

Synchronization (I)

---- Race Condition & Mutual exclusion

Example

Shared variable:

```
int count =0;
```


Code for P1:

```
count++;
```


Code for P2:

```
count--;
```

Race Condition (The order of execution of instructions influencing the result produced)

 `count++` is compiled to

```
register1 = count  
register1 = register1 + 1  
count = register1
```

 `count--` is compiled to

```
register2 = count  
register2 = register2 - 1  
count = register2
```

 Consider these executions interleaving (“count = 0” initially):

```
S0: P1 executes register1 = count {register1 = 0}  
S1: P1 executes register1 = register1 + 1 {register1 = 1}  
S2: P2 executes register2 = count {register2 = 0}  
S3: P2 executes register2 = register2 - 1 {register2 = -1}  
S4: P1 executes count = register1 {count = 1}  
S5: P2 executes count = register2 {count = -1}
```

Observations

- ❏ Concurrent access to shared data may result in **data inconsistency** (i.e., the results are different from different execution sequences → un-predictable)
- ❏ Maintaining **data consistency** requires mechanisms to ensure mutual exclusive access to shared data by processes/threads.

Critical-Section (CS)

- ❏ A system consist of n processes (threads) $P_0, P_1, \dots P_{n-1}$.
- ❏ Each process has a segment of code, called **critical section (CS)**, in which the process may be changing common variables, updating a shared table, writing a shared file, and so on.
- ❏ Ideally, **when one process is executing in its CS, no other process is to be allowed to execute in its CS.**

Solution to Critical-Section Problem: Requirements

Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Progress (**no starvation**) - If no process is executing in its critical section and there exists some process P that wishes to enter their critical section, then the selection of P to enter the critical section cannot be postponed indefinitely.

Peterson's Solution

- ❏ Two process (thread) solution
- ❏ Assume that the LOAD and STORE 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 wishes to enter the critical section. $\text{flag}[i] = \text{true}$ implies that process P_i wishes to!

Algorithm for Process P_i

```
flag[i] = TRUE;  
turn = j; //j=1-i is the ID of other process  
while (flag[j] && turn == j);  
..... critical section .....  
flag[i] = FALSE;  
..... remainder section ...
```


Single-wood Bridge



Generalization: Solution to CS Problem Using Locks

acquire lock

critical section

release lock

remainder section



Synchronization Hardware

- ❏ Many systems provide hardware support for critical section code
- ❏ Uniprocessors – could disable interrupts
 - ❏ Currently running code would execute without preemption
 - ❏ Not effective or too inefficient on multiprocessor systems
- ❏ Modern machines provide special atomic hardware instructions
 - ❏ Test memory word and set value ([atomic TestAndSet](#))
 - ❏ Swap contents of two memory words ([atomic Swap](#))

TestAndSet Instruction

 Definition:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Solution using TestAndSet

 Shared boolean variable lock., initialized to false.

 Solution:

```
while ( TestAndSet (&lock ));  
... critical section ...  
lock = FALSE;  
... remainder section ...
```

Swap Instruction

 Definition:

```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```
key = TRUE;
while ( key == TRUE) Swap (&lock, &key );
... critical section ...
lock = FALSE;
... remainder section ...
```

Semaphore

- ❏ An abstract data type
 - ❏ Semaphore S – integer variable
 - ❏ Two standard operations modify S : `wait()` and `signal()`
 - ❏ Originally called `P()` and `V()`
- ❏ Can only be accessed via two (atomic) operations
 - ❏ `wait (S) {`
 - `while (S <= 0); //blocked`
 - `S--;`
 - `}`
 - ❏ `signal (S) {`
 - `S++;`
 - `}`

Usage of Semaphores

- ❏ **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement

 - ❏ Also known as **mutex** (i.e., **mutual exclusive**) locks

 - ❏ Provides mutual exclusion

```
Semaphore mutex; // initialized to 1
```

```
wait (mutex);
```

```
... Critical Section ...
```

```
signal (mutex);
```

```
... remainder section ...
```

- ❏ **Counting** semaphore – integer value can range over an unrestricted domain