CPU Scheduling
Reading

Silberschatz et al: Chapters 5.2, 5.3, 5.4
When to Schedule

- Required on these occasions:
  - When a process exits
  - When a process blocks on I/O or a semaphore (more on this later)
  - When a new process is created
  - When an I/O interrupt occurs
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.
Alternating CPU And I/O Bursts

- CPU - I/O burst cycle:
  - Characterizes process execution
  - Alternates, between CPU and I/O activity.

- CPU times are generally much shorter than I/O times.
Histogram of CPU-burst Times
Behavior of Processes in Execution

- Which do you think is better: Having the scheduler favor I/O-bound processes or CPU bound processes or neutral?

- Necessary to determine as quickly as possible the nature (CPU-bound or I/O-bound) of a process, since usually not known in advance.
CPU Scheduler

- Selects from the Ready processes in memory

- CPU scheduling decisions occur when process:
  1. A process switches from running to waiting state.
  2. A process switches from running to ready state.
  3. A process terminates.
When to Schedule

- **Non-preemptive**
  - Picked process runs until it voluntarily relinquishes CPU
    - Blocks on an event e.g., I/O or waiting on another process
    - Process terminates
When to Schedule

- **Preemptive**
  - Picked process runs for a maximum of some fixed time; or until
    - Picked process voluntarily relinquishes CPU
  - Requires a clock interrupt to occur at the end of the time interval to give control of the CPU back to the scheduler
Preemptive Scheduling

- Consider the case of two processes that share data.

- While a process is updating the data it is preempted e.g.,
  - $X = X + 1$ requires several machine level instructions
    - Load R1 X
    - ADD R1 1
    - Load X R1
  - What if the process is pre-empted after the second instruction?

- The second process now tries to read the data.
Preemptive Scheduling

- What if the OS pre-empts an OS process that is updating the state of process
  - E.g., updating the state from running to wait
- Most OS do not allow some of their OS processes to be pre-empted
- Other processes have to expect that they may be pre-empted - more later;
Scheduling Evaluation Metrics

Many quantitative criteria for evaluating a scheduling algorithm:

- **CPU utilization**: Percentage of time the CPU is not idle
- **Throughput**: Completed processes per time unit
- **Turnaround time**: Submission to completion
- **Waiting time**: Time spent on the ready queue
- **Response time**: Response latency
- **Predictability**: Variance in any of these measures
Scheduler Options

- May use priorities to determine who runs next

- Dynamic vs. Static algorithms
  - Dynamically alter the priority of the tasks while they are in the system (possibly with feedback)
  - Static algorithms typically assign a fixed priority when the job is initially started.
First-Come, First-Served (FCFS) Scheduling

- The process that requests the CPU first is allocated the CPU first
- When a process enters the ready state its process control block (PCB) is linked onto the tail of the ready queue
- The code for FCFS scheduling is simple to write and understand
First-Come, First-Served (FCFS) Scheduling

- We will illustrate the use of FCFS with three processes that are currently in a CPU burst phase.
- Two of the three processes are considered I/O bound since their CPU bursts are small.
First-Come, First-Served (FCFS) Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

- Suppose that the processes arrive in the order: P1, P2, P3. The Gantt Chart for the schedule is:

```
   0   24   27   30
P1
P2
P3
```

- Waiting time for P1 = 0; P2 = 24; P3 = 27
- Average waiting time: \( \frac{(0 + 24 + 27)}{3} = 17 \)
Suppose that the processes arrive in the order $P_2, P_3, P_1$.

The Gantt chart for the schedule is:

<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: $(6 + 0 + 3)/3 = 3$

Much better than previous case

Convey effect short process behind long process
FCFS Scheduling

Order of arrival was P1,P2,P3

- P1 gets the CPU
- P2, P3 are in the ready queue
- The I/O queues are idle
- P1 finishes its current CPU burst and goes for I/O
- P2, P3 quickly finish their CPU bursts
- At this point P1,P2,P3 may be waiting for I/O leaving the CPU idle
FCFS Scheduling

Order of arrival was P1, P2, P3

- P1 gets the CPU first
- P2, P3 are in the ready queue
- The I/O queues are idle
- P1 finishes its current CPU burst and goes for I/O
- P2, P3 quickly finish their CPU bursts
- At this point P1, P2, P3 may be waiting for I/O leaving the CPU idle
FCFS Scheduling

Order of arrival was P2, P3, P1

- P2 gets the CPU first
- P3, P1 are in the ready queue
- P2 finishes quickly as does P3
- P2 and P3 go for I/O while P1 is executing
  - Remember that I/O is slower than CPU
Consider a scenario with one CPU-bound process and many I/O bound processes

- Assume the CPU-bound process gets and holds the CPU
- Meanwhile, all other processes finish their I/O and move into the ready queue to wait for the CPU
  - Leaves the I/O queues idle
- CPU-bound process finishes its CPU burst and moves to an I/O device
- All the I/O-bound processes (short CPU bursts) execute quickly and move back to the I/O queues
- CPU is idle
- The above repeats!
- Are the I/O devices and CPU utilized as much as they could be?

