Inter Process Communication

Vidisha Thakkar

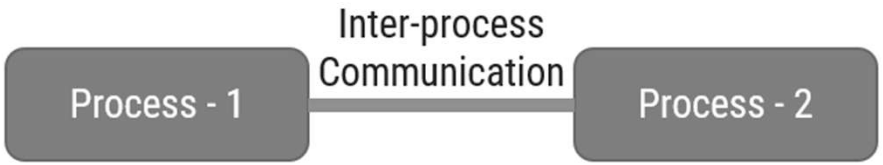
Chapter Outline

- Critical Section
- Race Conditions
- Mutual Exclusion
- Hardware Solution
- Strict Alternation
- Peterson's Solution
- The Producer Consumer Problem
- Semaphores
- Event Counters
- Monitors
- Message Passing
- Classical IPC Problems: Reader's & Writer Problem
- Dinning Philosopher Problem

Inter-process communication (IPC)

- Inter-process communication is the **mechanism provided by the operating system that allows processes to communicate with each other**.
- This communication could involve a process letting another process know that some event has occurred or transferring of data from one process to another.
- Processes in a system can be independent or cooperating.
  - ⮑ Independent process cannot affect or be affected by the execution of another process.
  - ⮑ Cooperating process can affect or be affected by the execution of another process.
- Cooperating processes need inter process communication mechanisms.

Exercise Give an example of independent processes?



Inter-process communication (IPC)

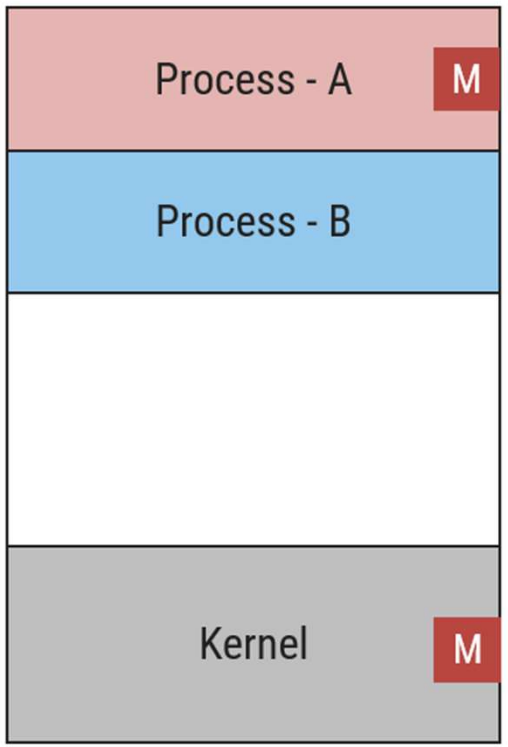
- Reasons of process cooperation
  - **Information sharing**: Several processes may need to access the same data (such as stored in a file.)
  - **Computation speed-up**: A task can often be run faster if it is broken into subtasks and distributed among different processes.
  - **Modularity**: It may be easier to organize a complex task into separate subtasks, then have different processes or threads running each subtask.
  - **Convenience**: An individual user can run several programs at the same time, to perform some task.

- Issues of process cooperation
  - Data corruption, deadlocks, increased complexity
  - Requires processes to synchronize their processing

Models for Inter-process communication (IPC)

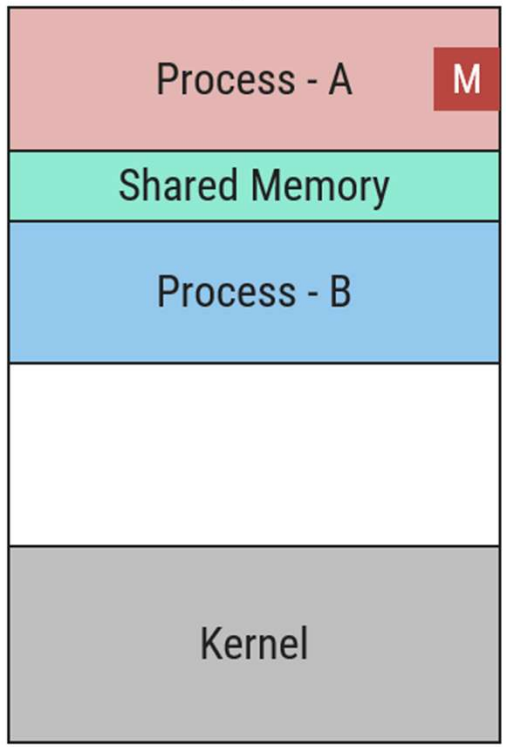
Message Passing

- Process A send the message to Kernel and then Kernel send that message to Process B



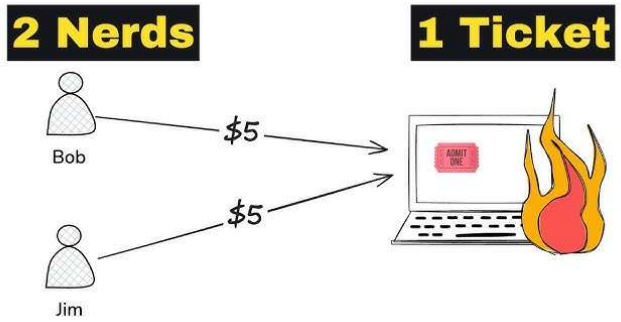
Shared Memory

- Process A put the message into Shared Memory and then Process B read that message from Shared Memory



RACE CONDITION

- The situation where **several processes access and manipulate shared data concurrently**. The **final value of the shared data depends upon which process finishes last**.
- A race condition is an **undesirable situation** that **occurs when a device or system attempts to perform two or more operations at the same time**.
- But, because of the nature of the device or system, the **operations must be done in the proper sequence** to be done correctly.
- To prevent race conditions, **concurrent processes must be synchronized**.

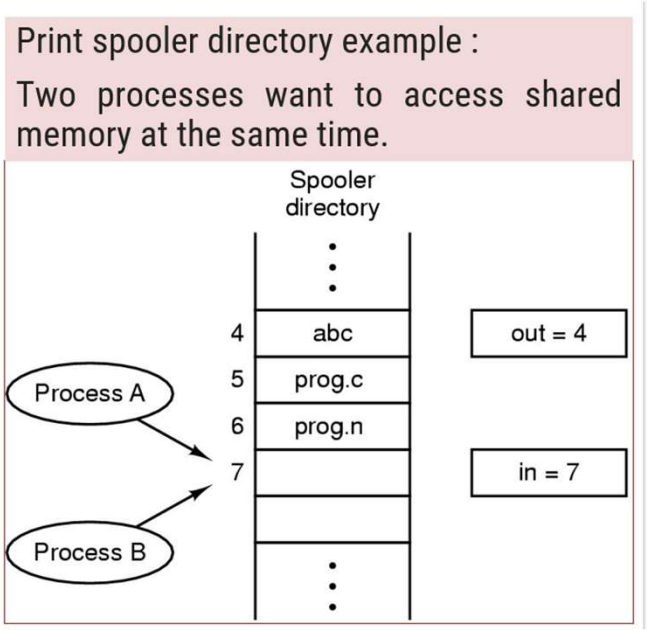


RACE CONDITION EXAMPLE

- **Process A**
- next free slot=in
- write file at slot(**7**)
- next_free_slot+=1
- in=next_free_slot(**8**)

CONTEXT SWITCH

- **Process B**
- next free slot=in
- write file at slot(**8**)
- next_free_slot+=1
- in=next_free_slot(**9**)



- **Process A**
- next free slot=in(**7**)

CONTEXT SWITCH

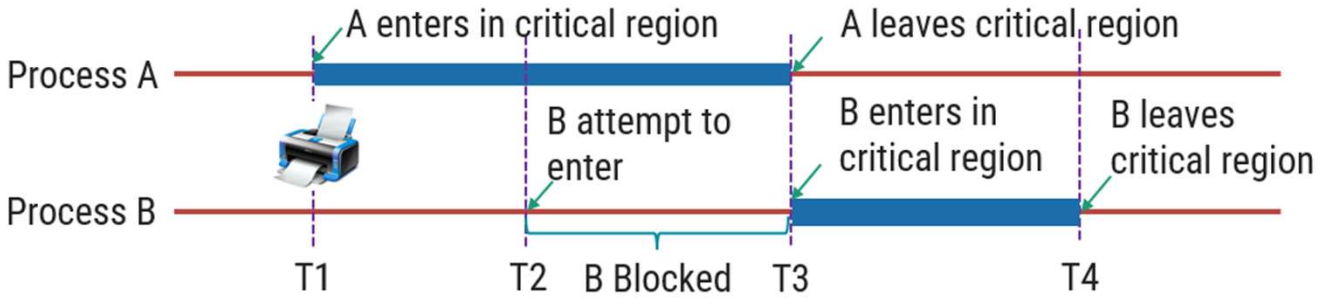
- **Process B**
- next free slot=in(**7**)
- write file at slot(**7**)
- next_free_slot+=1
- in=next_free_slot(**8**)

CONTEXT SWITCH

- **Process A**
- write file at slot(**7**)
- next_free_slot+=1
- in=next_free_slot(**8**)

CRITICAL SECTION

Critical Section: The **part of program where the shared resource is accessed** is called critical section or critical region.



Mutual Exclusion

Mutual Exclusion: Way of making sure that if one process is using a shared variable or file; the other process will be excluded (stopped) from doing the same thing.

Solving Critical-Section Problem

- Any good solution to the problem must satisfy following four conditions:
- Mutual Exclusion
  - No two processes may be simultaneously inside the same critical section.

- Bounded Waiting
  - No process should have to wait forever to enter a critical section.

- Progress
  - No process running outside its critical region may block other processes.

- Arbitrary Speed
  - No assumption can be made about the relative speed of different processes (though all processes have a non-zero speed).

Solution to Critical Section Problem

- Mutual Exclusion with busy waiting
  - **Hardware approaches :**
  - Disabling interrupts
  - Test and Set Lock instruction
  - Exchange instructions
  - **Software approaches :**
  - Shared Lock Variables
  - Strict Alteration
  - Dekker's Solution
  - Peterson's Solution

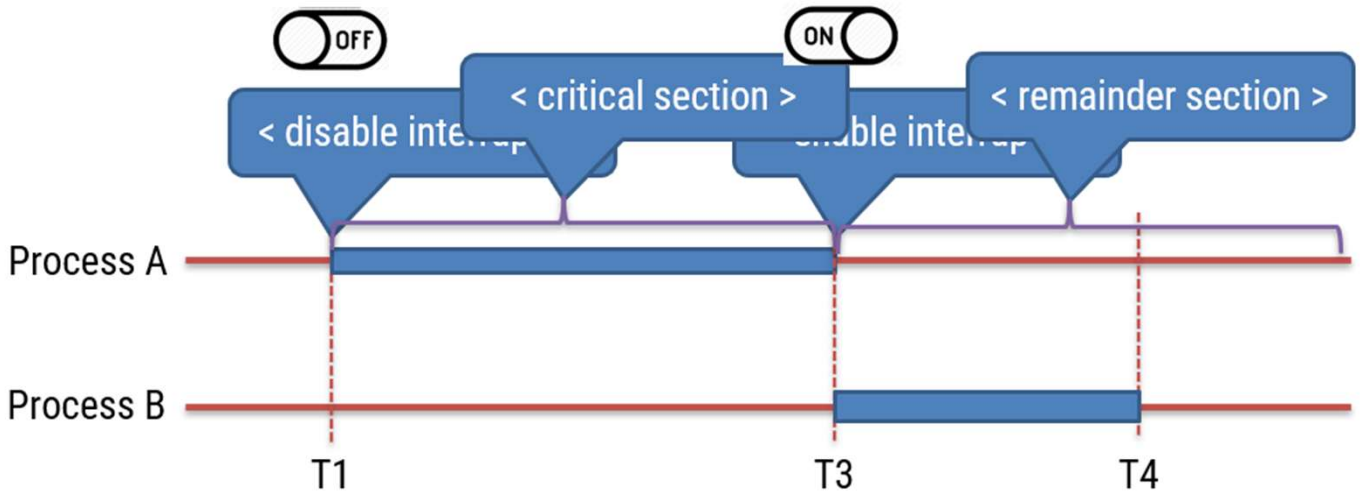
- Mutex
- Semaphores
- Monitors
- Pipes and Message Passing

Real life example to explain mechanisms for achieving mutual exclusion with busy waiting

Disabling interrupts (Hardware approach)

- while (true)

```
{
        < disable interrupts >;
        < critical section >;
        < enable interrupts >;
        < remainder section >;
}
```



Problems in Disabling interrupts (Hardware approach)

- Unattractive or **unwise to give user processes the power to turn off interrupts**.
- What if one of the **process** did it (**disable interrupt**) and **never** turned them on (**enable interrupt**) again? That could be the **end of the system**.
- If the **system is a multiprocessor**, with two or more CPUs, **disabling interrupts affects only the CPU** that executed the disable instruction. The other ones will continue running and can access the shared memory.

Shared lock variable (Software approach)

- A shared variable lock **having value 0 or 1**.
- Before entering into critical region a process checks a shared variable lock's value.
  - **If the value of lock is 0 then set it to 1** before entering the critical section and **enters into critical section** and **set it to 0** immediately after leaving the critical section.
  - **If the value of lock is 1 then wait until it becomes 0** by some other process which is in critical section.

PSEUDO CODE

```
while(lock==1);
lock = 1
```

CRITICAL SECTION ENTRY CODE

CRITICAL SECTION

```
lock = 0
```

CRITICAL SECTION EXIT CODE

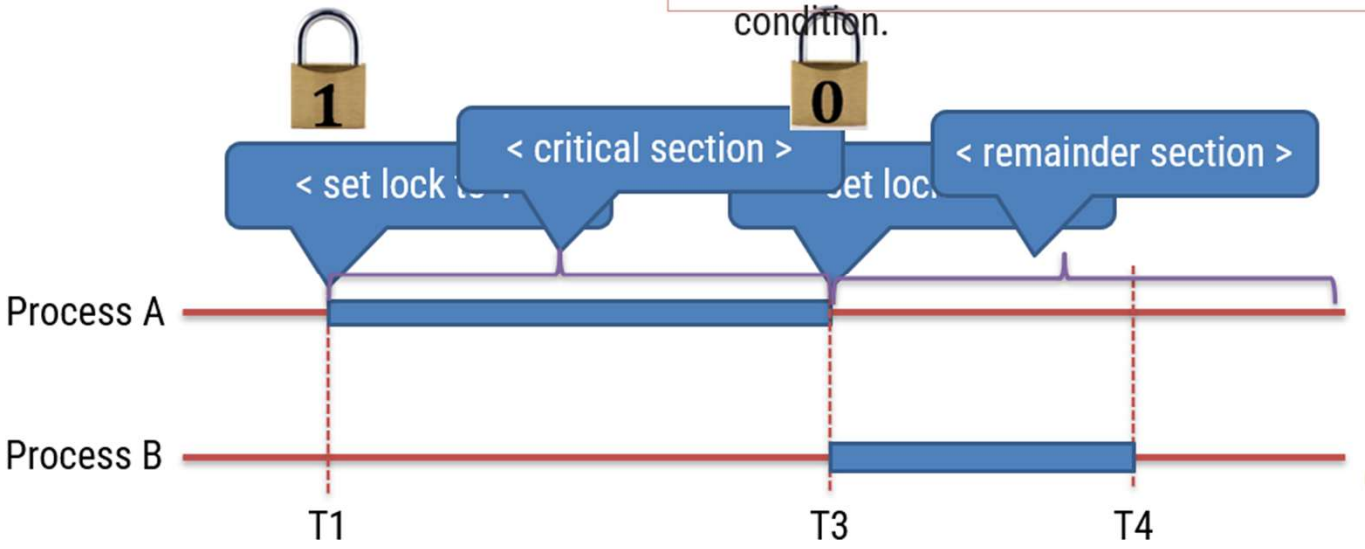
Shared lock variable (Software approach)

