Lecture Topics

- Today: Uniprocessor Scheduling
  (Stallings, chapter 9.1-9.3)
- Next: Advanced Scheduling
  (Stallings, chapter 10.1-10.4)

Announcements

- Self-Study Exercise #10
- Project #8 (due 11/15)
- Project #9 (due 11/29)
Processor Scheduling

- Operating system responsible for allocating resources among competing processes
- OS must schedule processes to use processor cycles
- Scheduling must balance multiple objectives, including effective utilization of the CPU and fairness to all processes

Uniprocessor Scheduling

- Goal: allocate CPU cycles to processes in a way that meets system objectives, such as fast response time, high throughput, and high processor efficiency
- In other words, provide efficient service to all users
- Objectives often conflict, so there must be trade-offs to achieve a balance
### Scheduling Levels

- **Long term**: select processes that are allowed to enter the system (primarily batch systems)
- **Medium term**: schedule processes based on the resources they require (primarily RAM); memory manager handles suspending and resuming processes in most systems
- **Short term**: select process which is allowed to use the CPU next

### Short-Term Scheduler

- Selects one process from the Ready pool
- Crucial that the decision be made quickly
- Good data structure for Ready pool will help
- The point in time when the short-term scheduler executes depends on whether the algorithm is non-preemptive or preemptive
Non-preemptive vs. Preemptive

- OS must have control of CPU to make scheduling decision

- Non-preemptive algorithm:
  - Running ==> Exit
  - Running ==> Blocked

- Preemptive algorithm: whenever OS has control (any interrupt)

---

Process Switch

- Process currently in Running state
  - save context (PC, other registers)
  - update PCB (state, accounting info)
  - move PCB to appropriate pool

- Process selected by OS
  - remove PCB from Ready pool
  - update PCB (state, accounting info)
  - restore context (PC, registers)
First-come-first-served Scheduling

- Process which has been in the Ready queue for the longest time is selected
- Non-preemptive strategy
- Low overhead (minimizes process switches)
- Not practical unless based on priority system (separate FIFO queues for each priority level)

First-come-first-served Scheduling

- Throughput may be low (CPU-bound processes will hold CPU for long periods)
- Response time and turnaround time may be high (short processes and I/O-bound processes have to wait for long periods of time while CPU-bound processes hold the CPU)
- Starvation not possible (as long as processes complete)
Round-robin Scheduling

- Timer interrupt gives OS control of CPU, selects next process from Ready queue (time slicing)
- Preemptive strategy (current process interrupted and must give up CPU)
- Low overhead (but more than FCFS)
- Effective for general-purpose interactive systems

Round-robin Scheduling

- Throughput may be low (CPU-bound processes still favored)
- Response time good for short processes
- Starvation not possible
- Some overhead due to frequent timer interrupts and process switches
Round-robin Time Quantum

- Long quantum:
  - good CPU utilization (few process switches)
  - response time deteriorates

- Short quantum:
  - lower CPU utilization (more process switches)
  - response time improves

- Ideal quantum: somewhat longer than time required for typical interaction

Assume process switch takes 5 ms

- If quantum is 20 ms, response time is good, but 20% of CPU cycles used for process switching

- If quantum is 500 ms, 1% overhead, but response time is poor (with 10 user processes, response time would be 5 seconds)
Summary: FCFS and RR

- FCFS: simple and low overhead, not practical without some priority strategy
- FCFS favors CPU-bound processes (hold CPU for long time periods)
- RR: relatively simple and relatively low overhead, used in many systems
- RR favors CPU-bound processes (placed back in Ready queue sooner)

Feedback Scheduling

- Give preference to short processes by penalizing processes which have been executing for a long time (lower their priority)
- Preemptive strategy
- Moderate to high overhead
- Effective for general-purpose interactive systems
Feedback Scheduling

- Process which gives up the CPU voluntarily remains at the same level
- Process which uses up its time quantum is moved to a lower level
- Older, longer processes drift to lower levels
- Starvation is possible (processes in lower levels may not get access to CPU)
Feedback Scheduling: Variations

- Use a different quantum for each level (longer quantum in lower queues)
- Allow a process to remain at the same level for several preemptions before moving it to a lower level
- Move a process to a higher level if it has not been scheduled for some period of time (prevents starvation)

Priority-based Scheduling

Many systems use priority-based scheduling
- Each process assigned a priority
- Scheduler selects process with highest priority (FCFS within a category)
- Starvation possible for processes with low priority (more complex variations prevent starvation)
Batch Systems

Scheduling policies can be different in a batch environment, since more information is known (user is required to specify resources needed). For example:

- Total CPU time
- Total memory
- Other resources (disk drives, etc)
SPN: Shortest Process Next Scheduling

- Process which has the shortest expected processing time is selected
- Non-preemptive strategy (current process must voluntarily give up the CPU)
- Moderate overhead (although there is more overhead when extended to an interactive environment)

SPN: Shortest Process Next Scheduling

- Service for short processes is good
- Variability of service for long processes increases (and thus less predictable)
- Starvation possible (long processes may never get CPU if there is a steady supply of short processes)
- Can be extended to interactive processes (estimate next CPU burst based on past)
### Example: SPN

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Turnaround time: waiting time + service time

#### Normalized turnaround time: ratio of turnaround time to service time

<table>
<thead>
<tr>
<th>Process</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival Time</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Service Time ($T_s$)</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>SPN</td>
<td>Mean</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finish Time</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td>20</td>
<td>11</td>
</tr>
<tr>
<td>Turnaround Time ($T_r$)</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>$T_r/T_s$</td>
<td>1.00</td>
<td>1.17</td>
<td>2.75</td>
<td>2.80</td>
<td>1.50</td>
</tr>
</tbody>
</table>
SRT: Shortest Remaining Time