Not used in modern operating systems
Scheduling Algorithms LIFO

- Last-In First-out (LIFO)
  - New processes are placed at head of ready queue
  - Improves response time for newly created processes

- Problem:
  - May lead to starvation - early processes may never get CPU
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.
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>6</td>
</tr>
<tr>
<td>P₂</td>
<td>8</td>
</tr>
<tr>
<td>P₃</td>
<td>7</td>
</tr>
<tr>
<td>P₄</td>
<td>3</td>
</tr>
</tbody>
</table>

- SJF scheduling chart

- Average waiting time = \( \frac{3 + 16 + 9 + 0}{4} = 7 \)
Shortest Job First Prediction

- Approximate next CPU-burst duration
  - Based on the durations of the previous bursts
    - The past can be a good predictor of the future
- No need to remember entire past history
- Use exponential average:
  \[
  \tau_{n+1} = \alpha \cdot t_n + (1- \alpha) \cdot \tau_n
  \]
  where \(0 \leq \alpha \leq 1\)
  \(\alpha\) determines the weight placed on past behavior
Prediction of the Length of the Next CPU Burst

![Graph showing prediction of CPU burst length]

<table>
<thead>
<tr>
<th>CPU burst ($t_i$)</th>
<th>6</th>
<th>4</th>
<th>6</th>
<th>4</th>
<th>13</th>
<th>13</th>
<th>13</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;guess&quot; ($\tau_i$)</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Priority Scheduling

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

- SJF is a priority scheduling where priority is the predicted next CPU burst time

- Problem: Starvation
  - Low priority processes may never execute

- Solution: Aging
  - As time progresses the priority of the process increases
Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum), 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.
Round Robin (RR)

- Performance
  - $q$ is too large $\Rightarrow$ FIFO-like behaviour
  - $q$ is too 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:

```
<table>
<thead>
<tr>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
<th>P_1</th>
<th>P_1</th>
<th>P_1</th>
<th>P_1</th>
<th>P_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>22</td>
<td>26</td>
</tr>
</tbody>
</table>
```

- Typically, higher average turnaround than SJF, but better response
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

<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>
Turnaround Time Varies With The Time Quantum

- Turnaround time also depends on the size of the time quantum

- The average turnaround time of a set of processes does not necessarily improve as the time quantum size increases
Multilevel Queue Scheduling

- Today most schedulers use multiple queues
- Essentially the ready queue is really multiple (separate) queues
- The reason is that processes can be classified into different groups
  - Example: foreground (interactive) vs background (batch) processes
Multilevel Queue Scheduling

- Each queue has its own scheduling algorithm e.g.,
  - RR with time quantum of 5
  - RR with time quantum of 8
  - FIFO
Multilevel Queue

- 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 e.g.,
    - 80% to foreground in RR
    - 20% to background in FCFS
Multilevel Queue Scheduling

There can be many queues
Multilevel Feedback Queue Scheduling

- A process can move between queues
- Separate processes according to the characteristics of the CPU bursts (feedback)
  - If a process uses too much CPU time, it will be moved to a lower-priority queue
  - Leave I/O bound and interactive processes in the higher-priority queues
  - In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue
Example: Multilevel Feedback Queues

- Three queues:
  - $Q_0$ - (round robin) RR with time quantum 8 milliseconds
  - $Q_1$ - RR time quantum 16 milliseconds
  - $Q_2$ - FCFS

- The scheduler first executes all processes in $Q_0$; it then proceeds to queue $Q_1$ followed by queue $Q_2$.

- Processes in a queue are served in the order they enter the queue.

- Processes entering $Q_0$ will preempt a running $Q_1$ or $Q_2$ process.
Example: Multilevel Feedback Queues

- Quantum = 8
- Quantum = 16
- FCFS
Example: Multilevel Feedback Queues

- **Scheduling**
  - A new process is placed on $Q_0$
  - When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds (runs entire time), process is moved to queue $Q_1$.
  - At $Q_1$ job process receives 16 additional milliseconds. If it still does not complete (runs entire time), it is preempted and moved to queue $Q_2$. 
Example: Multilevel Feedback Queues

- Scheduling
  - A process is placed on $Q_0$
  - When it gains CPU, job doesn’t use all the 8 milliseconds because it needs I/O.
  - When I/O is completed process returns to $Q_0$
  - Similar situation for $Q_0$
Example: Multilevel Feedback Queues

- What does the algorithm prioritize?
  - I/O bound processes with CPU bursts 8 milliseconds or less
  - These processes quickly get the CPU, finish its CPU burst and go off to the next I/O burst

- Processes that need more than 8 but less than 24 are also served quickly but with lower priority than shorter processes

- CPU bound processes receive the lowest priority
Lottery Scheduling

- Scheduler gives each thread some lottery tickets

- To select the next process to run...
  - The scheduler randomly selects a lottery number
  - The winning process gets to run

- Example
  - Process A gets 50 tickets
  - Process B gets 15 tickets
  - Process C gets 35 tickets
  - There are 100 tickets outstanding.
Lottery Scheduling

- Scheduler gives each thread some lottery tickets.
- To select the next process to run...
  - The scheduler randomly selects a lottery number
  - The winning process gets to run
- **Example**  
  - Process A gets 50 tickets → 50% of CPU
  - Process B gets 15 tickets → 15% of CPU
  - Process C gets 35 tickets → 35% of CPU
  - There are 100 tickets outstanding.
Summary

- Reviewed several scheduling algorithms