Process and Thread Scheduling

Raju Pandey
Department of Computer Sciences
University of California, Davis
Winter 2005
Scheduling

- Context switching
  - an interrupt occurs (device completion, timer interrupt)
  - a thread causes a trap or exception
  - may need to choose a different thread/process to run
- We glossed over the choice of which process or thread is chosen to be run next
  - “some thread from the ready queue”
- This decision is called scheduling
  - scheduling is a policy
  - context switching is a mechanism
Objectives

After this lecture, you should understand:

- the goals of scheduling.
- preemptive vs. non-preemptive scheduling.
- the role of priorities in scheduling.
- scheduling criteria.
- common scheduling algorithms.
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
I/O bound: Many short cpu bursts
CPU bound: few very long cpu bursts
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 Objectives

• Different objectives depending on system
  ▪ Maximize throughput
  ▪ Maximize number of interactive processes receiving acceptable response times
  ▪ Minimize resource utilization
  ▪ Avoid indefinite postponement
  ▪ Enforce priorities
  ▪ Minimize overhead
  ▪ Ensure predictability

• Several goals common to most schedulers
  ▪ Fairness
  ▪ Predictability
  ▪ Scalability
Scheduling Objectives: Fairness

- No single, compelling definition of “fair”
  - How to measure fairness?
    - Equal CPU consumption? (over what time scale?)
  - Fair per-user? per-process? per-thread?
  - What if one process is CPU bound and one is I/O bound?

- Sometimes the goal is to be unfair:
  - Explicitly favor some particular class of requests (priority system), but...
  - avoid starvation (be sure everyone gets at least some service)
Preemptive vs. Nonpreemptive Scheduling

- Preemptive processes
  - Can be removed from their current processor
  - Can lead to improved response times
  - Important for interactive environments
  - Preempted processes remain in memory

- Nonpreemptive processes
  - Run until completion or until they yield control of a processor
  - Unimportant processes can block important ones indefinitely
Priorities

- **Static priorities**
  - Priority assigned to a process does not change
  - Easy to implement
  - Low overhead
  - Not responsive to changes in environment

- **Dynamic priorities**
  - Responsive to change
  - Promote smooth interactivity
  - Incur more overhead than static priorities
    - Justified by increased responsiveness
Multiple levels of scheduling decisions

- **Long term**
  - Should a new “job” be “initiated,” or should it be held?
    - typical of batch systems
    - what might cause you to make a “hold” decision?
- **Medium term**
  - Should a running program be temporarily marked as non-
    runnable (e.g., swapped out)?
- **Short term**
  - Which thread should be given the CPU next? For how long?
  - Which I/O operation should be sent to the disk next?
  - On a multiprocessor:
    - should we attempt to coordinate the running of threads from
      the same address space in some way?
    - should we worry about cache state (processor affinity)?
Levels of scheduling
Queuing Diagram for Scheduling

- Batch jobs
- Interactive users
- Long-term scheduling
- Ready Queue
- Short-term scheduling
- Processor
- Time-out
- Release
- Ready, Suspend Queue
- Blocked, Suspend Queue
- Blocked Queue
- Event Occurs
- Event Wait
- Medium-term scheduling
Scheduling levels

- **Long term scheduling:**
  - Determines which programs are admitted to the system for processing
  - Controls the degree of multiprogramming
  - More processes, smaller percentage of time each process is executed

- **Midterm scheduling:**
  - Part of the swapping function
  - Based on the need to manage the degree of multiprogramming

- **Short term scheduling:**
  - Known as the dispatcher
  - Executes most frequently
  - Invoked when an event occurs
    - Clock interrupts
    - I/O interrupts
    - Operating system calls
    - Signals
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
Organization of Schedulers

- **Embedded**
  - Called as function at end of kernel call
  - Runs as part of calling process

- **Autonomous**
  - Separate process
  - May have dedicated CPU on a multiprocessor
  - On single-processor, run at every quantum: scheduler and other processes alternate
Framework for Scheduling

- **When** is scheduler invoked?
  - **Decision mode**
    - **Preemptive**: scheduler called periodically (quantum-oriented) or when system state changes
    - **Nonpreemptive**: scheduler called when process terminates or blocks

- **How** does it select highest priority process?
  - **Priority function**:
    \[ P = \text{Priority}(p) \]
  - **Arbitration rule**: break ties
    - Random
    - Chronological (First In First Out = FIFO)
    - Cyclic (Round Robin = RR)
Priority function

- Different ways to determine priority
- Possible attributes of processes used to define priority:
  - Attained service time \( (a) \): amount of CPU time allocated
  - Real time in system \( (r) \): attained time + waiting time
  - Total service time \( (t) \): total time between arrival and departure
  - Periodicity \( (d) \): repetition of a computation
  - Deadline (explicit or implied by periodicity): Point in real-time by which process must be completed
  - External priority \( (e) \)
  - Memory requirements (mostly for batch)
  - System load (not process-specific)
First-Come, First-Served (FCFS) Scheduling
First In First Out

<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:

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

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Priority function = $r$ ($r$ = arrival time)
- Decision mode: non-preemptive
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
- *Convoy effect* short process behind long process
Shortest-Job-First (SJR) Scheduling

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

- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
    - Priority Function = -total service time
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)
    - Priority function = -(t-a), t = total service time; a = total attained service time

- SJF is optimal – gives minimum average waiting time for a given set of processes
### Example of 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)**

![Process Scheduling Diagram]

- Average waiting time = \( (0 + 6 + 3 + 7)/4 - 4 \)
Example of 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 (preemptive)

- Average waiting time = $(9 + 1 + 0 + 2)/4 - 3$
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. \( t_n = \text{actual length of } n^{th} \text{ CPU burst} \)
2. \( \tau_{n+1} = \text{predicted value for the next CPU burst} \)
3. \( \alpha, 0 \leq \alpha \leq 1 \)
4. Define: \( \tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n. \)
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 \(\equiv\) highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem \(\equiv\) Starvation – low priority processes may never execute.
- Solution \(\equiv\) Aging – as time progresses increase the priority of the process.
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.