- while (true)

```
{       < set shared variable to 1 >;
        < critical section >;
        < set shared variable to 0 >;
        < remainder section >;
}
```

Problem:

- If process-A sees the value of lock variable 0 and before it can set it to 1 context switch occurs.
- Now process-B runs and finds value of lock variable 0, so it sets value to 1, enters critical region.
- At some point of time process-A resumes, sets the value of lock variable to 1, enters critical region.
- Now two processes are in their critical regions accessing the same shared memory, which violates the mutual exclusion condition.

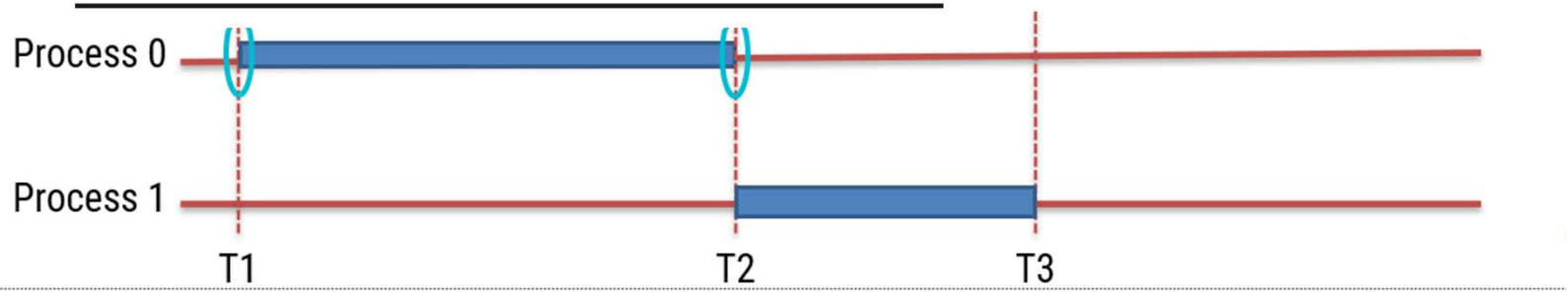


TSL (Test and Set Lock) instruction (Hardware approach)

- Algorithm

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```



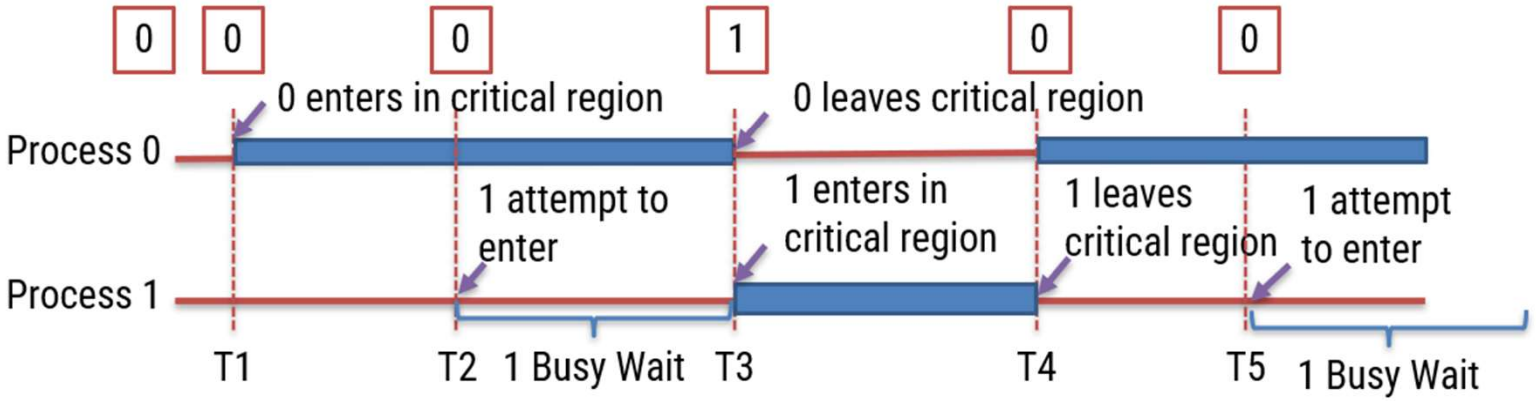
Strict alteration (Software approach)

Process 0

```
while (TRUE)
{
        while (turn != 0) /* loop */ ;
        critical_region();
        turn = 1;
        noncritical_region();
}
```

