Basic Definitions

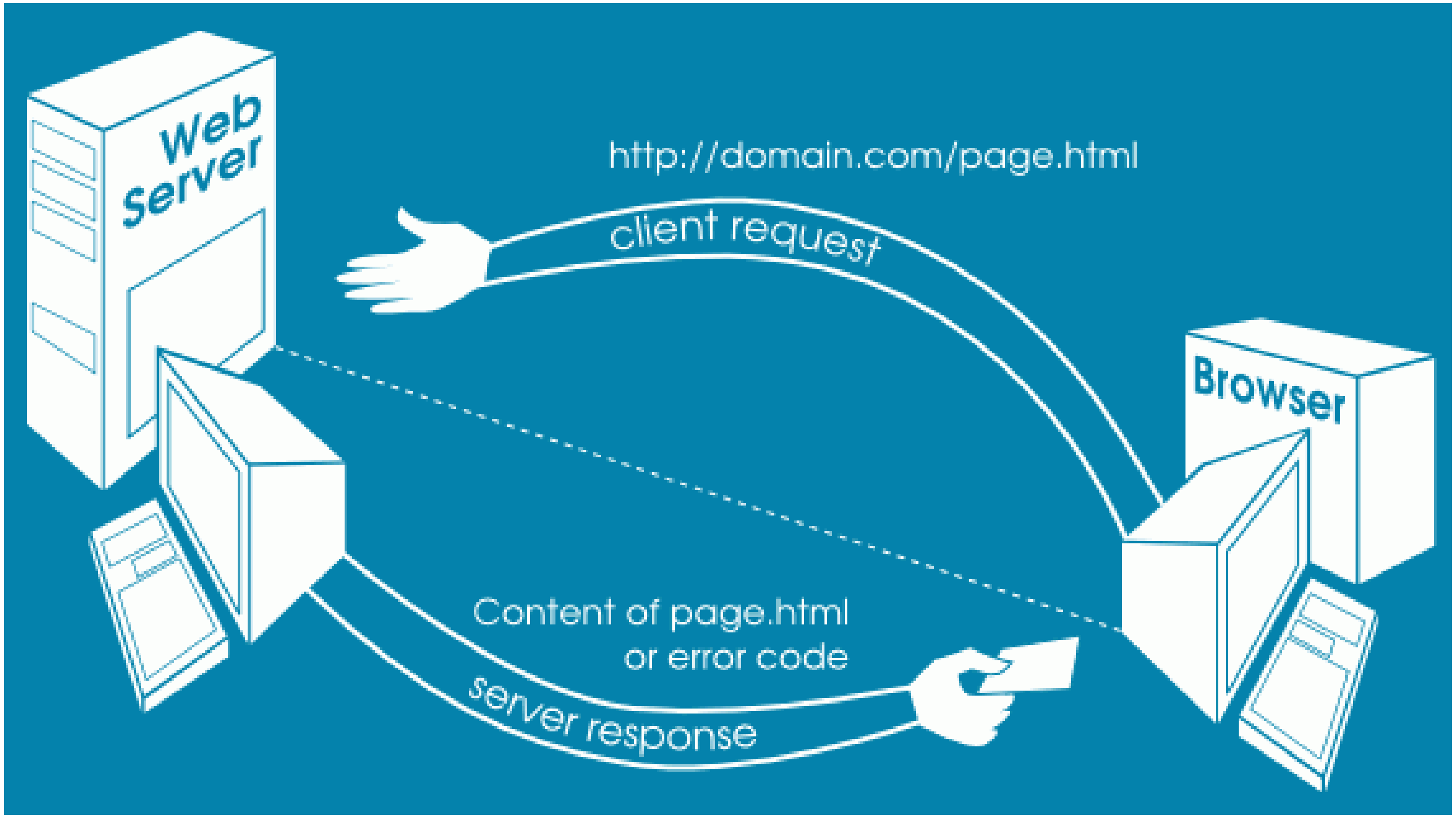
- **Inter Process Communication**: It is a **communication between two or more processes**.

Basic Definitions

- **Inter Process Communication**: It is a **communication between two or more processes**.

