Questions from last time

• Scheduling time unit: often millisec (1/1000 of a sec)

• 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

• Sometimes tie-breaking rules may be needed

• Simplified examples
  – In reality, processes keep arriving at random times

• Estimation & probabilistic approaches in computing optimal algorithms, cache, virtual memory, data centers etc. Based on field/recent data.
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>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

- Suppose that the processes arrive in the order: \( P₁, P₂, P₃ \) but almost the same time 0.

The Gantt Chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>0</td>
<td>24</td>
<td>27</td>
</tr>
</tbody>
</table>

- Waiting time for \( P₁ = 0; P₂ = 24; P₃ = 27 \)
  - Average waiting time: \( (0 + 24 + 27)/3 = 17 \)
- Throughput: processes finished per unit time \( 3/30 = 0.1 \) per unit
- Turnaround time for \( P₁, P₂, P₃ = 24, 27, 30 \) thus average = 8.2
- Response time for \( P₁, P₂, P₃ = 0, 24, 27 \) assuming .. Thus the average is ..

Turnaround time – time to execute a process from submission to completion.
Response time – time it takes from when a request was submitted until the first response is produced (assumption: beginning of execution), not final output.
Example: FCFS (from IC Q)

<table>
<thead>
<tr>
<th>Process ID</th>
<th>Arrival Time</th>
<th>Burst time</th>
<th>Begins</th>
<th>Completion time</th>
<th>Turnaround time</th>
<th>Waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2-0=2</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>5-1=4</td>
<td>2-1=1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>10-2=8</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>14</td>
<td>14-3=11</td>
<td>7</td>
</tr>
<tr>
<td>P5</td>
<td>4</td>
<td>6</td>
<td>14</td>
<td>20</td>
<td>20-4=16</td>
<td>10</td>
</tr>
<tr>
<td>Av</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>41/5=8.2</td>
<td>21/5=4.2</td>
</tr>
</tbody>
</table>

Note: Processes arrive when they want to. They have to wait when CPU is busy.
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 $P_1, P_2, P_3, P_4 = (\quad + \quad + \quad + \quad ) / =$

Pause for students to do the computation
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

```
  P_4  P_1  P_3  P_2
  0    3    9   16   24
```

- Average waiting time for $P_1, P_2, P_3, P_4 = \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 \( \frac{1}{2} \)
Prediction of the Length of the Next CPU Burst

Blue points: guess
Black points: actual
\( \alpha = 0.5 \)

Ex:
0.5x6 +0.5x10 = 8
Exponential Averaging: Rationale

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - 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:
  
  $\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots + (1 - \alpha)\alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1} t_0$

- 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.
Shortest-remaining-time-first (preemptive SJF)

- Preemptive version called **shortest-remaining-time-first**

- 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>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>4</td>
<td>$P_2$ preempts $P_1$ (will preempt because 4&lt;7)</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>9</td>
<td>$P_3$ doesn’t $P_2$ (will not preempt)</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

- **Preemptive SJF Gantt Chart**

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$P_1$</td>
</tr>
<tr>
<td>1</td>
<td>$P_2$ preempts $P_1$</td>
</tr>
<tr>
<td>2</td>
<td>$P_3$ doesn’t $P_2$</td>
</tr>
<tr>
<td>3</td>
<td>..</td>
</tr>
<tr>
<td>4</td>
<td>..</td>
</tr>
<tr>
<td>5</td>
<td>RT: $P_1$:7, $P_3$:9, $P_4$:5. Thus ..</td>
</tr>
</tbody>
</table>

- Average waiting time for $P_1,P_2,P_3,P_4$
  
  $$\frac{[\text{(10-1)}+(\text{1-1})+(\text{17-2})+(\text{5-3})]}{4} = \frac{26}{4} = 6.5 \text{ msec}$$

- Preempted process gets into Ready Queue (not FCFS here)
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</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>P₄</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

At t=1, P₁ needs 7, P₂ needs 4. Thus P₂ preempts P₁. Etc.

At t=2, P₂ (running) needs 3, P₁ needs 7, P₃ needs 9. No preemption.

Preempted process gets into Ready Queue (not FCFS here).
FAQ: Formula for obtaining Av waiting time etc?

• Is there a formula I can memorize for calculating average waiting time etc, so that I don’t have to bother with understanding how a scheduling approach works?

