FAQ

• Development project: You need to get your own board/hardware and locate the software needed.
• What if processes have the same priority?
• Time quantum- what? How?
• Thread scheduling similar to process scheduling
• Multilevel Feedback Queue: implementations may vary
  – Q0, Q1, Q2 ..
  – Does CPU go to Q1 only when Q0 is empty?
  – If processes keep coming in, will it ever move to Q1?
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 (long CPU burst):** priority falls, quantum raised,
**I/O-bound:** priority rises, quantum lowered
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 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:
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

- \( n = \text{average queue length} \)
- \( W = \text{average waiting time in queue} \)
- \( \lambda = \text{average arrival rate into queue} \)
- 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:
- CPU 10
- I/O 213
- CPU 12
- I/O 112
- CPU 2
- I/O 147
- CPU 173

Trace tape

Simulation:
- FCFS
  - Performance statistics for FCFS
- SJF
  - Performance statistics for SJF
- 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
Process Synchronization

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>
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?*
The 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
counter-- could be compiled as

register1 = counter
register1 = register1 + 1
counter = register1

register2 = counter
register2 = register2 - 1
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!
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
Introductions
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

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

Request permission to enter

Housekeeping to let processes to enter other
Algorithm for Process $P_i$

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

Process $P_j$ 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