• Shortest remaining time first (Preemptive SJF)
  – Need to track the remaining time for all processes

• Round Robin
  – Need to track the position of the processes in the Ready Queue
  – Also need to track the remaining time needed
  – Illustration on youtube
  – Animation CPU Scheduling Algorithm Visualization

• Time quantum- How to decide?
  – Rule of thumb: 80% of CPU bursts should be shorter than q

Disclaimer: I have not verified the accuracy of the on-line sources.
Round Robin Scheduling

Time 1: P2 arrives, gets in RQ.
Time 2: P2 starts.
    P3 arrives, gets in RQ, P1 gets in RQ. RQ={P1, P3}
Time 3: P2 executing.
    P4 arrives, gets in RQ, RQ={P4, P1, P3}
Time 4: P3 starts.
    P5 arrives, gets in RQ, P2 gets in RQ. RQ={P2, P5, P4, P1}
Time 5: no change
Time 6: P1 starts.
    P6 arrives, gets in RQ, P3 done. RQ={P6, P2, P5, P4}
Time 8: P4 starts
    RQ={P6, P2, P5}
Time 9: P4 done, P5 starts
    RQ={P6, P2}
Time 11: P2 starts.
    RQ={P5, P6}
FAQ

• Thread scheduling similar to process scheduling
  – process-contention scope (PCS)
  – system-contention scope (SCS)

• Non-uniform memory access: Multiple processor system with some memory closer to each processor
**Tues Q1**

i. Pthreads are a POSIX standard API for thread creation and synchronization. **True**

ii. A Pthread library is always implemented in the user space. **False**

**Tues Q2.**

In a thread with deferred cancellation, cancellation only occurs when

**Ans: The thread reaches the Cancellation point**

**Thurs Q3.**

A process burst is 20 milliseconds. How long will it execute at the lowest level? **Ans: 0**
Schedulers

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

• Multilevel Feedback Queue Details at ARPACI-DUSSEAU

• Linux Completely fair scheduler (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).
  – 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 for av 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

- Example: average 7 processes arrive per sec, average wait time is 2 sec, then queue length is \( n = \lambda \times W = 7 \times 2 = 14 \).

Each process takes \( 1/\lambda \) time to move one position. Beginning to end delay \( W = n \times (1/\lambda) \). Hence Little’s law
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
  – [Illustration](#)
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

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

Performance statistics:
- for FCFS
- for SJF
- 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
- However note that
  - Most flexible schedulers can be modified per-site or per-system
  - Or may use APIs to modify priorities
CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
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
Overview

• We synchronization is needed
• Critical section: access controlled to permit just one process
  – How the critical section be implemented
  – Mutex locks and semaphores
• Classic synchronization problems
• Will a solution cause a deadlock?
## Too Much Milk Example

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<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, put milk away.</td>
<td>Buy milk</td>
</tr>
<tr>
<td>12:55</td>
<td>Arrive home, put milk away. 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**

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

**Consumer**

```c
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
register1 = counter
register1 = register1 + 1
counter = register1

counter-- could be compiled as
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 follows.

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
A process is prohibited from entering the critical section while another process is in it. Multiple processes are executing the same code.
A good solution to the critical-section problem should have these attributes

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
Peterson’s Solution

• Good algorithmic description of solving the problem
• Two process solution only
• Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
• The two processes share two variables:
  – int turn;
  – Boolean flag[2]

  – The variable turn indicates whose turn it is to enter the critical section
  – The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready to enter!
Algorithm for Process $P_i$

```c
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j); /*Wait*/
    critical section
    flag[i] = false;
    remainder section
} while (true);
```

- The variable `turn` indicates whose turn it is to enter the critical section.
- The `flag` array is used to indicate if a process is ready to enter the critical section. $flag[i] = true$ implies that process $P_i$ is ready!
- Note: Entry section- Critical section-Exist section
- These algorithms assume 2 or more processes are trying to get in the critical section.

For process $P_i$, $P_j$ runs the same code concurrently.
Provable that the three CS requirement are met:

1. Mutual exclusion is preserved
   \( \mathcal{P}_i \) enters CS only if:
   
   either \( \text{flag}[j] = \text{false} \) or \( \text{turn} = i \)

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met.
   
   A process waits only one turn.

**Detailed proof in the text.**

**Note:** there exists a generalization of Peterson’s solution for more than 2 processes, but bounded waiting is not assured.
Synchronization: Hardware Support

• Many systems provide hardware support for implementing the critical section code.
• All solutions below based on idea of **locking**
  – Protecting critical regions via locks
• Modern machines provide special atomic hardware instructions
  • **Atomic** = non-interruptible
    – test memory word and set value
    – swap contents of two memory words
Solution 1: using \texttt{test\_and\_set}()

- \textbf{Shared Boolean variable lock}, initialized to FALSE
- \textbf{Solution:}
  
  \begin{verbatim}
  do {
      while (test_and_set(&lock)) ; /* do nothing */
      /* critical section */
      ....
      lock = false;
      /* remainder section */
      ...
  } while (true);
  \end{verbatim}

  To break out:
  Return value of \texttt{TestAndSet} should be FALSE

  Lock TRUE: locked,    Lock FALSE: not locked.
  If two \texttt{TestAndSet()} are attempted \textit{simultaneously}, they
  will be executed \textit{sequentially} in some arbitrary order

  \texttt{test\_and\_set}(&lock) returns the lock value and then sets it to True.  

\textit{Colorado State University}
Solution 2: Swap: Hardware implementation

Remember this C code?

```c
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
Using Swap (concurrently executed by both)

do {
    key = TRUE;
    while (key == TRUE) {
        Swap(&lock, &key)
    }

    critical section

    lock = FALSE;

    remainder section
} while (TRUE);

Lock is a SHARED variable. Key is a variable local to the process.

Lock == false when no process is in critical section.

Cannot enter critical section UNLESS lock == FALSE by other process or initially

If two Swap() are executed simultaneously, they will be executed sequentially in some arbitrary order.
Swap()

Process 0
Key = TRUE
Swap ( )
Key ==FALSE, enter
Critical section
Lock = FALSE

Lock
Locked by Process 0
Lock = TRUE

Process 1
Key = TRUE
Swap ( )
Key == TRUE, wait
Critical section
Locked by Process 1
Lock = FALSE

Note: I created this to visualize the mechanism. It is not in the book. - Yashwant
Bounded-waiting Mutual Exclusion with test_and_set

For process $i$:

do {  
    waiting[$i$] = true;
    key = true;
    while (waiting[$i$] && key)
        key = test_and_set(&lock);
    waiting[$i$] = false;
    /* critical section */
    j = ($i + 1) \% n;
    while (($j != \text{i}) && !waiting[$j$])
        j = ($j + 1) \% n;
    if ($j == \text{i}$)
        lock = false;
    else
        waiting[$j$] = false;
    /* remainder section */
} while (true);

Shared Data structures initialized to FALSE
• boolean waiting[$n$]; Pr $n$ wants to enter
• boolean lock;

The entry section for process $i$:
• First process to execute TestAndSet will find key == false; ENTER critical section,
• EVERYONE else must wait

The exit section for process $i$:
Attempts to finding a suitable waiting process $j$ (while loop) and enable it,
or if there is no suitable process, make lock FALSE.
The previous algorithm satisfies the three requirements

- **Mutual Exclusion**: The first process to execute TestAndSet(lock) when lock is false, will set lock to true so no other process can enter the CS.
- **Progress**: When a process i exits the CS, it either sets lock to false, or waiting[i] to false (allowing j to get in), allowing the next process to proceed.
- **Bounded Waiting**: When a process exits the CS, it examines all the other processes in the waiting array in a circular order. Any process waiting for CS will have to wait at most n-1 turns.
Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is `mutex` lock
- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a `spinlock`
acquire() and release()

```
acquire() {
    while (!available)
        ; /* busy wait */
}

release() {
    available = true;
}
```

• Usage
  do {
    acquire lock
    critical section
    release lock
    remainder section
  } while (true);
acquire() and release()
### acquire() and release()

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Lock</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>open</td>
<td>Attempt to acquire lock</td>
</tr>
<tr>
<td>..</td>
<td>closed</td>
<td>Acquires lock</td>
</tr>
<tr>
<td>Attempt to acquire lock</td>
<td>closed</td>
<td>Critical section</td>
</tr>
<tr>
<td>Attempt to acquire lock</td>
<td>closed</td>
<td>Critical section</td>
</tr>
<tr>
<td>Attempt to acquire lock</td>
<td>open</td>
<td>Release lock</td>
</tr>
<tr>
<td>Acquires lock</td>
<td>closed</td>
<td>..</td>
</tr>
<tr>
<td>Critical section</td>
<td>closed</td>
<td>..</td>
</tr>
</tbody>
</table>
How are locks supported by hardware?

• Atomic read-modify-write
• Atomic instruction in x86
  – LOCK instruction prefix, which applies to an instruction does a read-modify-write on memory (INC, XCHG, CMPXCHG etc)
  – Ex: lock cmpxchg <dest>, <source>
• In RISK processors? Instruction-pairs
  – LL (Load Linked Word), SC (Store Conditional Word) instructions in MIPS
  – LDREX, STREX in ARM
  – Creates an atomic sequence