Lecture 6: Scheduling

Prof. Alan Mislove  (amislove@ccs.neu.edu)
Basic Concepts

• Maximum CPU utilization obtained with multiprogramming

• CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait

• **CPU burst** distribution
Alternating Sequence of CPU and I/O Bursts

- load store
- add store
- read from file

wait for I/O

store increment index
write to file

wait for I/O

- load store
- add store
- read from file

wait for I/O

CPU burst

I/O burst

CPU burst

I/O burst

CPU burst

I/O burst
Histogram of CPU-burst Times
CPU Scheduler

• Selects from among the processes in ready queue, and allocates the CPU to one of them
  • Queue may be ordered in various ways

• CPU scheduling decisions may take place when a process:
  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 nonpreemptive
• All other scheduling is preemptive
  • Consider access to shared data
  • Consider preemption while in kernel mode
  • Consider interrupts occurring during crucial OS activities
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

- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running
Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible

- **Throughput** – # 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 output (for time-sharing environment)
Scheduling Algorithm Optimization Criteria

• Max CPU utilization
• Max throughput
• Min turnaround time
• Min waiting time
• Min response time
Non-preemptive scheduling algorithms
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 is:

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
Suppose that the processes arrive in the order: 
\[ P_2, P_3, P_1 \]

- The Gantt chart for the schedule is:

\[ \begin{array}{c|c|c|c|}
0 & 3 & 6 & 30 \\
\hline
P_2 & P_3 & P_1 & \\
\end{array} \]

- 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

**Convoy effect** - short process behind long process
- Consider one CPU-bound and many I/O-bound processes
Shortest-Job-First (SJF) Scheduling

• Associate with each process the length of its next CPU burst
  • Use these lengths to schedule the process with the shortest time

• SJF is optimal – gives minimum average waiting time for a given set of processes
  • The difficulty is knowing the length of the next CPU request
  • Could ask the user
## Example of 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>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>0.0</td>
<td>3</td>
</tr>
</tbody>
</table>

- SJF scheduling chart

![Scheduling Chart](image)

- Average waiting time $= (3 + 14 + 5 + 0) / 4 = 5.5$
Determining Length of Next CPU Burst

• Can only estimate the length – should be similar to the previous one
  • Then pick process with shortest predicted next CPU burst

• Can be done by using the length of previous CPU bursts, using exponential averaging

1. \( t_n = \) actual length of \( n^{th} \) CPU burst
2. \( \tau_{n+1} = \) predicted value for the next CPU burst
3. \( \alpha, 0 \leq \alpha \leq 1 \)
4. Define:
   \[
   \tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n.
   \]

• Commonly, \( \alpha \) set to \( \frac{1}{2} \)
• Preemptive version called \texttt{shortest-remaining-time-first}
Prediction of the Length of the Next CPU Burst

\[
\begin{array}{cccccccc}
\tau_i & 10 & 8 & 6 & 6 & 5 & 9 & 11 & 12 & \ldots \\
\text{CPU burst } (t_i) & 6 & 4 & 6 & 4 & 13 & 13 & 13 & \ldots \\
\end{array}
\]
Priority 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
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process
Example of Priority Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$P_5$</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

- Priority scheduling Gantt Chart

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_2$</td>
<td>$P_5$</td>
<td>$P_1$</td>
<td>$P_3$</td>
<td>$P_4$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>16</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

- Average waiting time = 8.2 msec
Preemptive scheduling algorithms
Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum \( q \)), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

- If there are \( n \) processes in the ready queue and the time quantum is \( q \), then each process gets \( 1/n \) of the CPU time in chunks of at most \( q \) time units at once. No process waits more than \((n-1)q\) time units.

- Timer interrupts every quantum to schedule next process

- Performance
  - \( q \) large \( \Rightarrow \) FIFO
  - \( q \) small \( \Rightarrow \) \( q \) must be large with respect to context switch, otherwise overhead is too high
## Example of RR with Time Quantum = 4

<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>

- The Gantt chart is:

```
+-----+-----+-----+-----+-----+-----+-----+
| P_1 | P_2 | P_3 | P_1 | P_1 | P_1 | P_1 |
+-----+-----+-----+-----+-----+-----+-----+
|  0  |  4  |  7  | 10  | 14  | 18  | 22  |
| 26  | 30  |     |     |     |     |     |
```

- Typically, higher average turnaround than SJF, but better response
- $q$ should be large compared to context switch time
- $q$ usually 10ms to 100ms, context switch < 10 usec
Time Quantum and Context Switch Time

process time = 10

quantum

context switches

12
0

6
1

1
9
Turnaround Time Varies With The Time Quantum

Rule of Thumb:
80% of CPU bursts should be shorter than $q$
Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - foreground (interactive)
  - background (batch)

- Process permanently in a given queue

- 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 processes; 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

• MQ requires process to be assigned a priori

• A process 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 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
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
  • A new job enters queue $Q_0$ which is served RR 8ms
    • 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 RR and receives 8 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
Thread Scheduling

• Distinction between user-level and kernel-level threads

• When threads supported, threads scheduled, not processes

• Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  • Known as process-contention scope (PCS) since scheduling competition is within the process
  • Typically done via priority set by programmer

• Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available

- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing

- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common

- **Processor affinity** – process has affinity for processor on which it is currently running
  - soft affinity
  - hard affinity
  - Variations including processor sets
NUMA and CPU Scheduling

Note that memory-placement algorithms can also consider affinity
Multicore Processors

• Recent trend to place multiple processor cores on same physical chip

• Faster and consumes less power

• Multiple threads per core also growing
  • Takes advantage of memory stall to make progress on another thread while memory retrieve happens

• Hyperthreading is an example
Multithreaded Multicore System

Diagram showing the execution of threads in a multicore system. The diagram includes time slots labeled as compute cycles (C) and memory stall cycles (M). Two threads are depicted: thread_0 and thread_1, each occupying different parts of the execution timeline.
Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)

- Each guest doing its own scheduling
  - Not knowing it doesn’t own the CPUs
  - Can result in poor response time
  - Can effect time-of-day clocks in guests

- Can undo good scheduling algorithm efforts of guests