Questions from last time

• CPU burst

• Comment: probabilistic approaches in computing optimal algorithms, cache, virtual memory, data centers etc. Based on field/recent data.

• 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).
  – Initial prediction? Prior field data

• 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)
Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible: **Maximize**
  - **Throughput** – # of processes that complete their entire execution per time unit: **Maximize**
  - **Turnaround time** – time to execute a process from submission to completion: **Minimize**
  - **Waiting time** – total 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 (assumption: beginning of execution), **not** final 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:

- 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

Henry Gantt, 1910s

Colorado State University
**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

![](image)

- Average waiting time for \(P_1, P_2, P_3, P_4\) = \((3 + 16 + 9 + 0) / 4 = 7\)
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 \( \frac{1}{2} \) selected based on past data
- Preemptive version called *shortest-remaining-time-first*
Prediction of the Length of the Next CPU Burst

Ex:
\[0.5 \times 6 + 0.5 \times 10 = 8\]

Blue points: guess
Black points: actual
\[\alpha = 0.5\]

<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>$P_1$</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 P1, P2, P3, P4

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

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>10</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>14</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>18</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

• Waiting times for $P_1$,$P_2$,$P_3$: $0+(10-4) = 6$, 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
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 $q = 1, \ldots, 6$

Response time: Arrival to \textit{beginning} of execution
Turnaround time: Arrival to finish of execution
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

Real-time processes may have the highest 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!
• 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.

Variations of the scheme were used in earlier versions of Linux.
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
  – Concurrent
  – Parallel: with hyper-threading hardware
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 Version 2.5: Two multilevel priority (“nice values”) queue sets
  – Linux Completely fair scheduler (CFS, 2007):
Goal: fairness in dividing processor time to tasks

- **Balanced (red-black) tree** to implement a ready queue;
  - Efficient. $O(\log n)$ insert or delete time
  - Queue ordered in terms of “virtual run time”
    - execution time on CPU added to value
  - smallest value picked for using CPU
  - small values: tasks have received less time on CPU
  - I/O bound tasks (shorter CPU bursts) will have smaller values

- Priorities (**niceness**) cause different decays of values: higher priority processes get to run for longer time
  - virtual run time is the weighted run-time

- Maximum execution time based on number of waiting processes ($Q/n$)
  - Fewer processes waiting, they get more time each
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
  - Average waiting time for FCS is 28ms:
  - Non-preemptive SFJ is 13ms:
  - RR (Q= 10) 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 \textit{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

Little’s law – in steady state, processes leaving queue must equal processes arriving.

- \( n = \) average queue length
- \( W = \) average waiting time in queue
- \( \lambda = \) average arrival rate into queue
- 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

Each process takes \( 1/\lambda \) time to move one position. Beginning to end delay = \( n \times (1/\lambda) \)
Simulations

• Queueing models limited,
  – require simplifying assumptions

• **Simulations** more versatile
  – 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
  - CPU 10
  - I/O 213
  - CPU 12
  - I/O 112
  - CPU 2
  - I/O 147
  - CPU 173

- Simulations:
  - FCFS
  - SJF
  - RR (q = 14)

- Performance statistics for FCFS
- Performance statistics for SJF
- Performance statistics for RR (q = 14)
Test by 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
Fall 2019 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.
Process Synchronization

EW Dijkstra *Go To Statement Considered Harmful*
## Too Much Milk Example

<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:35</td>
<td>Leave for store.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12:45</td>
<td>Buy milk.</td>
<td></td>
<td>Leave for store.</td>
</tr>
<tr>
<td>12:50</td>
<td>Arrive home, put milk away.</td>
<td></td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>12:55</td>
<td></td>
<td></td>
<td>Buy milk.</td>
</tr>
<tr>
<td>1:00</td>
<td></td>
<td>Arrive home, put milk away.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Oh no!</td>
<td></td>
</tr>
</tbody>
</table>
Background

• 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_SIZE
    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

counter++ could be compiled as  
\[
\begin{align*}
\text{register1} &= \text{counter} \\
\text{register1} &= \text{register1} + 1 \\
\text{counter} &= \text{register1}
\end{align*}
\]

counter-- could be compiled as  
\[
\begin{align*}
\text{register2} &= \text{counter} \\
\text{register2} &= \text{register2} - 1 \\
\text{counter} &= \text{register2}
\end{align*}
\]

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

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

\[
\begin{align*}
S0: \text{producer execute register1} &= \text{counter} & \{\text{register1} = 5\} \\
S1: \text{producer execute register1} &= \text{register1} + 1 & \{\text{register1} = 6\} \\
S2: \text{consumer execute register2} &= \text{counter} & \{\text{register2} = 5\} \\
S3: \text{consumer execute register2} &= \text{register2} - 1 & \{\text{register2} = 4\} \\
S4: \text{producer execute counter} &= \text{register1} & \{\text{counter} = 6\} \\
S5: \text{consumer execute counter} &= \text{register2} & \{\text{counter} = 4\}
\end{align*}
\]

Overwrites!
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, ... 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**, will follow **critical section** with **exit section**, then **remainder section**

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

- critical-section problem to ensure the consistency of shared data
- Software and hardware solutions of the critical-section problem
  - Peterson’s solution
  - Atomic instructions
  - Mutex locks and semaphores
- Classical process-synchronization problems
  - Bounded buffer, Readers Writers, Dining Philosophers
- Another approach: Monitors
General structure: Critical section

```java
do {
  entry section
  critical section
  exit section
  remainder section
} while (true);
```

- Request permission to enter
- Housekeeping to let other processes to enter
Algorithm for Process $P_i$

do {
    while (turn == j);
    critical section
    turn = j;
    remainder section
} while (true);

But $P_j$ is also trying to enter the critical section, and is also running similar code.
Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed.
   - No assumption concerning relative speed of the $n$ processes.
Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode