FAQ

• Multilevel Feedback Queue: Q0, Q1, Q2 ..
  – Does CPU go to Q1 only when Q0 is empty?
  – If processes keep coming in, will it ever move to Q1?

• “Completely Fair Scheduling” (2007, not in text)
  – Somewhat complex. Maximum execution time based on waiting processes (Q/n). Process with lowest weighted spent execution (virtual run time) time is picked next. Weighted by priority (niceness).

• Virtualization: VMM, guest? Sharing of CPU time among guests and processes.
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></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></td>
<td>Arrive home, put milk away. 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?
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_SIZ
    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*}
\text{S0: producer execute register1} &= \text{counter} \\
\text{S1: producer execute register1} &= \text{register1} + 1 \\
\text{S2: consumer execute register2} &= \text{counter} \\
\text{S3: consumer execute register2} &= \text{register2} - 1 \\
\text{S4: producer execute counter} &= \text{register1} \\
\text{S5: consumer execute counter} &= \text{register2}
\end{align*}

\{\text{register1 = 5}\} \\
\{\text{register1 = 6}\} \\
\{\text{register2 = 5}\} \\
\{\text{register2 = 4}\} \\
\{\text{counter = 6}\} \\
\{\text{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**
Process Synchronization: Outline

- Process synchronization: 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

```c
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 $P_i$

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

- $P_i$ Waits to enter
- $P_i$ Executes critical section
- $P_i$ lets j enter critical section
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
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!
For process $P_i$

\[
\text{do } \{ \\
\quad \text{flag}[i] = \text{true}; \\
\quad \text{turn} = j; \\
\quad \text{while} \ (\text{flag}[j] \land \text{turn} = j); \quad /*\text{Wait}*/ \\
\quad \text{critical section} \\
\quad \text{flag}[i] = \text{false}; \\
\quad \text{remainder section} \\
\} \text{ while } (\text{true});
\]

- The variable $\text{turn}$ indicates whose turn it is to enter the critical section.
- The $\text{flag}$ 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

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 using test_and_set()

- **Shared Boolean variable lock**, initialized to FALSE
- **Solution:**
  ```
  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 FALSE:** not locked.
If two TestAndSet() are executed *simultaneously*, they will be executed *sequentially* in some arbitrary order.
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}

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

    critical section

    lock = FALSE;

    remainder section
} while (TRUE);

Cannot enter critical section UNLESS lock == FALSE

lock is a SHARED variable key is a LOCAL variable

If two Swap() are executed simultaneously, they will be executed sequentially in some arbitrary order
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 != 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];
• 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 $i$:
Part I: Finding a suitable waiting process $j$ and enable it to get through the while loop,
or if thre 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 exits the CS, it either sets lock to false, or waiting[j] 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*

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>
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
  ```c
  wait(S) {
    while (S <= 0) // busy wait
    S--;
  }
  ```
- Definition of the `signal()` operation
  ```c
  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)

Semaphore S

Process 0
- Wait(S)
- Critical section
- Signal (S)
- S = 0
- Locked by Process 1
- S = 1

Process 1
- Wait (S)
- Busy waiting
- Gets lock, S -
- Critical section
- Signal (S)
- S = 1
- Locked by Process 1
- S = 0

Colorado State University
### 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
  
  **P1:**
  
  $S_1$
  
  `signal(synch);`

  **P2:**

  `wait(synch);`

  $S_2$

- 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)
• 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

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

```c
wait(semaphore *S) {
    S->value--;  // If value < 0
    if (S->value < 0) {  // abs(value) is the number of waiting processes
        add this process to S->list;
        block();
    }
}

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

```c
typedef struct{
    int value;
    struct process *list;
} semaphore;
```
Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Let $s$ and $q$ be two semaphores initialized to 1

  $P_0$
  
  ```
  wait(S);
  wait(Q);
  ...
  signal(S);
  signal(Q);
  ```

  $P_1$
  
  ```
  wait(Q);
  wait(S);
  ...
  signal(Q);
  signal(S);
  ```

- **P0 executes wait(s), P1 executes wait(Q)**
  - P0 must wait till P1 executes signal(Q)
  - P1 must wait till P0 executes signal(S)  

  Deadlock!
Priority Inversion

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

- Solved via **priority-inheritance protocol**
  - Process accessing resource needed by higher priority process
    - Inherits higher priority till it finishes resource use
  - Once done, process reverts to lower priority
Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

- Monitors