- Process which has the shortest remaining time is selected
- Preemptive strategy (OS checks whenever a process enters the Ready pool)
- Moderate overhead

SRT: Shortest Remaining Time

- Service for short processes is good
- Variability of service for long processes increases (and thus less predictable)
- Starvation possible (long processes may never get CPU if there is a steady supply of short processes)
- Can be extended to interactive processes (estimate next CPU burst based on past)
Example: SRT

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

Example: SPN and SRT

<table>
<thead>
<tr>
<th>Process</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival Time</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Service Time ($T_s$)</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

- **SPN**
  - Finish Time: 3, 9, 15, 20, 11
  - Turnaround Time ($T_r$): 3, 7, 11, 14, 3
  - $T_r/T_s$: 1.00, 1.17, 2.75, 2.80, 1.50

- **SRT**
  - Finish Time: 3, 15, 8, 20, 10
  - Turnaround Time ($T_r$): 3, 13, 4, 14, 2
  - $T_r/T_s$: 1.00, 2.17, 1.00, 2.80, 1.00
Traditional Unix Scheduling

- Targeted at interactive environment:
  - good response time for interactive users
  - avoid starvation for low-priority jobs

- Variation on multi-level feedback scheduling
  - Round-robin within each priority queue
  - Priority based on process type and execution history

Processes divided into 5 groups ("bands"):  
- Swapper (highest priority)  
- Block I/O device control  
- File manipulation  
- Character I/O device control  
- User processes (lowest priority)

Priority of processes within a group vary dynamically, but processes cannot migrate from one group to another
Traditional Unix Scheduling

- Purpose of groups ("bands"):  
  - swapper (medium-term scheduler) has highest priority to control number of processes in RAM  
  - fast response to system calls  
  - efficient use of I/O devices  
- User processes: favors I/O bound processes over CPU bound processes (appropriate for interactive environment)

Traditional Unix Scheduling

- Process in Running state preempted after one second of CPU time  
- Timer issues interrupt 60 times per second, interrupt service routine increments counter of process in Running state  
- Priority level of all processes recomputed once per second
Traditional Unix Scheduling

- Base priority assigned by system (places in group)
- User can adjust priority with "nice" value

\[
CPU_j(i) = \frac{CPU_j(i-1)}{2}
\]

\[
P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j
\]

\(CPU_j(i)\) = measure of processor utilization by process \(j\) through interval \(i\)

\(P_j(i)\) = priority of process \(j\) at beginning of interval \(i\); lower values equal higher priorities

\(Base_j\) = base priority of process \(j\)

\(nice_j\) = user-controllable adjustment factor

Example: three CPU bound processes

All three are in the Ready state with the following order: A B C

All three have a "nice" value of 0 (default)

At time 0:

Process A selected: highest priority, oldest
At time 1:

\[ A: \quad 30 = 60/2 \]
\[ 75 = 60 + 30/2 + 0 \]

\[ B: \quad 0 = 0/2 \]
\[ 60 = 60 + 0/2 + 0 \]

\[ C: \quad 0 = 0/2 \]
\[ 60 = 60 + 0/2 + 0 \]

Process B selected:

highest priority, oldest

At time 2:

\[ A: \quad 15 = 30/2 \]
\[ 67 = 60 + 15/2 + 0 \]

\[ B: \quad 30 = 60/2 \]
\[ 75 = 60 + 30/2 + 0 \]

\[ C: \quad 0 = 0/2 \]
\[ 60 = 60 + 0/2 + 0 \]

Process C selected:

highest priority
At time 3:

A: \( 7 = \frac{15}{2} \)
\[ 63 = 60 + \frac{7}{2} + 0 \]

B: \( 15 = \frac{30}{2} \)
\[ 67 = 60 + \frac{15}{2} + 0 \]

C: \( 30 = \frac{60}{2} \)
\[ 75 = 60 + \frac{30}{2} + 0 \]

Process A selected:

### highest priority

Note: count starts at 7

At time 4:

A: \( 33 = \frac{67}{2} \)
\[ 76 = 60 + \frac{33}{2} + 0 \]

B: \( 7 = \frac{15}{2} \)
\[ 63 = 60 + \frac{7}{2} + 0 \]

C: \( 15 = \frac{30}{2} \)
\[ 67 = 60 + \frac{15}{2} + 0 \]

Process B selected:

### highest priority

Note: count starts at 7
At time 5:

A: \[ 16 = \frac{33}{2} \]
\[ 68 = 60 + \frac{16}{2} + 0 \]

B: \[ 33 = \frac{67}{2} \]
\[ 76 = 60 + \frac{33}{2} + 0 \]

C: \[ 7 = \frac{15}{2} \]
\[ 63 = 60 + \frac{7}{2} + 0 \]

Process C selected:
highest priority

Note: count starts at 7

What about shorter CPU bursts?

- Consider only two processes (A and B):
  - A has bursts of 10, 12 and 8
  - B has bursts of 35 and 5

- B will need the CPU two times just to complete the first burst

- A's CPU count will be lower because of short bursts, so it will be favored
Assume A has bursts of 10, 12 and 8, B has bursts of 35 and 5

A: \[60\ 0\] B: \[60\ 0\]

A selected, burst ends with CPU count 10

A: \[62\ 5\] B: \[60\ 0\]

B selected, burst ends with CPU count 60

A: \[61\ 2\] B: \[75\ 30\]

A selected, burst ends with CPU count 14

A: \[63\ 7\] B: \[67\ 15\]

A selected, burst ends with CPU count 15

Limitations: Traditional Unix Scheduling

- Did not provide support for real-time processing
- Did not provide support for multiple processors ( multicore chips did not exist)
- Later versions of Unix support both