- Performance
  
  - $q$ large $\Rightarrow$ FIFO
  
  - $q$ small $\Rightarrow q$ must be large with respect to context switch, otherwise overhead is too high

- Priority function
  
  - All processes have same priority
Example of RR with Time Quantum = 20

<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 is:

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_1$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_1$</th>
<th>$P_3$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>20</td>
<td>37</td>
<td>57</td>
<td>77</td>
<td>97</td>
<td>117</td>
<td>121</td>
<td>134</td>
</tr>
</tbody>
</table>

- Typically, higher average turnaround than SJF, but better response
Time Quantum and Context Switch Time

The diagram illustrates the process time and quantum for context switches. The process time is 10 units. The quantum is divided into segments, with context switches occurring at specific intervals:

- Starting from 0 to 6: Quantum length is 12, and there are 0 context switches.
- Starting from 6 to 10: Quantum length is 6, and there is 1 context switch.
- Starting from 6 to 10: Another quantum length is 1, and there are 9 context switches.
Multilevel Queue

- 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
Multilevel Queue Scheduling

highest priority
- system processes
- interactive processes
- interactive editing processes
- batch processes
- student processes

lowest priority
Multilevel Feedback Queue

- 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:
  - $Q_0$ – time quantum 8 milliseconds
  - $Q_1$ – 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 $Q_2$. 

Multilevel Feedback Queues

quantum = 8

quantum = 16

FCFS
Scheduling Algorithms: Real-time systems

- **Real-time systems**: Periodic in nature so that computations are repeated at fixed intervals.
- Typically: a process has a period of $d$, it is activated every $d$ seconds, and its computation (total service time) must be completed before start of the next period.

- **Rate Monotonic (RM)**:
  - $d$: periodicity
  - Preemptive
  - Highest priority: shortest period: $P = -d$
• **Earliest Deadline First (EDF):**
  - Intended for periodic (real-time) processes
  - Preemptive
  - $r =$ time since the process first entered the system
  - $d =$ Periodicity
  - Highest priority: shortest time to next deadline
    - $r \div d =$ number of completed periods
    - $r \% d =$ time in current period
    - $d - r \% d =$ time remaining in current period
    - $P = -(d - r \% d)$
## Scheduling algorithms

<table>
<thead>
<tr>
<th>Name</th>
<th>Decision Mode</th>
<th>Priority</th>
<th>Arbitration</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO</td>
<td>Nonpreemptive</td>
<td>( P = r )</td>
<td>random</td>
</tr>
<tr>
<td>Shortest Job First (SJF)</td>
<td>Nonpreemptive</td>
<td>( P = -t )</td>
<td></td>
</tr>
<tr>
<td>Shortest Remaining Time (SRT)</td>
<td>Preemptive</td>
<td>( P = -(t-a) )</td>
<td>Chronological</td>
</tr>
<tr>
<td>Round Robin (RR)</td>
<td>Preemptive</td>
<td>( P=0 )</td>
<td>Cyclic</td>
</tr>
<tr>
<td>Multi Level Priority (MLF)</td>
<td>Preemptive</td>
<td>( P = e )</td>
<td>cyclic</td>
</tr>
<tr>
<td></td>
<td>Non-preemptive</td>
<td>( P = e )</td>
<td>chronological</td>
</tr>
<tr>
<td>Rate Monotonic</td>
<td>Pre-emptive</td>
<td>(-d)</td>
<td>Chronological</td>
</tr>
<tr>
<td>Earliest Deadline First</td>
<td>Pre-emptive</td>
<td>(-(d-r%d))</td>
<td>Chronological</td>
</tr>
</tbody>
</table>

**Notes:**
- FIFO: First In First Out
- SJF: Shortest Job First
- SRT: Shortest Remaining Time
- RR: Round Robin
- MLF: Multi Level Priority
Comparison of Methods

- FIFO, SJF, SRT: Primarily for batch systems
  - FIFO simplest, SJF & SRT have better average turnaround times: \((r_1+r_2+...+r_n)/n\)

- Time-sharing systems
  - Response time is critical
  - RR or MLF with RR within each queue are suitable
  - Choice of quantum determines overhead
    - When \(q \to \infty\), RR approaches FIFO
    - When \(q \to 0\), context switch overhead \(\to 100\%\)
    - When \(q >>\) context switch overhead,
      - \(n\) processes run concurrently at \(1/n\) CPU speed
Priority Inversion Problem

- Assume priority order $p_1 > p_2 > p_3$
- (Unrelated) $p_2$ may delay $p_1$ indefinitely.
- Naïve “solution”: Always run CS at priority of highest process that shares the CS.
  Problem: $p_1$ cannot interrupt lower-priority process inside CS -- a different form of priority inversion.
Priority Inversion Problem

- **Solution: Dynamic Priority Inheritance**
  - $p_3$ is in its CS
  - $p_1$ attempts to enter its CS
  - $p_3$ inherits $p_1$’s (higher) priority for the duration of CS

![Figure 5-11](image-url)
Summary

- Scheduling takes place at many levels
- It can make a huge difference in performance
  - this difference increases with the variability in service requirements
- Multiple goals, sometimes conflicting
- There are many “pure” algorithms, most with some drawbacks in practice – FCFS, SPT, RR, Priority
- Real systems use hybrids that exploit observed program behavior