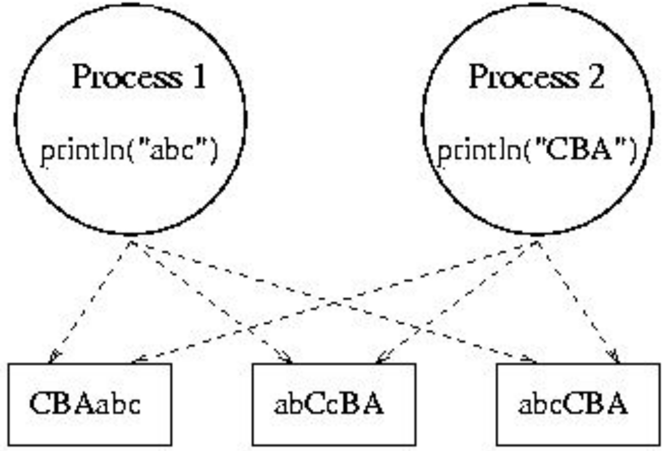
CENG 322
LAB 7

Synchronization

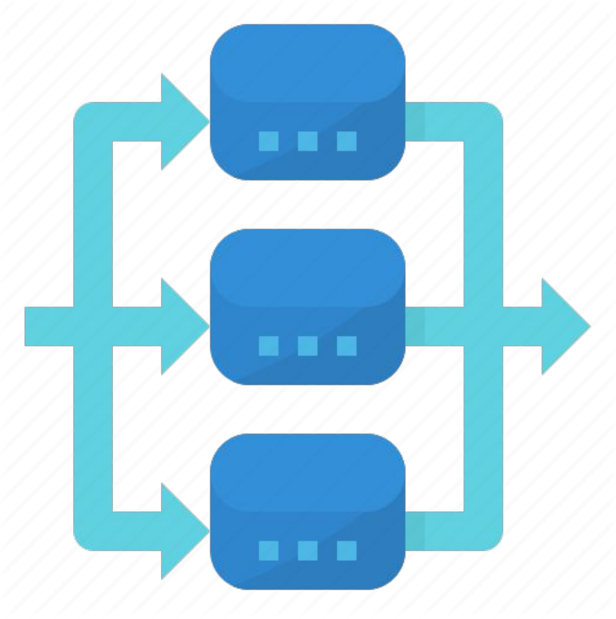
Processes and Synchronization

- Two types of processes in terms of synchronization:
    - **Independent Process :** Execution of one process does not affects the execution of other processes.
    - **Cooperative Process :** Execution of one process affects the execution of other processes.
- Cooperating processes **share resources** and **may cause inconsistency** in processes data therefore **process synchronization** is required for consistency of data.



Processes and Synchronization

Why do we permit processes to **cooperate**?



Processes and Synchronization

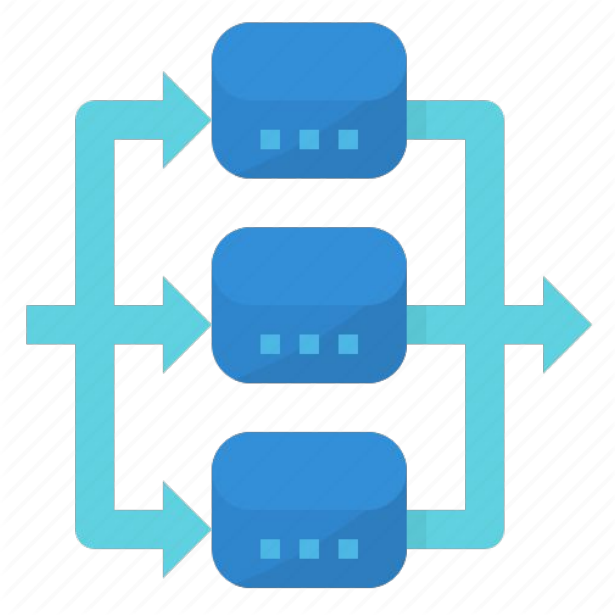
Why do we permit processes to **cooperate**?

Want to **share resources:**

- One computer, many users.
- One database of checking account records, many tellers.

Want to do things **faster:**

- Read next block while processing current one.
- Divide job into sub-jobs, execute in parallel.



Processes and Synchronization

Assume that:

`A = 1`

`B = 3`

Process 1

`A = B + 1`

Process 2

`B = B * 2`

Result: `A = ?` and `B = ?`

Processes and Synchronization

Assume that:

```
A = 1
```

```
B = 3
```

Process 1

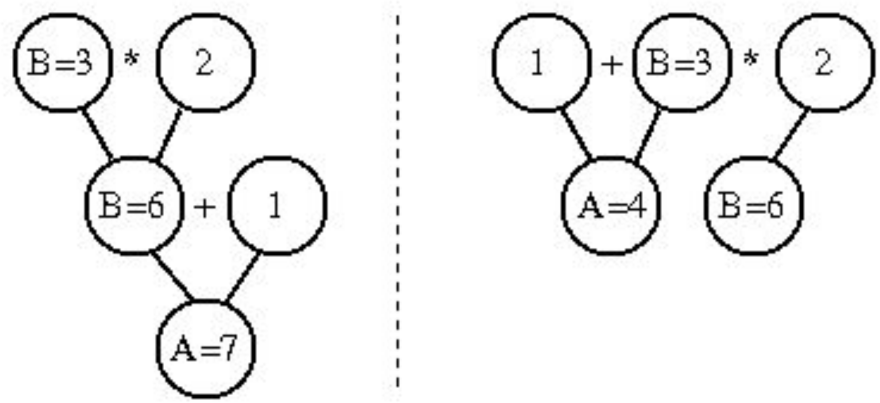
```
A = B + 1
```

Process 2

```
B = B * 2
```

Inconsistent !

Non-deterministic !



Race Condition

Race condition mostly **occurs** when two or more processes (or threads) try to **read**, **write** and possibly **make the decisions** based on the memory that they are accessing concurrently.

- Multiple processes
- Executing same code
- Accessing the same memory
- Using shared variable

Race Condition Example

- **counter++** could be implemented as

register1 = counter

register1 = register1 + 1

counter = register1

- **counter--** could be implemented as

register2 = counter

register2 = register2 - 1

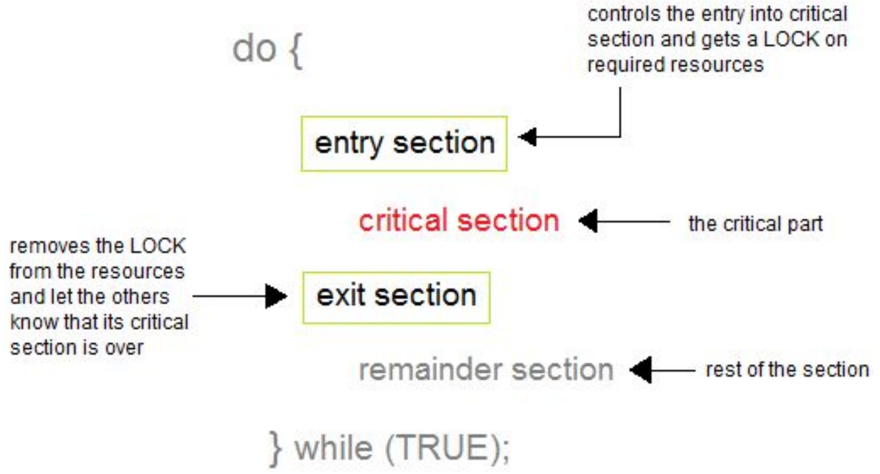
counter = register2

- 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}

Critical Section

A **Critical Section** is a code segment that accesses **shared variables** and has to be executed as an **atomic action**.

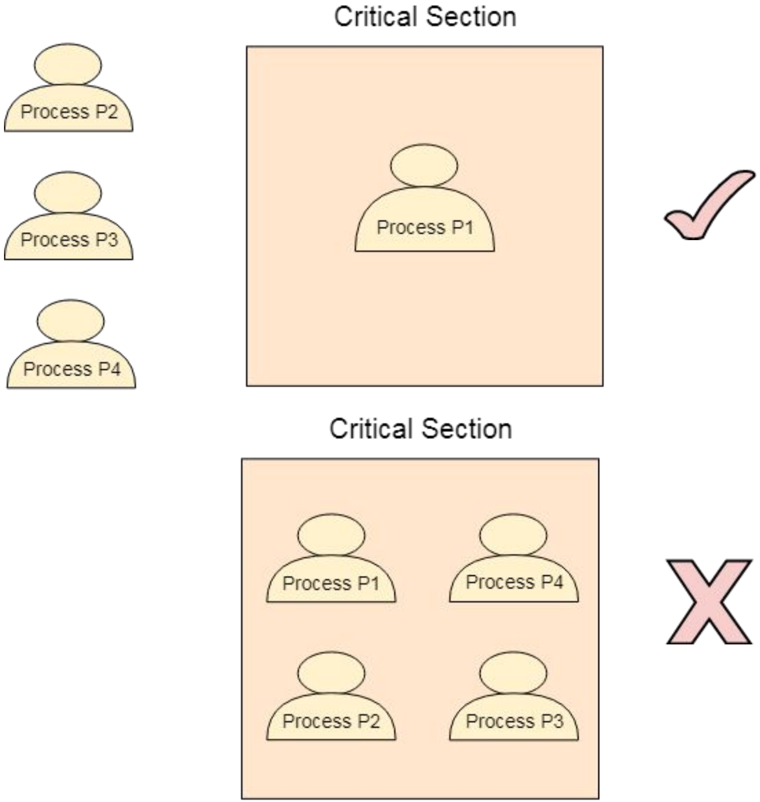


- It may cause **race conditions**.

Solutions of Critical Section Problem

Any solution must satisfy these requirements:

- **Mutual Exclusion:** At most one process in the critical section.
- **Progress:** If process P1 is outside the critical section, then P1 cannot prevent process P2 from entering the critical section.
- **Bounded Buffer:** Every process will get a chance to enter its critical section without waiting indefinitely.



Peterson's Solution in C

Check **synchronization_1.c** !

```
do
{
    Flag[i] = TRUE;
    Turn = j;
    while(flag[j] && turn = = j)
        Critical Section
    Flag[i] = FALSE;
        Remainder section
}while(TRUE);
```

```c
void lock(int self)
{
    // Acquire lock
    flag[self] = 1;
    // Give the other thread the chance
    //to acquire lock
    turn = 1-self;
    // Wait
    while (flag[1-self]==1 && turn==1-self) ;
}
void unlock(int self)
{
    flag[self] = 0;
}
```

Peterson’s Solution in C

Thread 1 - 0

```c
flag[0] = 1;
turn = 1;
// Wait
while (flag[1]==1 && turn==1);
```

Thread 2 - 1

```c
flag[1] = 1;
turn = 0;
// Wait
while (flag[0]==1 && turn==0);
```

Case 1: Thread 1 first

```c
flag[0] = 1;
flag[1] = 0;
```

Case 2: Thread 2 first

```c
flag[0] = 0;
flag[1] = 1;
```

Bakery Algorithm in C

- It is also called **Lamport's bakery** algorithm.
- Solution for **N threads (or processes)**.
- Assign numbers to the processes then support **FCFS**.

Assign:

```
(ticket number, process(or thread) id)
```

Waiting condition:

```
while (tickets[other] != 0 &&
(tickets[other]<tickets[thread]
||(tickets[other]==tickets[thread]
&& other < thread)))
```

Check **synchronization_2.c** !

Mutual Exclusion (Mutex) in C

- A mutex lock is an object that **only one thread** at a given time can **obtain it** while the others must **wait** until the lock **owner thread** release it.
- We can implement mutex with **pthread** library:
- Create new mutex object:

```
#include <pthread.h>
pthread_mutex_t new_mutex;
```

- **Init** the mutex (pointer of mutex, attribute of the mutex):

```
pthread_mutex_init(&new_mutex, NULL);
```

Mutual Exclusion (Mutex) in C

- **Lock** and **unlock** the mutex object:

```
pthread_mutex_lock(&new_mutex);
pthread_mutex_unlock(&new_mutex);
```

- **Destroy** the mutex object:

```
pthread_mutex_destroy(&new_mutex);
```

Check **synchronization_3.c** !

Conditional Variables in C

Busy-waiting in application is a bad idea due to:

- Thread consumes CPU even when **can't make progress**.
- Unnecessarily **slows other** threads and processes.

Better to inform scheduler of **which threads can run**, typically done with condition variables:

```
// Initialize
void cond_init (cond_t *, ...);
// Atomically unlock m and sleep until c signaled
// Then re-acquire m and resume executing
void cond_wait (cond_t *c, mutex_t *m);
// Wake one/all threads waiting on c
void cond_signal (cond_t *c);
void cond_broadcast (cond_t *c);
```

Semaphore in C

- Sometimes you need **multiple threads** to enter critical section at the same time i.e.: reader-writer problem.
- Semaphores allow you to define the **# threads** that can be allowed

Library:

```c
#include <semaphore.h>
```

Initialization:

```c
sem_t :
sem_init(sem_t *sem, int pshared, unsigned int value)
```

Usage:

```c
sem_destroy(sem_t *sem)
sem_wait(sem_t *sem)
sem_post(sem_t *sem)
```