $Q_1$ the job receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to the queue $Q_2$. 
Multilevel Feedback Queue

- quantum = 8
- quantum = 16
- FCFS
Multilevel Feedback Queue

- Multilevel feedback queue scheduler is defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

- The scheduler can be configured to match the requirements of a specific system.
Unix Scheduler

- The canonical Unix scheduler uses a MLFQ
  - 3-4 classes spanning ~170 priority levels
    - Timesharing: first 60 priorities
    - System: next 40 priorities
    - Real-time: next 60 priorities
    - Interrupt: next 10 (Solaris)

- Priority scheduling across queues, RR within a queue
  - The process with the highest priority always runs
  - Processes with the same priority are scheduled RR

- Processes dynamically change priority
  - Increases over time if process blocks before end of quantum
  - Decreases over time if process uses entire quantum
Thread Scheduling

- When processes have also threads, we have two levels of (pseudo-)parallelism.

- Scheduling in such systems differs depending on whether user-level or kernel-level threads are supported.

- With user-level threads, in addition to the process scheduling performed by O.S., we must have thread scheduling performed by the run-time thread management system of each process.

- With kernel-level threads, the operating system can directly perform thread scheduling.
Thread scheduling with user-level threads

Possible scheduling of user-level threads with a 50 ms time quantum and threads that yield the CPU after 5 ms.
Thread scheduling with kernel-level threads

Possible scheduling of kernel-level threads with a 50 ms time quantum and threads that yield the CPU after 5 ms.
Thread scheduling

- With kernel-level threads, the O.S. can exercise a finer control on the execution sequence of threads belonging to different processes.

- With user-level threads, each process can use its own customized thread scheduling algorithm.

- With kernel-level threads, a thread switch involves more overhead.
Scheduling in SMP Environments

- Partitioned versus Global Scheduling
- Processor Affinity
- Load Balancing
- Trade-offs
Solaris 9 Scheduling

- highest priority
- global priority
- scheduling order
- first
- last
- interrupt threads
- realtime (RT) threads
- system (SYS) threads
- fair share (FSS) threads
- fixed priority (FX) threads
- timeshare (TS) threads
- interactive (IA) threads

GMU CS 571
## Windows XP Scheduling Priorities

<table>
<thead>
<tr>
<th></th>
<th>real-time</th>
<th>high</th>
<th>above normal</th>
<th>normal</th>
<th>below normal</th>
<th>idle priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>time-critical</td>
<td>31</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>highest</td>
<td>26</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>above normal</td>
<td>25</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>normal</td>
<td>24</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>below normal</td>
<td>23</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>lowest</td>
<td>22</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>idle</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
## Linux Scheduling

<table>
<thead>
<tr>
<th>numeric priority</th>
<th>relative priority</th>
<th>time quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>highest</td>
<td>200 ms</td>
</tr>
<tr>
<td>99</td>
<td></td>
<td>real-time tasks</td>
</tr>
<tr>
<td>100</td>
<td>lowest</td>
<td>10 ms</td>
</tr>
<tr>
<td>140</td>
<td></td>
<td>other tasks</td>
</tr>
</tbody>
</table>
Scheduling Summary

- Scheduler (dispatcher) is the module that gets invoked when a context switch needs to happen
- Scheduling algorithm determines which process runs, where processes are placed on queues
- Many potential goals of scheduling algorithms
  - Utilization, throughput, wait time, response time, etc.
- Various algorithms to meet these goals
  - FCFS/FIFO, SJF, Priority, RR
- Can combine algorithms
  - Multiple-level feedback queues
  - Unix example