CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2019 Lecture 9
Scheduling

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
Questions from last time

• Prediction of next burst
  – Based on actual recent duration and predicted value (which is based on past actual values)
  – More recent data points get more weight (based on alpha).
  – Why predict if you know the exact value?
  – Computation time needed for prediction? vs typical burst time

• Does average wait time matters if the throughput is the same?

• Shortest Job First (SJF) vs Preemptive SJF
  – SJF is not preemptive
  – Preemptive SJF (also termed Shortest remaining time first)
  – A new process that will take shorter time will preempt a process with longer remaining time.
  – Thus processes with a shorter remaining time have a higher priority.

• What system is responsible for preventing CPU from overheating?
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**
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:

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

• 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
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

![SJF Scheduling Chart]

- Average waiting time for P₁, P₂, P₃, P₄ = \( \frac{3 + 16 + 9 + 0}{4} = 7 \)
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 *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 *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>
Shortest-remaining-time-first (preemptive SJF)

• 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>P₁</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>P₂</td>
<td>1</td>
<td>4 (will preempt because 4&lt;7)</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>9 (will not preempt)</td>
</tr>
<tr>
<td>P₄</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

• Preemptive SJF Gantt Chart

• Average waiting time for P₁,P₂,P₃,P₄

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

• A priority number (integer) is associated with each process

• The CPU is allocated to the process with the highest priority (smallest integer \( \equiv \) highest priority)
  – Can be Preemptive (evict low priority process)
  – Or Nonpreemptive

• SJF is priority scheduling where priority is the inverse of 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

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</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

![Gantt Chart]

- Average waiting time for $P_1, .. P_5$: $(6+0+16+18+1)/5 = 8.2$ msec
Round Robin (RR) with time quantum

- 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>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

- Arrive a time 0 in order P₁, P₂, P₃: The Gantt chart is:

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

- Waiting times: 10-4 =6, 4, 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. All processes arrive at about the same time. Turnaround time for \( P_1, P_2, P_3, P_4 \): 6, 9, 10, 17 \( \text{av} = 10.5 \)
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

Inventor Corbato won the Touring award!
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 $Q_2$

Upgrading may be based on aging. Periodically processes may be moved to the top level.
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
- Pthread API allows both, but Linux and Mac OSX allows only SCS.

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.
• 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
  - 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.

Memory stalls due to cache miss

```
| C | M | C | M | C | M | C | M |
```

```
| C | M | C | M | C | M | C | M |
```

```
| C | M | C | M | C | M | C | C |
```
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.
Operating System Examples

- 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 ready 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. VRN 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 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 (Q=10) is 23ms:
Probabilitistic 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 for avg Queue Length

- $n =$ 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
- Simulation
  - FCFS
  - SJF
  - RR (q = 14)

Performance statistics for FCFS, SJF, and 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
CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2018
Synchronization

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
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
EW Dijkstra *Go To Statement Considered Harmful*
### Too Much Milk Example

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:35</td>
<td>Leave for store.</td>
<td></td>
</tr>
<tr>
<td>12:45</td>
<td>Buy milk.</td>
<td>Leave for store.</td>
</tr>
<tr>
<td>12:50</td>
<td>Arrive home, put milk away.</td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>12:55</td>
<td></td>
<td>Buy milk.</td>
</tr>
<tr>
<td>1:00</td>
<td>Arrive home, put milk away.</td>
<td>Oh no!</td>
</tr>
</tbody>
</table>
• Processes can execute concurrently
  – May be interrupted at any time, partially completing execution
• Concurrent access to shared data may result in data inconsistency
• Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
• **Illustration**: we wanted to provide a solution to the consumer-producer problem that fills *all* the buffers.
  – have an integer `counter` that keeps track of the number of full buffers.
  – Initially, `counter` is set to 0.
  – It is incremented by the producer after it produces a new buffer
  – decremented by the consumer after it consumes a buffer.

*Will it work without any problems?*
Consumer-producer problem

**Producer**

```java
while (true) {
    /* produce an item*/
    while (counter == BUFFER_SIZE) ;
    /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

**Consumer**

```java
while (true) {
    while (counter == 0);
    /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZ
    counter--;
    /* consume the item in
    next consumed */
}
```

They run “concurrently” (or in parallel), and are subject to context switches at unpredictable times.

*In, out: indices of empty and filled items in the buffer.*
Race Condition

Race Condition

counter++ could be compiled as counter-- could be compiled as

register1 = counter  register2 = counter
register1 = register1 + 1  register2 = register2 - 1
counter = register1  counter = register2

They run concurrently, and are subject to context switches at unpredictable times.

Consider this execution interleaving with “count = 5” initially:

S0: producer execute register1 = counter  {register1 = 5}
S1: producer execute register1 = register1 + 1  {register1 = 6}
S2: consumer execute register2 = counter  {register2 = 5}
S3: consumer execute register2 = register2 - 1  {register2 = 4}
S4: producer execute counter = register1  {counter = 6}
S5: consumer execute counter = register2  {counter = 4}

Overwrites!
Critical Section Problem

We saw race condition between counter ++ and counter –

Solution to the “race condition” problem: critical section

• Consider system of \( n \) processes \( \{p_0, p_1, \ldots, p_{n-1}\} \)
• Each process has critical section segment of code
  – Process may be changing common variables, updating table, writing file, etc
  – When one process in critical section, no other may be in its critical section

• **Critical section problem** is to design protocol to solve this
• Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section

Race condition: when outcome depends on timing/order that is not predictable