CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2019 Lecture 8
Scheduling

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
FAQ

• **POSIX**: Portable Operating System Interface is a family of IEEE standards. It defines application programming interface (API), command line shells and utility interfaces, compatibility with variants of OSs.
  
  • Processes/threads/IPC/IO.

• **What is a pthread?**

• **How many threads can a processor (core) execute?** “Concurrently”: many. In parallel (hyperthreading): depending on additional hardware.

• **Java threads: user threads or kernel threads?** Most JVMs implement threads with native, OS level threads,

• **Examples:** Self exercise set 2

• **Iclicker: attendance/response/correctness?**
FAQ

• Thread-local storage (TLS) When is it needed?
  Ex: when using a thread pool, each transaction has a thread and a transaction identifier is needed. Syntax: C11, and Java use `thread_local` keyword to declare.

• Implicit threading: thread creating automated : higher level programming

• Unix signals vs interrupts: Signals are a limited form of inter-process communication. Interrupts are often initiated by hardware.

• Hyper-threading: yes still widely used

• Hardware-assisted parallelism, Out-of-order execution: How? How did they evaluate? Advanced computer architecture topic
Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation
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** followed by I/O burst
- CPU burst distribution is of main concern
Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible: **Maximize**
- **Throughput** – # of processes that complete their execution per time unit: **Maximize**
- **Turnaround time** – time to execute a process from submission to completion: **Minimize**
- **Waiting time** – amount of time a process has been waiting in the ready queue: **Minimize**
- **Response time** – time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment): **Minimize**
Terms for a single process

- Command arrives
- Command begins running
- The first output of command appears
- Command finishes executing

- Wait time
- Response time
- Execution time
- Turnaround time
We will now examine several major scheduling approaches

- **Decide** which process in the ready queue is allocated the CPU

- Could be preemptive or nonpreemptive
  - preemptive: remove in middle of execution ("forced")

- **Optimize** measure of interest
  - We will use **Gantt charts** to illustrate schedules
  - Bar chart with start and finish times for processes
• **Nonpreemptive**: Process keeps CPU until it relinquishes it when
  – It terminates
  – It switches to the waiting state
  – Used by initial versions of OSs like Windows 3.x

• **Preemptive** scheduling
  – Pick a process and let it run for a maximum of some fixed time
  – If it is still running at the end of time interval?
    • Suspend it and pick another process to run

• **A clock interrupt** at the end of the time interval to give control back of CPU back to scheduler
Scheduling Algorithms

Algorithms

• First- Come, First-Served (FCFS)
• Shortest-Job-First (SJF)
  – Shortest-remaining-time-first
• Priority Scheduling
• Round Robin (RR) with time quantum
• Multilevel Queue
  – Multilevel Feedback Queue

Comparing Performance

• Average waiting time etc.
First- Come, First-Served (FCFS) Scheduling

• Process requesting CPU first, gets it first
• Managed with a FIFO queue
  – When process enters ready queue
    • PCB is tacked to the tail of the queue
  – When CPU is free
    • It is allocated to process at the head of the queue
• Simple to write and understand
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: $P_1$, $P_2$, $P_3$ but almost the same time.

The Gantt Chart for the schedule is:

- Waiting time for $P_1$ = ; $P_2$ = ; $P_3$ = 
- Average waiting time: $( $ + $ + $ )/ = 
- Throughput: $/$ = per unit time

Henry Gantt, 1910s

Colorado State University
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: $P_1, P_2, P_3$ but almost the same time.

The Gantt Chart for the schedule is:

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Throughput: $3/30 = 0.1$ per unit time
Suppose that the processes arrive in the order: $P_2, P_3, P_1$

- The Gantt chart for the schedule is:

```
0  3  6  30
P2  P3  P1
```

- 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
- But note - Throughput: $3/30 = 0.1$ per unit same
- **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
- Reduction in waiting time for short process *GREATER THAN* Increase in waiting time for long process
- 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
  - Estimate or could ask the user
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
</tr>
</tbody>
</table>

- All arrive at time 0.
- SJF scheduling chart

- Average waiting time for $P_1, P_2, P_3, P_4 = ( + + + ) / =
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>

- All arrive at time 0.
- SJF scheduling chart

- Average waiting time for P₁, P₂, P₃, P₄ = (3 + 16 + 9 + 0) / 4 = 7
Determining Length of Next CPU Burst

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

• Can be done by using the length of previous CPU bursts, using \textit{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: \( \tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n \).

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

Blue points: guess
Black points: actual

$\alpha = 0.5$

Ex:
$0.5 \times 6 + 0.5 \times 10 = 8$

<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>
Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts
- $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.
- If we expand the formula, substituting for $\tau_n$, we get:
  $$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots + (1 - \alpha)^{n+1}\tau_0$$
- Since both $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Widely used for predicting stock-market etc
Now we add the concepts of varying arrival times and preemption to the analysis.

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

(Will preempt because 4 < 7)
(Will not preempt)

**Preemptive SJF Gantt Chart**

Average waiting time for $P_1, P_2, P_3, P_4$

$$= \frac{[(10-1)+(1-1)+(17-2)+(5-3)]}{4} = \frac{26}{4} = 6.5 \text{ msec}$$
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

MIT had a low priority job waiting from 1967 to 1973 on IBM 7094! 😊
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 (highest)</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>

- Arrived at time 0 in order $P_1, P_2, P_3, P_4, P_5$ (which does not matter)
- Priority scheduling Gantt Chart

- Average waiting time for $P_1, .. P_5$: $(6+0+16+18+1)/5 = 8.2$ msec
• Each process gets a small unit of CPU time (time quantum \( q \)), usually 1-10 milliseconds. After this, the process is preempted, 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 (overhead typically in 0.5\% range)
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>

- Arrive a time 0 in order $P_1$, $P_2$, $P_3$: The Gantt chart is:

```
    P1   P2   P3   P1   P1   P1   P1   P1
0  4  7  10  14  18  22  26  30
```

- Waiting times: $P_1$: $10 - 4 = 6$, $P_2$: 4, $P_3$: 7, average $17/3 = 5.66$ units
- 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 $< 10$ µsec

**Response time**: Arrival to beginning of execution
**Turnaround time**: Arrival to finish of execution
Much smaller quantum compared to burst: many switches
Turnaround Time Varies With The Time Quantum

Rule of thumb: 80% of CPU bursts should be shorter than $q$

Illustration

$q=7$:
Turnaround times for $P_1, P_2, P_3, P_4$:
$6, 9, 10, 17$ av = 10.5
Similarly for $q = 1, ..6$
Multilevel Queue

• Ready queue is partitioned into separate queues, e.g.:
  – foreground (interactive)
  – background (batch)
• Process permanently in a given queue
• Each queue has its own scheduling algorithm, e.g.:
  – 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. Or
  – 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:**
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS (no time quantum limit)

- **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 $Q_2$

CPU-bound: priority falls, quantum raised,
I/O-bound: priority rises, quantum lowered
Thread Scheduling

- Thread scheduling is similar
- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes

Scheduling competition
- 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

LWP layer between kernel threads and user threads in some older OSs
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
- **Assume Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – individual processors can be dedicated to specific tasks at design time
- **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 because of info in cache
  - **soft affinity**: try but no guarantee
  - **hard affinity** can specify processor sets
Note that memory-placement algorithms can also consider affinity
**Non-uniform memory access (NUMA)**, in which a CPU has faster access to some parts of main memory.
Multiple-Processor Scheduling – Load Balancing

• If SMP, need to keep all CPUs loaded for efficiency

• **Load balancing** attempts to keep workload evenly distributed
  – **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  – **Pull migration** – idle processors pulls waiting task from busy processor
  – Combination of push/pull may be used.
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core now common
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
  - See next
This is temporal multithreading. Simultaneous multithreading allows threads to compute in parallel.
Real-Time CPU Scheduling

• Can present obvious challenges
  – **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
  – **Hard real-time systems** – task must be serviced by its deadline

• For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  – But only guarantees soft real-time

• For hard real-time must also provide ability to meet deadlines
  – **periodic** ones require CPU at constant intervals
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 effect time-of-day clocks in guests
- VMM has its own scheduler
- Various approaches have been used
  - Workload aware, Guest OS cooperation, etc.
• Solaris scheduling: 6 classes, Inverse relationship between priorities and time quantum
• Windows XP scheduling: 32 priority levels (real-time, not real-time levels)
• Linux scheduling schemes have continued to evolve.
• Linux Completely fair scheduler (CFS, 2007):
  – Variable time-slice based on number and priority of the tasks in the queue.
  – Maximum execution time based on waiting processes (Q/n).
  – Processes kept in a red-black binary tree with scheduling complexity of O(log N)
  – Process with lowest weighted spent execution (virtual run time) time is picked next. Weighted by priority ("niceness").
Algorithm Evaluation

• How to select CPU-scheduling algorithm for an OS?
• Determine criteria, then evaluate algorithms
• **Deterministic modeling**
  – Type of analytic evaluation
  – Takes a particular predetermined workload and defines the performance of each algorithm for that workload

• Consider 5 processes arriving at time 0:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>10</td>
</tr>
<tr>
<td>$P_2$</td>
<td>29</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
</tr>
<tr>
<td>$P_5$</td>
<td>12</td>
</tr>
</tbody>
</table>
Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:
  - Non-preemptive SFJ is 13ms:
  - RR is 23ms:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>10</td>
</tr>
<tr>
<td>$P_2$</td>
<td>29</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
</tr>
<tr>
<td>$P_5$</td>
<td>12</td>
</tr>
</tbody>
</table>
Probabilistic Models

• Assume that the arrival of processes, and CPU and I/O bursts are random
  – Repeat deterministic evaluation for many random cases and then average

• Approaches:
  – Analytical: Queuing models
  – Simulation: simulate using realistic assumptions
Queueing Models

• Describes the arrival of processes, and CPU and I/O bursts probabilistically
  – Commonly exponential, and described by mean
  – Computes average throughput, utilization, waiting time, etc

• Computer system described as network of servers, each with queue of waiting processes
  – Knowing arrival rates and service rates
  – Computes utilization, average queue length, average wait time, etc
Little’s Formula

- \( n \) = average queue length
- \( W \) = average waiting time in queue
- \( \lambda \) = average arrival rate into queue
- Little’s law – in steady state, processes leaving queue must equal processes arriving, thus:
  \[ n = \lambda \times W \]
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds
Simulations

• Queueing models limited
• **Simulations** more accurate
  – Programmed model of computer system
  – Clock is a variable
  – Gather statistics indicating algorithm performance
  – Data to drive simulation gathered via
    • Random number generator according to probabilities
    • Distributions defined mathematically or empirically
    • Trace tapes record sequences of real events in real systems
Evaluation of CPU Schedulers by Simulation

- Actual process execution
- Trace tape

- Simulation: FCFS
  - Performance statistics for FCFS

- Simulation: SJF
  - Performance statistics for SJF

- Simulation: RR (q = 14)
  - Performance statistics for RR (q = 14)
Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary
Process Synchronization: Objectives

- Concept of process synchronization.
- The critical-section problem, whose solutions can be used to ensure the consistency of shared data.
- Software and hardware solutions of the critical-section problem.
- Classical process-synchronization problems.
- Tools that are used to solve process synchronization problems.