Today's class

• Will cover sections 5.1—5.4

• Terms and keywords
  – Scheduler, dispatcher, preemptive scheduling, non-preemptive, scheduling criteria and metrics,
  – Scheduling policies

• Learning objectives
  – To have an understanding of the various scheduling criteria; what their strengths and weaknesses are
  – To have an understanding of the various scheduling processes; what their strengths and weaknesses are
  – what metric each policy maximizes
The CPU scheduler

- Is the short-term scheduler in Chapter 3
- Chooses (select) what process to execute among the ready ones
- When does scheduling happen? It happens when a process:
  1. Switches into ready state
  2. Switches out of running state “on its own”
- Category 1 events cause preemptive scheduling, while category 2 events cause non-preemptive scheduling
The CPU dispatcher

- Performs the *process* and *thread switching*
- The time it takes to dispatch or switch is called *dispatching latency, switching latency, or switching overhead*
- *Dispatch* is usually the *first time* a process is put for execution on the CPU.
Gantt Chart

- It is a visual depiction of a schedule
- “x axis” is time while the “y axis” is slotted into the tasks or the processors. E.g.
Time quantum and context switch time

- Shorter time quantum
  - Better intermix of processes (can overlap IO)
  - Better verisimilitude of multiprogramming
  - More context switch overhead
- Longer time quantum
  - Less context switch overhead
  - More “choppy” or “bursty” operation (less multiprogramming verisimilitude)
Scheduling criteria and polices
Overview of 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)
First-come, first-serve

- Processes are selected in the order that they arrived in the system
- Does FCFS optimize a scheduling criterion?
Shortest job first

- Schedule jobs in order of the shortest execution length
- Advantage:
  - Minimizes average waiting time
- Requires knowledge of the execution length in the future
  - How is that done?
Determining Length of Next CPU Burst

- Can only estimate the length
- 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: \( \tau_{n=1} = \alpha t_n + (1 - \alpha)\tau_n \).

- If we expand the formula, we get:
  
  \[
  \tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots \]
  
  \[
  + (1 - \alpha)\alpha t_{n-j} + \ldots \]
  
  \[
  + (1 - \alpha)^{n+1}\tau_0 \]

- Since both \( \alpha \) and \((1 - \alpha)\) are less than or equal to 1, each successive term has less weight than its predecessor.
Prediction of the Length of the Next CPU Burst

![Graph showing prediction of CPU burst length](image)

<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 (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- 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 increase the priority of the process
Round robin

- 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.
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *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  P_1
0     4     7     10    14    18    22    26    30
```

- Typically, higher average turnaround than SJF, but better *response*
Multi-level queue

- Since there are various types of jobs (foreground, background, high-priority, low-priority etc), it makes sense to have a mixture of scheduling policies.
  - How is that possible?
- Ready queue is partitioned into separate queues: foreground (interactive), background (batch)
- 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
Example multi-level queue

highest priority

- system processes

- interactive processes

- interactive editing processes

- batch processes

- student processes

lowest priority
Multi-level feedback queue scheduling

- 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

- 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 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 $Q_2$. 
How are threads scheduled?

- Two options:
  - Process contention scope
  - System contention scope

- Process contention scope
  - When a process is to be schedule, then internally decide on what threads to be scheduled
  - Threads “compete” with each other

- System contention scope
  - When a scheduling decision is to be made, consider each thread as a separate process
  - Threads compete with other processes
  - Processes with more threads get a better chance of being scheduled