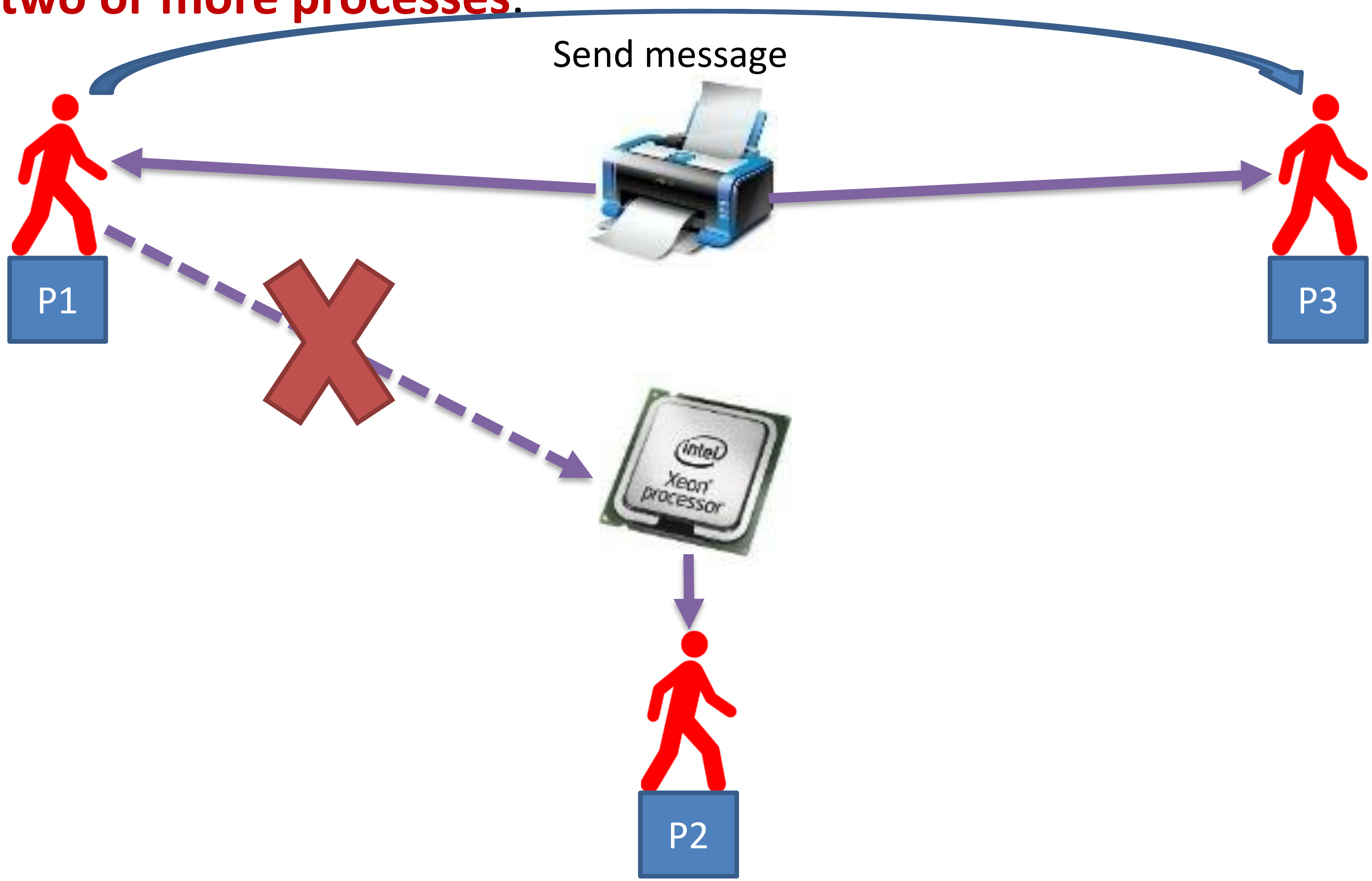
Basic Definitions

- **Inter Process Communication**: It is a **communication between two or more processes**.



Basic Definitions

- **Inter Process Communication**: It is a **communication between two or more processes**.



Inter process communication (IPC)

- Processes in a system can be independent or cooperating.
  1. **Independent process cannot affect** or be **affected** by the execution of another process.
  2. **Cooperating process can affect** or be **affected** by the execution of another process.

- Cooperating processes need **inter process communication** mechanisms.

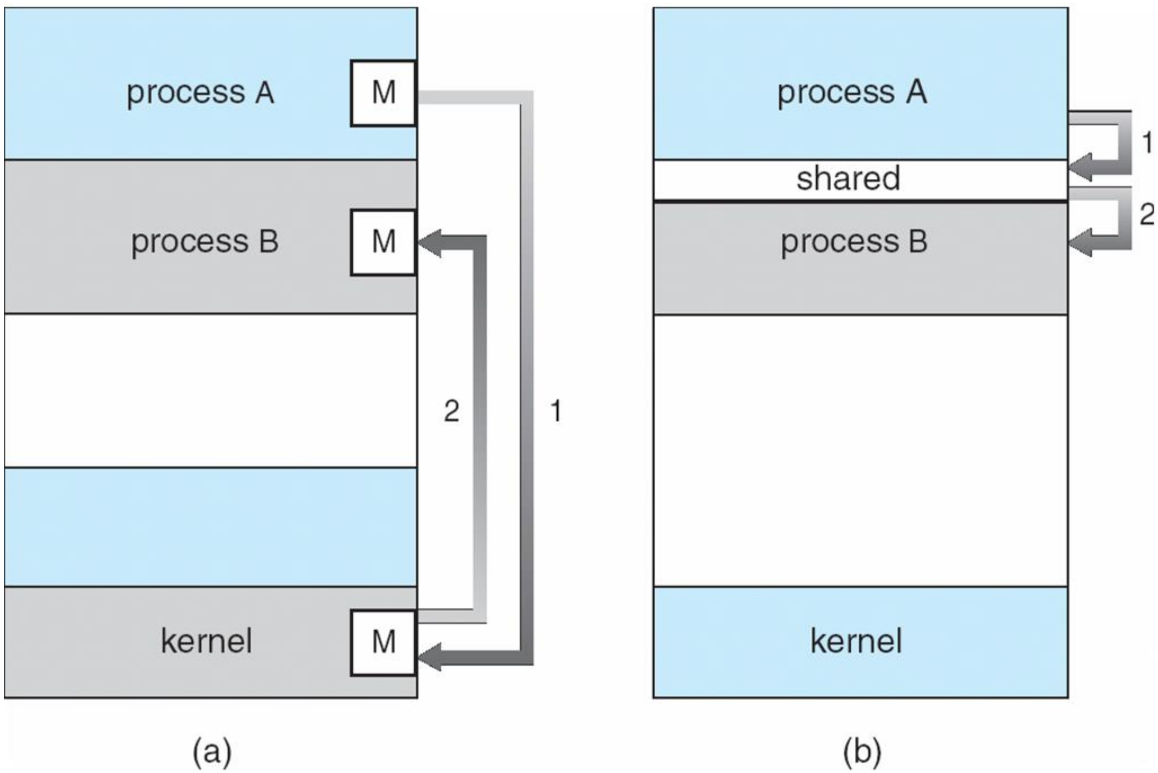
Inter process communication (IPC)

- Reasons of process cooperation
  1. Information sharing
  2. Computation speed-up
  3. Modularity
  4. Convenience

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

Models for Inter Process Communication (IPC)

- There are two models for IPC
  a. **Message Passing** (Process A send the message to Kernel and then Kernel send that message to Process B)
  b. **Shared Memory** (Process A put the message into Shared Memory and then Process B read that message from Shared Memory)



Race Condition

- **Race Condition**:
  - 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.

Basic Definitions

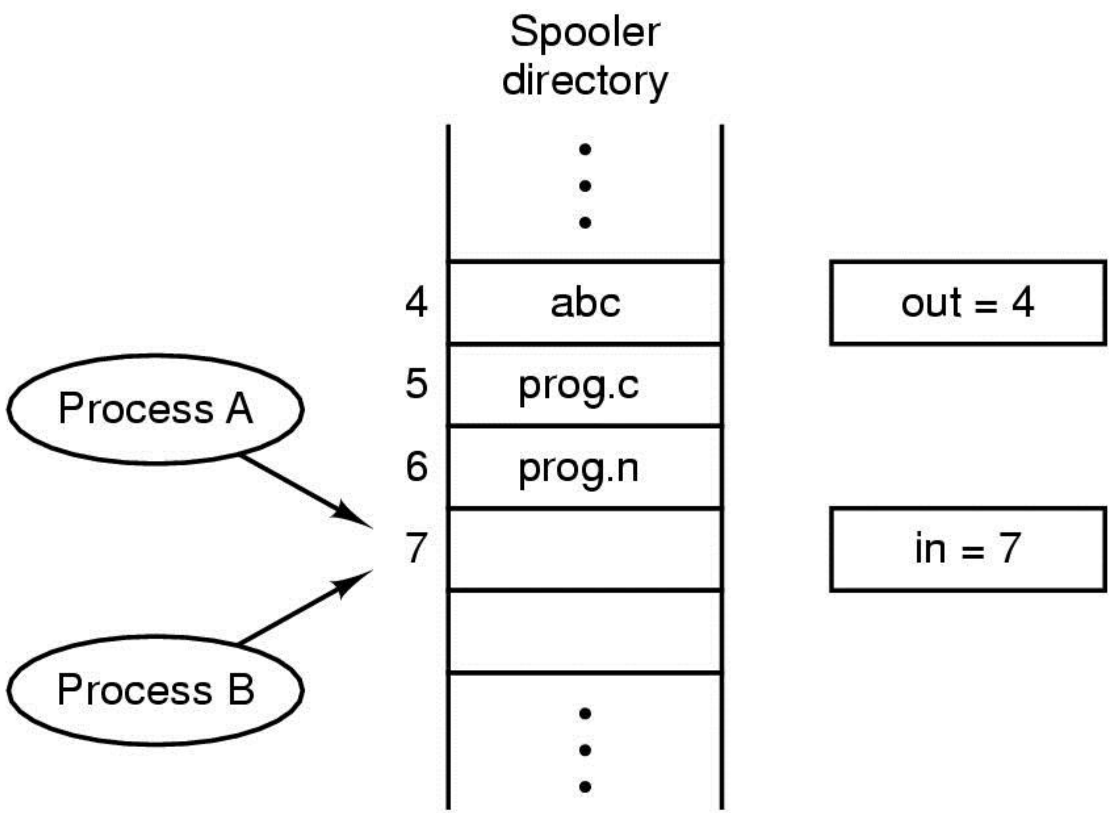
- **Race Condition**: **Situations** like this **where processes access the same data concurrently** and the **outcome of execution depends on the particular order** in which the access takes place is called a race condition. **OR**

- **Situation where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when**.

- Reasons for Race Condition
  1. Exact instruction execution order cannot be predicted
  2. Resource (file, memory, data etc…) sharing

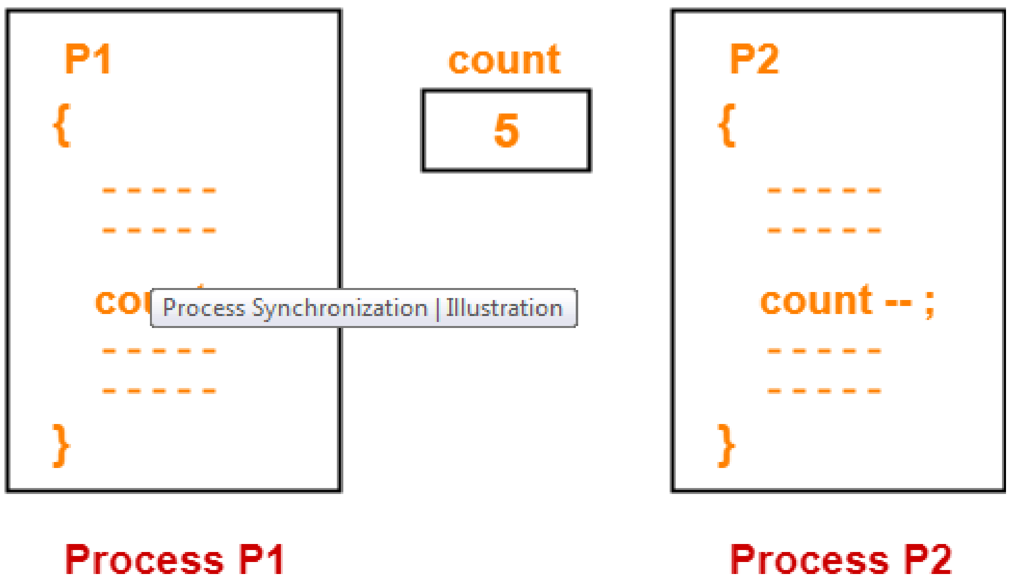
Example of Race Condition

- Print spooler directory example : Two processes want to access shared memory at the same time.