• Answer: Easiest thing to remember is how a scheduling approach works. You could write a program to do scheduling..
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! 😊
# Ex Priority Scheduling

Non-preemptive

<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>

- P1, P2, P3, P4, P5 all arrive at time 0.
- Priority scheduling Gantt Chart

![Gantt Chart](image)

- Average waiting time for P1, .. P5: \( \frac{(6+0+16+18+1)}{5} = 8.2 \) msec
Round Robin (RR) with time quantum

- Each process gets a small unit of CPU time (time quantum $q$), usually 10-100 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\textsubscript{1}</td>
<td>24</td>
</tr>
<tr>
<td>P\textsubscript{2}</td>
<td>3</td>
</tr>
<tr>
<td>P\textsubscript{3}</td>
<td>3</td>
</tr>
</tbody>
</table>

- Arrive at time 0 in order P\textsubscript{1}, P\textsubscript{2}, P\textsubscript{3}: The Gantt chart is:

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

- Picked from Ready Queue FCFS. Must track Ready Queue.
  - Preempted process joins Ready Queue.
  - Incoming process joins Ready Queue.
  - Incoming process gets in before Preempted one (tie-breaking)

Ready Queue at 0+: P3-P2
Ready Queue at 4+: P1-P3
Ready Queue at 7+: P2-P1
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 at time 0 in order $P_1, P_2, P_3$: The Gantt chart is:

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

- 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 $10$ms to $100$ms, context switch $< 10$ µsec

Response time: Arrival to beginning of execution
Turnaround time: Arrival to finish of execution
Ex: Round robin with quant q=7.
All processes arrive at about the same time.
Turnaround time for P1,P2,P3,P4:
6,9,10,17  av = 10.5
Similarly for q =1, .6 (try at home)

Rules of thumb:
• 80% of CPU bursts should be shorter than q,
  that will minimize turnaround time.

Response time: Arrival to 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
  - [Details at ARPACI-DUSSEAU](#)

Inventor FJ Corbató 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.

Variations of the scheme were used in earlier versions of Linux.
Goal: fairness in dividing processor time to tasks (Con Kolivas, Anaesthetist)

• Variable time-slice based on number and priority of the tasks in the queue.
  – Maximum execution time based on waiting processes (Q/n).
  – Fewer processes waiting, they get more time each

• 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

• Balanced (red-black) tree to implement a ready queue;
  – Efficient. O(log n) insert or delete time

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

Scheduling schemes have continued to evolve with continuing research. A comparison.
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

RTOS: real-time OS. QNX in automotive, FreeRTOS etc.
Virtualization and Scheduling

- Virtualization software schedules multiple guests OSs onto CPU(s)
- Each guest doing its own scheduling
  - Not knowing it doesn’t own the CPUs
  - Can affect time-of-day clocks in guests
- Virtual Machine Monitor 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, non-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):
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 for av Queue Length

• Little’s law – in steady state, processes leaving queue must equal processes arriving, thus:
  – \( n \) = average queue length
  – \( W \) = average waiting time in queue
  – \( \lambda \) = average arrival rate into queue
  \[ n = \lambda \times W \]
  – Valid for any scheduling algorithm and arrival distribution

• Example: average 7 processes arrive per sec, and 14 processes in queue,
  – then average wait time per process \( W = \frac{n}{\lambda} = \frac{14}{7} = 2 \) sec

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

• Queueing models limited

• **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
- simulation
  - FCFS
  - SJF
  - RR ($q = 14$)

Performance statistics for FCFS
Performance statistics for SJF
Performance statistics for RR ($q = 14$)
Actual 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

Synchronization

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources
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*
<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:30</td>
<td><strong>Look in fridge. Out of milk.</strong></td>
<td></td>
</tr>
<tr>
<td>12:35</td>
<td>Leave for store.</td>
<td><strong>Look in fridge. Out of milk.</strong></td>
</tr>
<tr>
<td>12:40</td>
<td>Arrive at store.</td>
<td>Leave for store</td>
</tr>
<tr>
<td>12:45</td>
<td>Buy milk.</td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>12:50</td>
<td>Arrive home, <strong>put milk away.</strong></td>
<td>Buy milk</td>
</tr>
<tr>
<td>12:55</td>
<td></td>
<td><strong>Arrive home, put milk away.</strong> 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?**