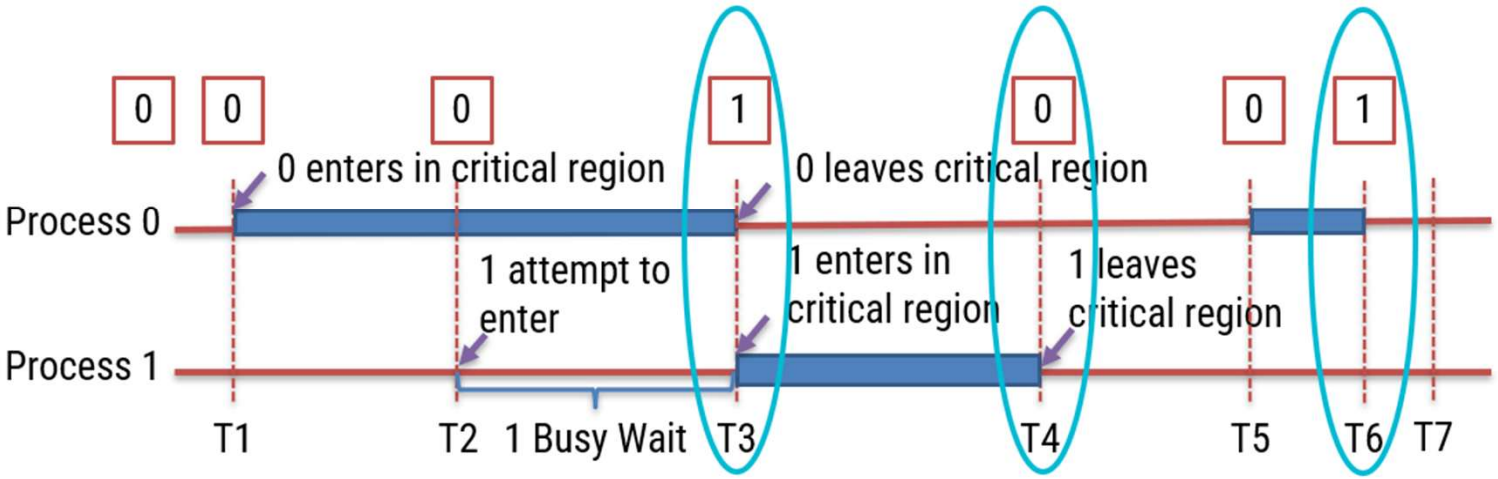
Process 1

```
while (TRUE)
{
        while (turn != 1) /* loop */ ;
        critical_region();
        turn = 0;
        noncritical_region();
}
```



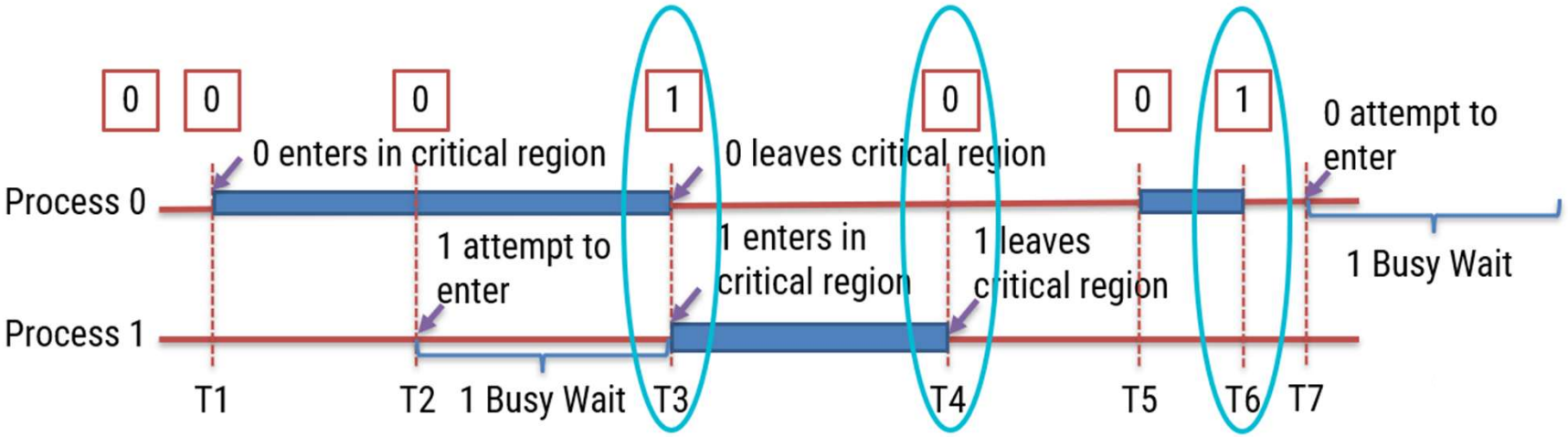
Disadvantages of Strict alteration (Software approach)

- **Taking turns is not a good idea when one of the processes is much slower** than the other.
- Consider the following situation for two processes P0 and P1.
  - P0 leaves its critical region, set turn to 1, enters non critical region.
  - P1 enters and finishes its critical region, set turn to 0.
  - Now both P0 and P1 in non-critical region.
  - P0 finishes non critical region, enters critical region again, and leaves this region, set turn to 1.
  - P0 and P1 are now in non-critical region.