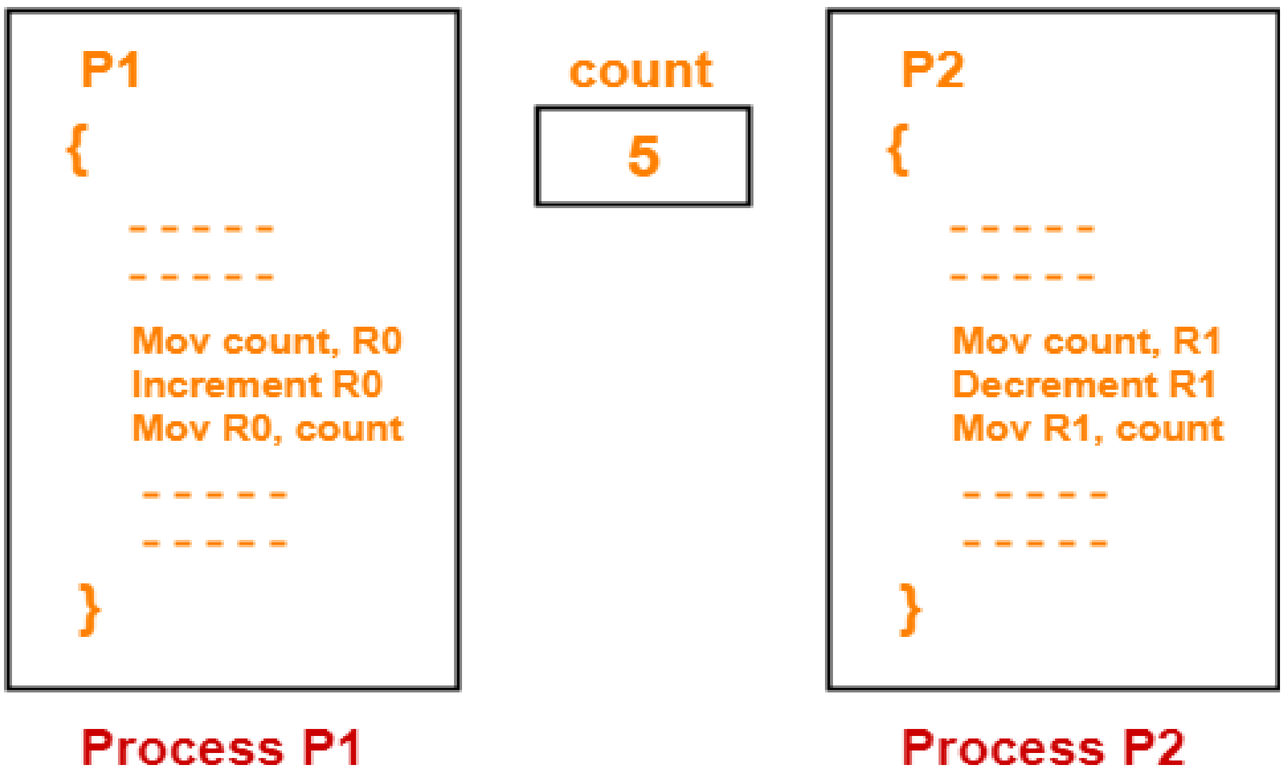


Race Condition

- Two processes P_1 and P_2 are executing concurrently.
- Both the processes share a common variable named "count" having initial value = 5.
- Process P1 tries to increment the value of count.
- Process P2 tries to decrement the value of count.



Race Condition



Race Condition

Case 1: The execution order of the instructions may be-

- $P_1(1)$, $P_1(2)$, $P_1(3)$, $P_2(1)$, $P_2(2)$, $P_2(3)$
- Final value of count = 5

Case 2: $P_2(1)$, $P_2(2)$, $P_2(3)$, $P_1(1)$, $P_1(2)$, $P_1(3)$

- In this case,
- Final value of count = 5

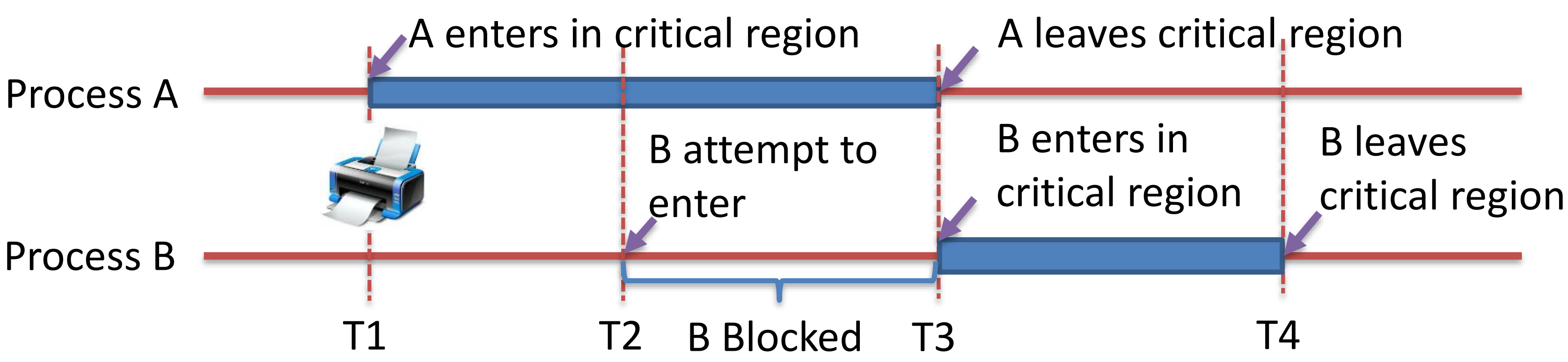
- It is clear from here that inconsistent results may be produced if multiple processes execute concurrently without any synchronization.

Basic Definitions

- Critical Section:

Basic Definitions

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



Basic Definitions

- **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:

1. Mutual Exclusion
   - **No two processes** may be **simultaneously inside** the same critical section.
2. Bounded Waiting
   - **No process should have to wait forever** to enter a critical section.
3. Progress
   - **No process running outside** its critical region may **block other processes**.
4. Arbitrary Speed
   - **No assumption can be made** about the relative speed of different processes (though all processes have a non-zero speed).

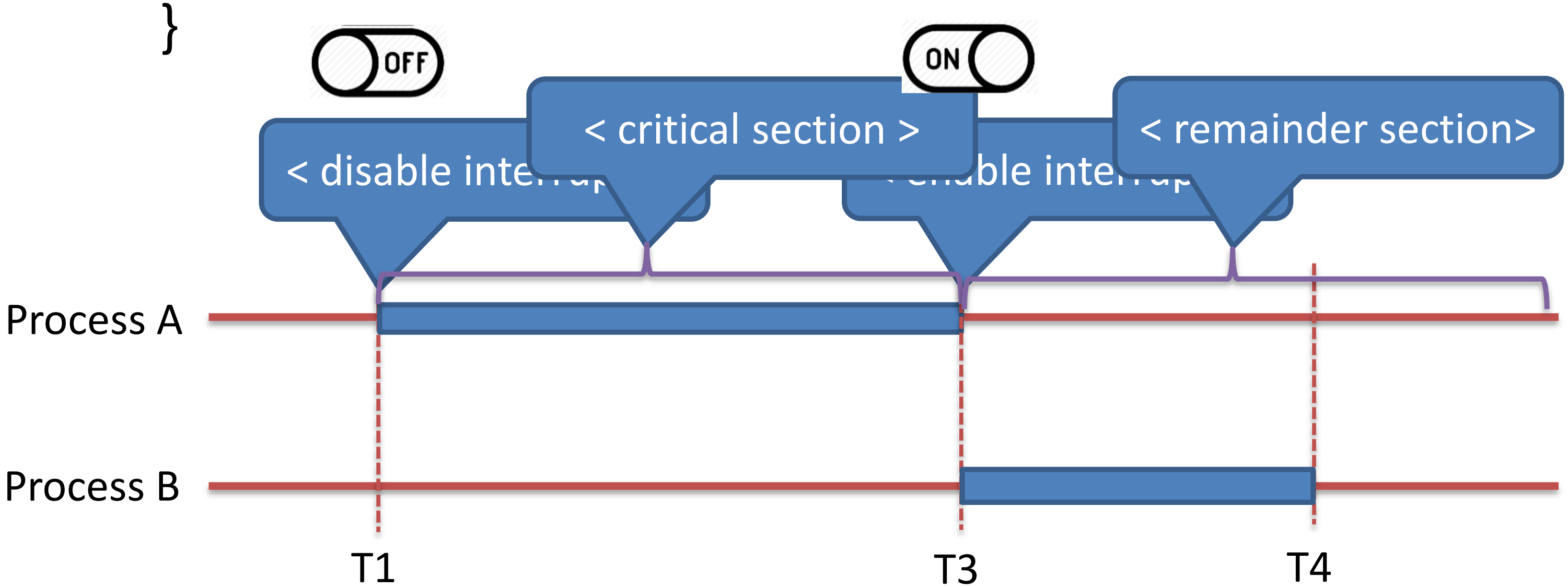
Mutual exclusion with busy waiting

- Mechanisms for achieving mutual exclusion with busy waiting
  1. Disabling interrupts (Hardware approach)
  2. Shared lock variable (Software approach)
  3. Strict alteration (Software approach)
  4. TSL (Test and Set Lock) instruction (Hardware approach)
  5. Exchange instruction (Hardware approach)
  6. Dekker’s solution (Software approach)
  7. Peterson’s solution (Software approach)

Disabling interrupts

- while (true)

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



Disabling interrupts

- Problems:
  - Unattractive or **unwise to give user processes the power to turn off interrupts**.
  - What if one of them 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

- 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.

Shared lock variable

- Algorithm:
- while (true)

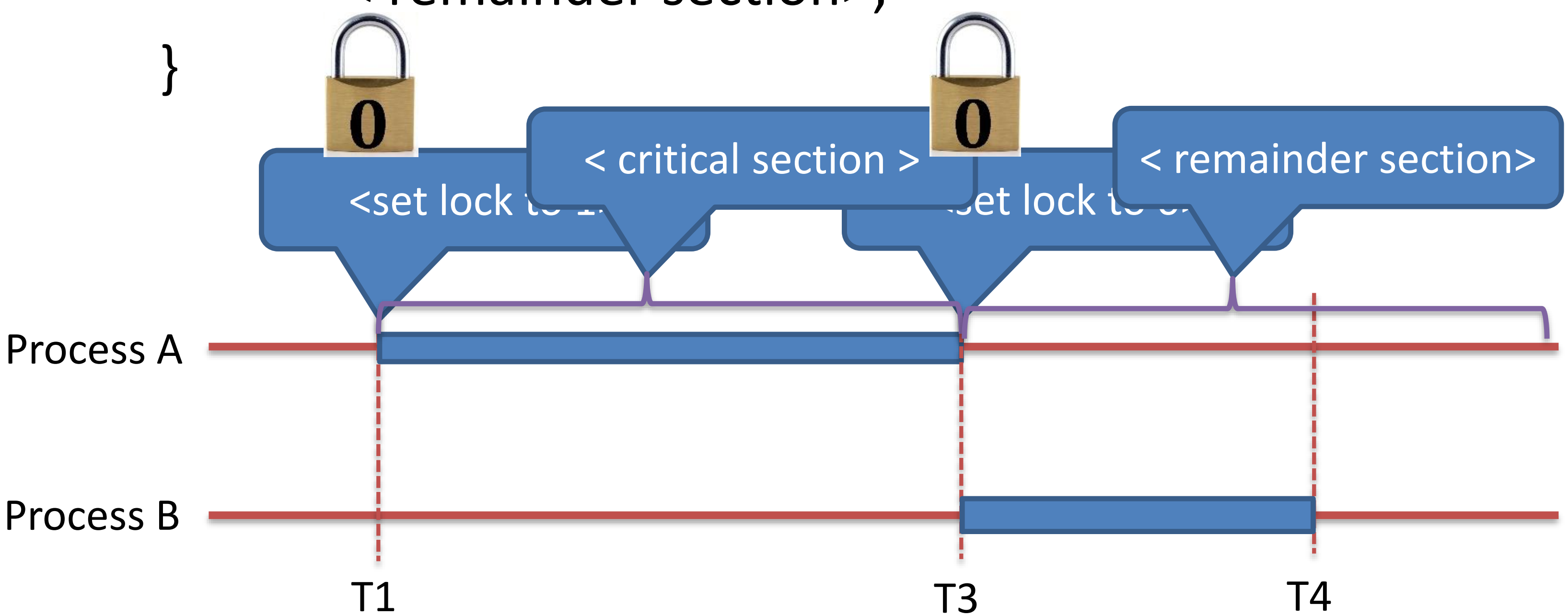
    {　　　< set shared variable to 1>;

    　　　< critical section >;

    　　　< set shared variable to 0>;

    　　　< remainder section>;

    }



Shared lock variable

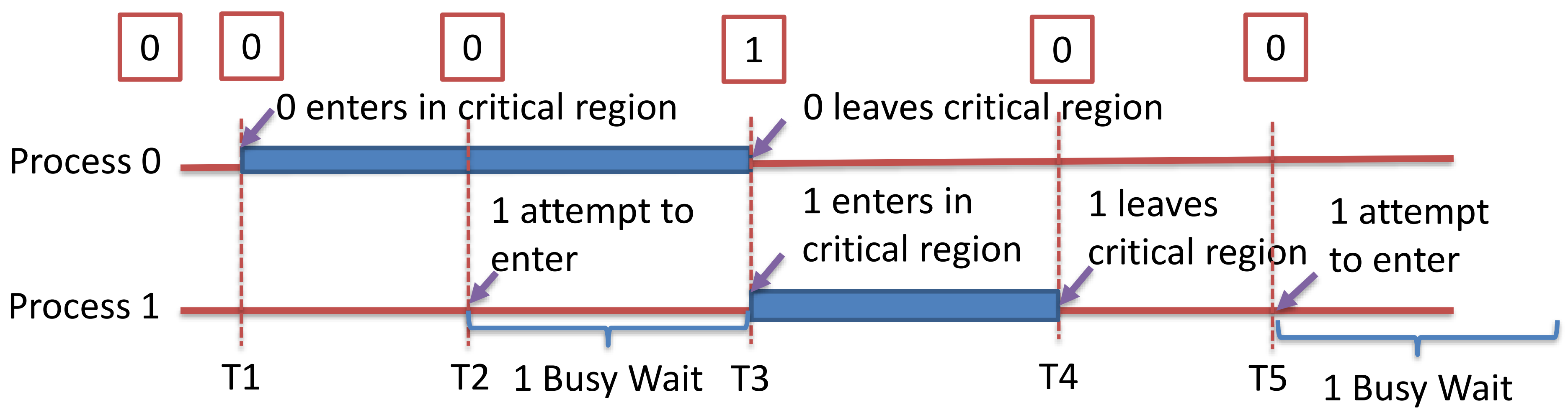
- Problem:
  - If process **P0 sees the value of lock variable 0** and **before it can set it to 1 context switch occurs**.
  - Now process **P1 runs and finds value of lock variable 0**, so it **sets value to 1, enters critical region**.
  - **At some point of time P0 resumes**, **sets the value of 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.

Strict Alteration

- Integer **variable 'turn' keeps track of whose turn is** to enter the critical section.
- **Initially turn=0**. Process 0 inspects turn, finds it to be 0, and enters in its critical section.
- **Process 1 also finds it to be 0** and therefore **sits in a loop** continually testing 'turn' to see **when it becomes 1**.
- **Continuously testing a variable** waiting for some event to appear is called the **busy waiting**.
- When **process 0 exits** from critical region it **sets turn to 1** and now **process 1 can find it to be 1 and enters in to critical region**.
- In this way, both the processes get **alternate turn** to enter in critical region.

Strict Alteration (Algorithm)

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

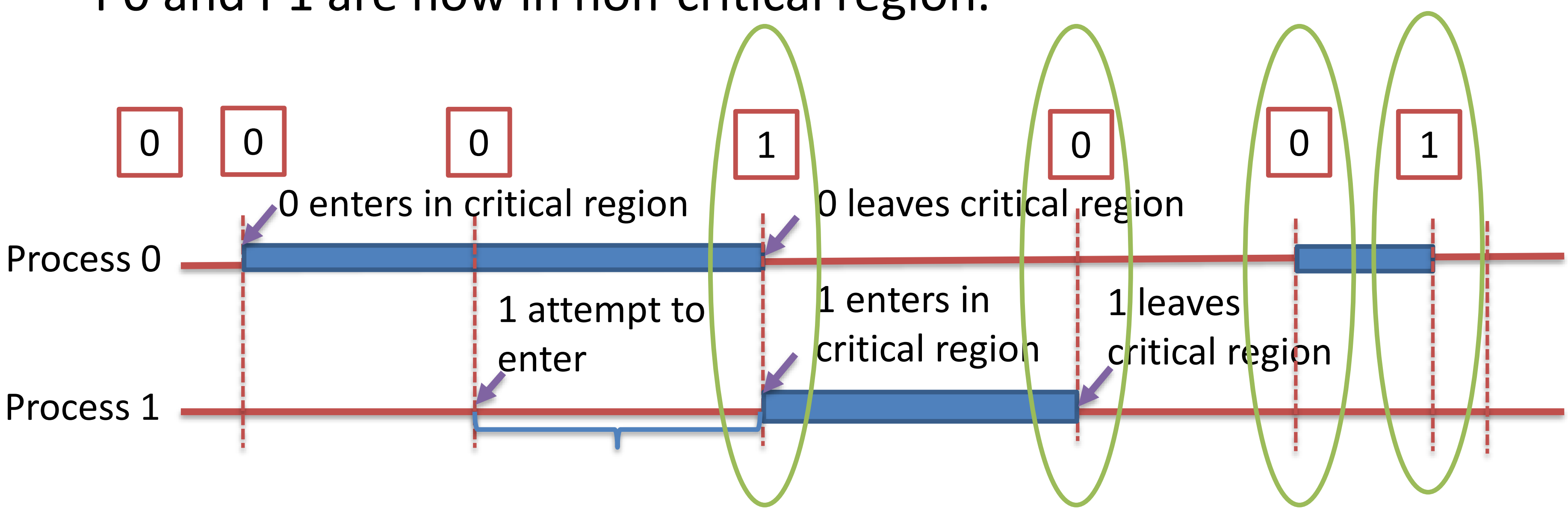


Strict Alteration (Disadvantages)

- **Taking turns is not a good idea when one of the processes is much slower** than the other.

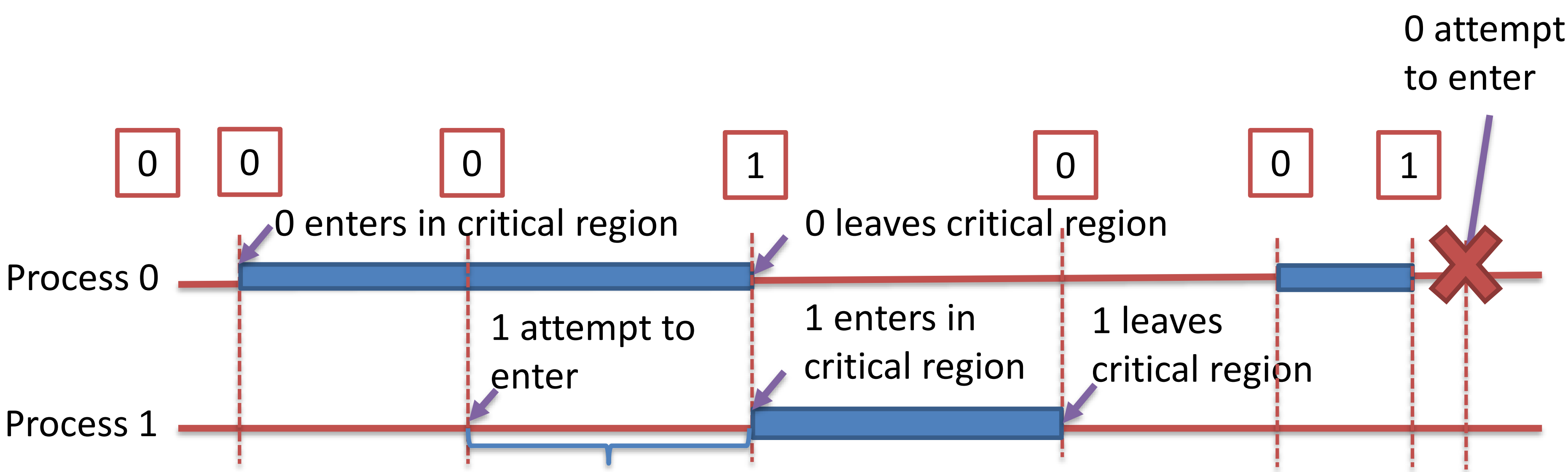
Strict Alteration (Disadvantages)

- 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.



Strict Alteration (Disadvantages)

- 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.



TSL (Test and Set Lock) Instruction

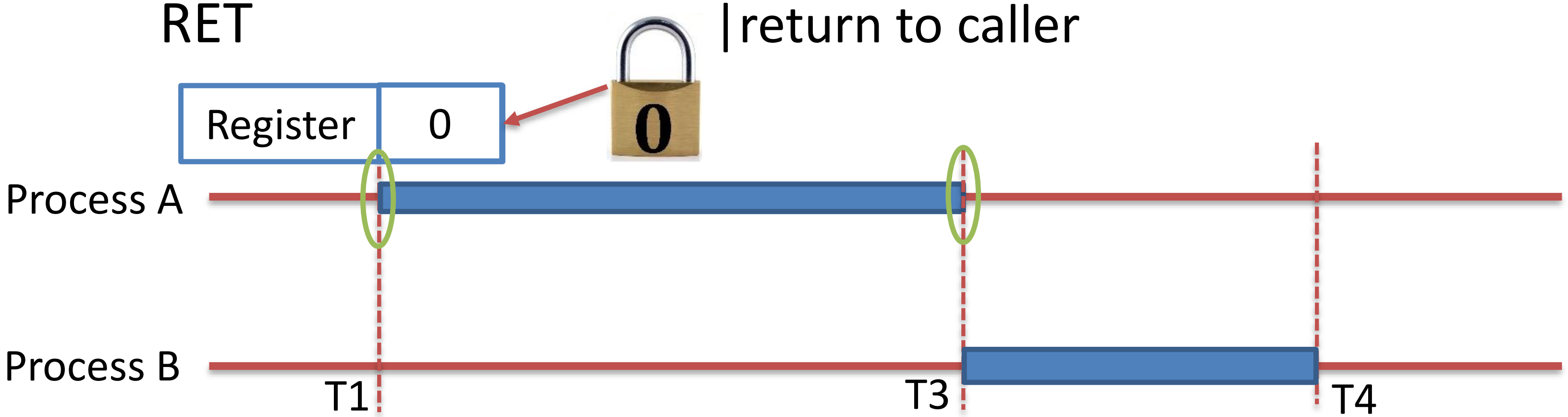
- Algorithm

enter_region: (Before entering its critical region, process calls enter_region)

```
    TSL REGISTER,LOCK      |copy lock variable to register set lock to 1
    CMP REGISTER,#0        |was lock variable 0?
    JNE enter_region       |if it was nonzero, lock was set, so loop
    RET                    |return to caller: critical region entered
```

leave_region: (When process wants to leave critical region, it calls leave_region)

```
    MOVE LOCK,#0           |store 0 in lock variable
    RET                    |return to caller
```



The TSL Instruction

- Test and Set Lock Instruction

  TSL REGISTER, LOCK

  - It reads the contents of the memory word lock into register RX and then stores a nonzero value at the memory address lock.
  - The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished.
  - The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

Exchange Instruction

- Algorithm

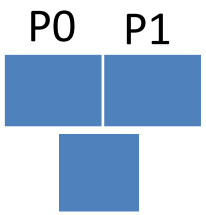
```
enter_region:        (Before entering its critical region, process calls enter_region)
    MOVE REGISTER,#1      |put 1 in the register
    XCHG REGISTER,LOCK    |swap content of register & lock variable
    CMP REGISTER,#0       |was lock variable 0?
    JNE enter_region      |if it was nonzero, lock was set, so loop
    RET                   |return to caller: critical region entered
leave_region:        (When process wants to leave critical region, it calls leave_region)
    MOVE LOCK,#0          |store 0 in lock variable
    RET                   |return to caller
```

Dekker's Algorithm

variables

wants_to_enter [2]: array of 2 booleans

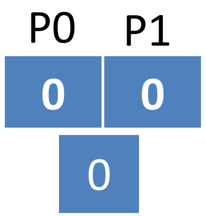
turn : integer



wants_to_enter[0] ← false

wants_to_enter[1] ← false

turn ← 0   // or 1



Dekker's Algorithm

P0:

```
wants_to_enter[0] ← true
while (wants_to_enter[1])
  {if (turn == 1)
    {wants_to_enter[0] ← false
                while (turn == 1)
         {// busy wait}
    wants_to_enter[0] ← true
    }
  }
// critical section
...
turn ← 1
wants_to_enter[0] ← false
// remainder section
```

P0	P1
1	0

1

P0	P1
0	0

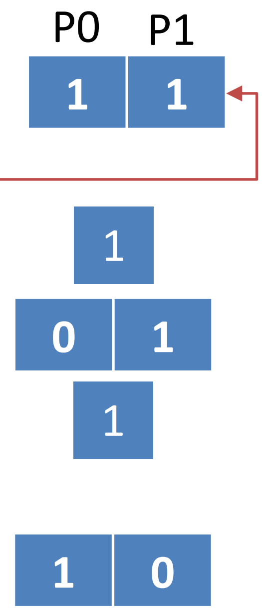
P1:

```
wants_to_enter[1] ← true
while (wants_to_enter[0])
  {if (turn == 0)
    {wants_to_enter[1] ← false
                while (turn == 0)
         {// busy wait}
    wants_to_enter[1] ← true
    }
  }
// critical section
...
turn ← 0
wants_to_enter[1] ← false
// remainder section
```

Dekker’s Algorithm

P0:

```
wants_to_enter[0] ← true
while (wants_to_enter[1])
  {if (turn == 1)
    {wants_to_enter[0] ← false
      while (turn == 1)
      {// busy wait}
    wants_to_enter[0] ← true
    }
  }
// critical section
...
turn ← 1
wants_to_enter[0] ← false
// remainder section
```

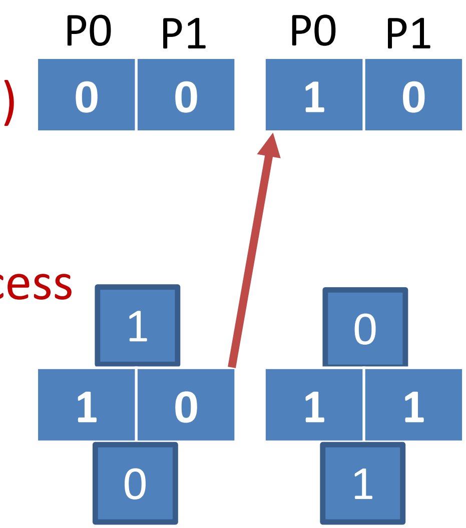


P1:

```
wants_to_enter[1] ← true
while (wants_to_enter[0])
  {if (turn == 0)
    {wants_to_enter[1] ← false
      while (turn == 0)
      {// busy wait}
    wants_to_enter[1] ← true
    }
  }
// critical section
...
turn ← 0
wants_to_enter[1] ← false
// remainder section
```

Peterson’s Solution

```
#define FALSE 0
#define TRUE 1
#define N 2                                  //number of processes
int turn;                                    //whose turn is it?
int interested[N];                           //all values initially 0 (FALSE)
void enter_region(int process)
{
int other;                                   // number of the other process
other = 1 - process;                         // the opposite process
interested[process] = TRUE;                  // this process is interested
turn = process;                              // set flag
while(turn == process && interested[other] == TRUE); // wait
}
void leave_region(int process)
{
interested[process] = FALSE;                 // process leaves critical region
}
```

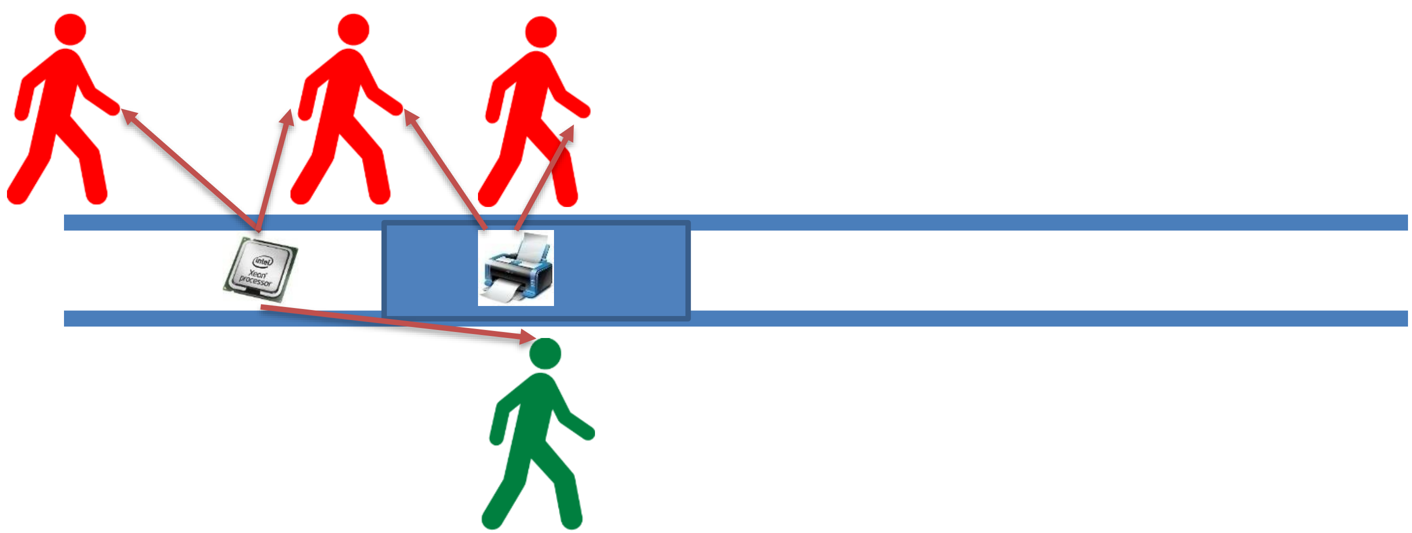


Priority inversion problem

- