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

• **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
Example: FCFS

<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>2-2=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>4-3=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>8-5=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>11-4=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>16-6=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>
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 average 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, and 14 processes in queue, then average wait time per process \( W = \frac{n}{\lambda} = \frac{14}{7} = 2 \text{ sec} \)

Each process takes \( 1/\lambda \) 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
  - 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)
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
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*
<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>Leave for store</td>
</tr>
<tr>
<td>12:40</td>
<td>Arrive at store.</td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>12:45</td>
<td>Buy milk.</td>
<td>Buy milk</td>
</tr>
<tr>
<td>12:50</td>
<td>Arrive home, put milk away.</td>
<td>Arrive home, put milk away.</td>
</tr>
<tr>
<td>12:55</td>
<td></td>
<td>Arrive home, put milk away. 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?
Producer
while (true) {
  /* produce an item*/
  while (counter == BUFFER_SIZE) ;
  /* do nothing */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}

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

Request permission to enter

Housekeeping to let other processes to enter
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
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 $P_i$

```plaintext
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. $\text{flag}[i] = \text{true}$ implies that process $P_i$ is ready!
Provable that the three CS requirement are met:

1. Mutual exclusion is preserved
   \( P_i \) 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 test_and_set()

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

  To break out: Return value of TestAndSet should be FALSE

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

  test_and_set(&lock) returns the lock value and then sets it to True.
Solution 2: Swap: Hardware implementation

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

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

    critical section

    lock = FALSE;

    remainder section
} while (TRUE);
```

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

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

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

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
  - Busy waiting
  - Swap(): Key == False
  - Critical section
  - Lock = FALSE

Locked by Process 0

Locked by Process 1
For process $i$:

```c
  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 != i) && !waiting[j])
      j = (j + 1) % n;
    if (j == 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()

<table>
<thead>
<tr>
<th>acquire() {</th>
<th>release() {</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (!available)</td>
<td>available = true;</td>
</tr>
<tr>
<td>; /* busy wait */</td>
<td>}</td>
</tr>
</tbody>
</table>

**Usage**

```c
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
Semaphores by Dijkstra

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore $S$ – integer variable
- Can only be accessed via two **indivisible (atomic)** operations
  - `wait()` and `signal()`
    - Originally called `P()` and `V()` based on Dutch words
- Definition of the `wait()` operation
  ```
  wait(S) {
    while (S <= 0) // busy wait
      S--;
  }
  ```
- Definition of the `signal()` operation
  ```
  signal(S) {
    S++;
  }
  ```

Binary semaphore: When $S$ is 0 or 1, it is a mutex lock

Waits until another process makes $S=1$
Wait(S) and Signal (S)

Process 0

Wait(S)

Critical section

Signal (S)

Semaphore S

S =1

S =0

S =1

Locked by Process 1

S =1

Process 1

Wait (S)

Busy waiting

Gets lock, S -

Critical section

Signal (S)
acquire() and release()

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Semaphore S</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical section</td>
<td>0</td>
<td>wait (), busy waiting</td>
</tr>
<tr>
<td>Signal ( ) S++</td>
<td>1</td>
<td>Waiting, finished</td>
</tr>
<tr>
<td>..</td>
<td>0</td>
<td>S- -</td>
</tr>
<tr>
<td>Wait( )</td>
<td>0</td>
<td>Critical section</td>
</tr>
<tr>
<td>Wait( )</td>
<td>0</td>
<td>Critical section</td>
</tr>
<tr>
<td>Waiting finished</td>
<td>1</td>
<td>Signal ( ) S++</td>
</tr>
<tr>
<td>S--</td>
<td>0</td>
<td>..</td>
</tr>
<tr>
<td>Critical section</td>
<td>0</td>
<td>..</td>
</tr>
</tbody>
</table>
Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a *mutex lock*
- Can solve various synchronization problems
- Consider $P_1$ and $P_2$ that require $S_1$ to happen before $S_2$
  
  Create a semaphore “**synch**” initialized to 0

<table>
<thead>
<tr>
<th>P1:</th>
<th>P2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$;</td>
<td>wait(synch);</td>
</tr>
<tr>
<td>signal(synch);</td>
<td>$S_2$;</td>
</tr>
</tbody>
</table>

- Can implement a counting semaphore $S$ as a binary semaphore
The counting semaphore

• Controls access to a finite set of resources
• Initialized to the number of resources
• Usage:
  – Wait (S): to use a resource
  – Signal (S): to release a resource
• When all resources are being used: $S == 0$
  – Block until $S > 0$ to use the resource
Semaphore Implementation

• Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time

• Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  – Could now have **busy waiting** in critical section implementation
    • But implementation code is short
    • Little busy waiting if critical section rarely occupied

• Note that applications may spend lots of time in critical sections and therefore this is not a good solution

• Alternative: block and wakeup (next slide)
Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

```c
typedef struct{
    int value;
    struct process *list;
} semaphore;
```
Implementation with no Busy waiting (Cont.)

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}

If value < 0
abs(value) is the number of waiting processes

typedef struct{
    int value;
    struct process *list;
} semaphore;