Disadvantages of Strict alteration (Software approach)

- **Taking turns is not a good idea when one of the processes is much slower** than the other.
- Consider the following situation for two processes P0 and P1.
  - P0 finishes non critical region but cannot enter its critical region because turn = 1 and it is turn of P1 to enter the critical section.
  - Hence, P0 will be blocked by a process P1 which is not in critical region. This violates one of the conditions of mutual exclusion.
  - It wastes CPU time, so we should avoid busy waiting as much as we can.



SOLUTION USING FLAG BIT

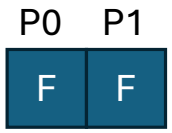
- P0

```
while(1)
{
flag[0]=T
while(flag[1]);
CS();
flag[0]=F
}
```

- P1

```
while(1)
{
flag[1]=T
while(flag[0]);
CS();
flag[1]=F
}
```

flag array → Array of two boolean integer of the choice (whether process wants to enter Critical Section or not.



flag array

If a process wants to enter to CS, it will check the choice of other process, if other process is willing to enter the critical section, the process will let other process to enter to the CS catering the mutual exclusion condition.

Find out is this solution perfect?

SOLUTION USING PETERSON'S ALGORITHM

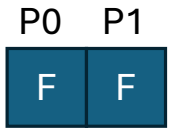
P0

```
while(1)
{
flag[0]=T
turn=1
while(flag[1]==T && turn==1);
CS();
flag[0]=F
        }
```

P1

```
while(1)
{
flag[1]=T
turn=0
while(flag[0]==T && turn==0);
CS();
flag[1]=F
        }
```

flag array → Array of two boolean integer of the choice (whether process wants to enter Critical Section or not.



flag array

If a process wants to enter to CS, it will check the choice of other process as well as give chance the other process in the variable turn, if other process is willing to enter the critical section, the process will let other process to enter to the CS catering the mutual exclusion condition.

PRIORITY INVERSION PROBLEM

Priority Inversion Problem

Priority Inversion is a problem in scheduling where a **high-priority task** is forced to wait because a **low-priority task** is holding a required resource (like a lock or shared variable).

This leads to **unexpected delays** and can cause critical system failures, especially in **real-time systems**.

Imagine a system with three tasks:

Task	Priority
T1 (Low-priority)	**Low**
T2 (Medium-priority)	**Medium**
T3 (High-priority)	**High**

Scenario:

1.T1 (low-priority task) acquires a lock to access a shared resource.

2.T3 (high-priority task) arrives and needs the same resource, but it is locked by T1.

1. **T3 must wait** for T1 to finish.

3.Meanwhile, **T2 (medium-priority task) starts running** because it does not need the lock.

4.T2 keeps executing, preventing T1 from finishing.

5.T3 (high-priority task) remains stuck waiting for T1, even though T3 should have executed before T2!

Mutual Exclusion with Busy Waiting

- Disabling Interrupts
  - ↪ is not appropriate as a general mutual exclusion mechanism for user processes
- Lock Variables
  - ↪ contains exactly the same fatal flaw that we saw in the spooler directory
- Strict Alternation
  - ↪ process running outside its critical region blocks other processes.
- Peterson's Solution
- The TSL/XCHG instruction
  - ↪ Both Peterson's solution and the solutions using TSL or XCHG are correct.
  - ↪ Limitations:
    - Busy Waiting: this approach waste CPU time
    - Priority Inversion Problem: a low-priority process blocks a higher-priority one

Busy Waiting (Sleep and Wakeup)

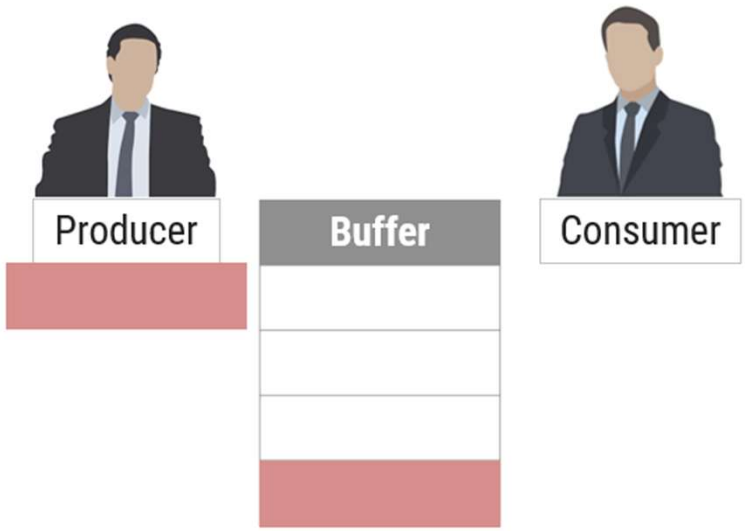
- Peterson's solution and solution using TSL and XCHG have the limitation of requiring **busy waiting**.
  - when a processes **wants to enter** in its critical section, it **checks** to see if the entry is allowed.
  - If it is **not allowed**, the process **goes into a loop and waits** (i.e., **start busy waiting**) until it is allowed to enter.
  - This approach **waste CPU-time**.
- But we have interprocess communication primitives (the **pair of sleep & wakeup**).
- **Sleep**: It is a system call that **causes the caller to be blocked** (**suspended**) until some other process wakes it up.
- **Wakeup**: It is a system call that **wakes up** the process.
- Both 'sleep' and 'wakeup' system calls have one parameter that represents a memory address used to match up 'sleeps' and 'wakeups'.

The Producer Consumer Problem

Section - 3

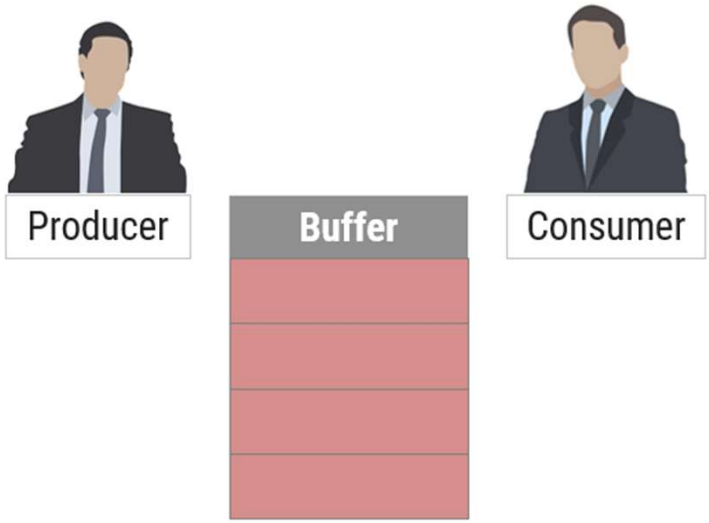
The Producer Consumer Problem

- It is **multi-process synchronization** problem.
- It is also known as **bounded buffer problem**.
- This problem describes two processes producer and consumer, who share common, fixed size buffer.
- Producer process
  - ↪ Produce some information and put it into buffer
- Consumer process
  - ↪ Consume this information (remove it from the buffer)



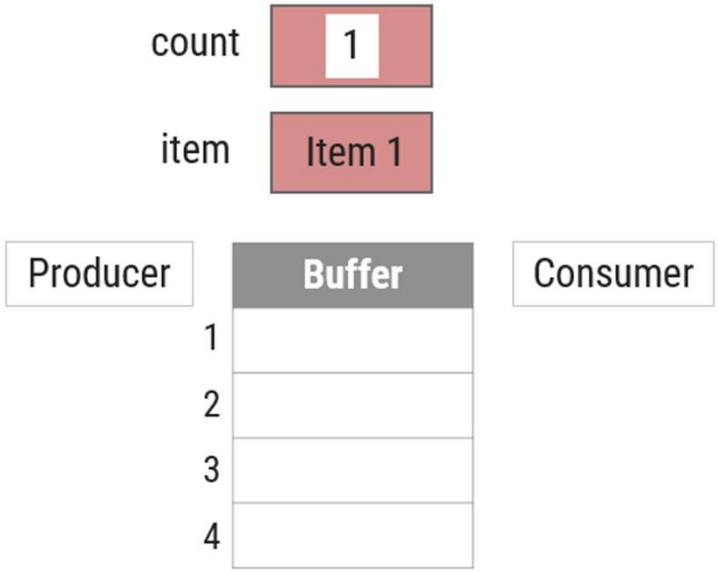
What Producer Consumer problem is?

- Buffer is **empty**
  - **Producer** want to **produce** √
  - **Consumer** want to **consume** X
- Buffer is **full**
  - **Producer** want to **produce** X
  - **Consumer** want to **consume** √
- Buffer is **partial filled**
  - **Producer** want to **produce** √
  - **Consumer** want to **consume** √



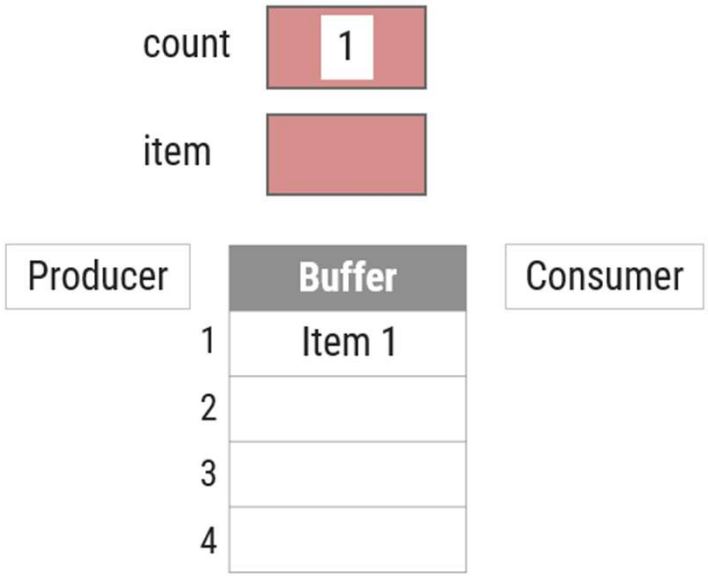
Producer Consumer problem using Sleep & Wakeup

```
#define N 4
int count=0;
void producer (void)
{     int item;
      while (true)
      {     item=produce_item();
            if (count==N) sleep();
            insert_item(item);
            count=count+1;
            if(count==1) wakeup(consumer);
      }
}
```



Producer Consumer problem using Sleep & Wakeup

```
void consumer (void)
{       int item;
        while (true)
        {       if (count==0) sleep();
                item=remove_item();
                count=count-1;
                if(count==N-1)
                        wakeup(producer);
                consume_item(item);
        }
}
```



Problem in Sleep & Wakeup

- Problem with this solution is that it **contains a race condition that can lead to a deadlock**. (How???)

Producer

```
#define N 4
int count=0;
void producer (void)
{       int item;
        while (true)
        {       item=produce_item();
                if (count==N) sleep();
                insert_item(item);
                count=count+1;
                if(count==1) wakeup(consumer);
        }
}
```

Wakeup Signal

count: 1

item: Item 1

	Buffer
1	
2	Item 2
3	Item 3
4	Item 4

Consumer

```
void consumer (void)
{       int item;
        while (true)
        {       if (count==0) sleep();
                item=remove_item();
                count=count-1;
                if(count==N-1)
                        wakeup(producer);
                consume_item(item);
        }
}
```

Context Switching

Semaphore

Section - 4

Semaphore

- A semaphore is a **variable that provides an abstraction for controlling the access of a shared resource** by multiple processes in a parallel programming environment.
- There are 2 types of semaphores:
  - **Binary semaphores** :-
    - Binary semaphores can take only 2 values (0/1).
    - Binary semaphores have 2 methods associated with it (up, down / lock, unlock).
    - They are used to acquire locks.
  - **Counting semaphores** :-
    - Counting semaphore can have possible values more than two.

Semaphore

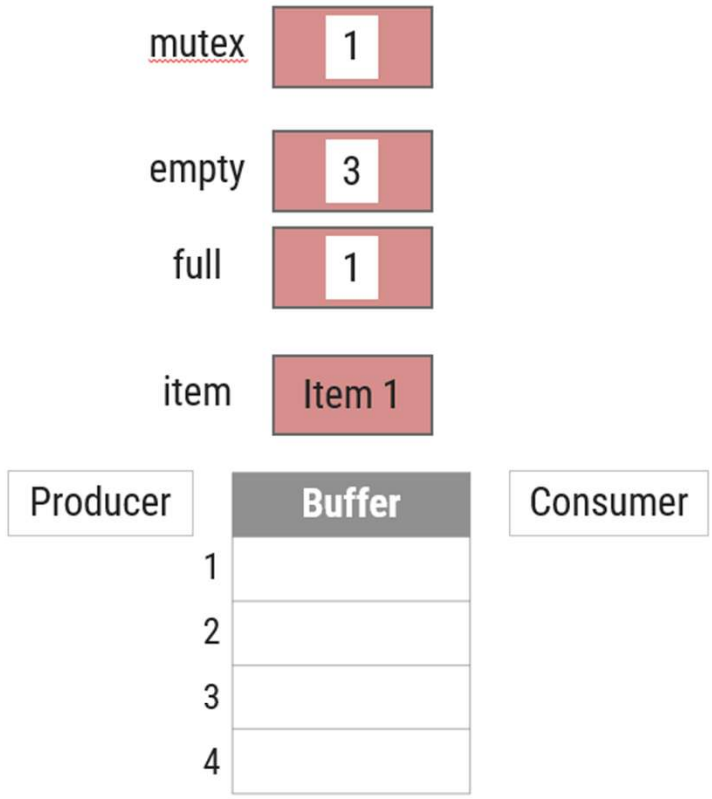
- We want functions insert _item and remove_item such that the following hold:
  - **Mutually exclusive access to buffer**: At any time only one process should be executing (either insert_item or remove_item).
  - **No buffer overflow**: A process executes insert_item only when the buffer is not full (i.e., the process is blocked if the buffer is full).
  - **No buffer underflow**: A process executes remove_item only when the buffer is not empty (i.e., the process is blocked if the buffer is empty).
  - **No busy waiting**.
  - **No producer starvation**: A process does not wait forever at insert_item() provided the buffer repeatedly becomes full.
  - **No consumer starvation**: A process does not wait forever at remove_item() provided the buffer repeatedly becomes empty.

Operations on Semaphore

- Wait(): a process performs a wait operation to **tell the semaphore that it wants exclusive access to the shared resource**.
  - If the semaphore is empty, then the semaphore enters the full state and allows the process to continue its execution immediately.
  - If the semaphore is full, then the semaphore suspends the process (and remembers that the process is suspended).
- Signal(): a process performs a signal operation to **inform the semaphore that it is finished using the shared resource**.
  - If there are processes suspended on the semaphore, the semaphore wakes one of the up.
  - If there are no processes suspended on the semaphore, the semaphore goes into the empty state.

Producer Consumer problem using Semaphore

```
#define N 4
typedef int semaphore;
semaphore mutex=1;
semaphore empty=N;
semaphore full=0;
void producer (void)
{           int item;
            while (true)
            {           item=produce_item();
                        down(&empty);
                        down(&mutex);
                        insert_item(item);
                        up(&mutex);
                        up(&full);          }
}
```



Producer Consumer problem using Semaphore

```
void consumer (void)
{       int item;
        while (true)
        {
                down(&full);
                down(&mutex);
                item=remove_item(item);
                up(&mutex);
                up(&empty);
                consume_item(item);
        }
}
```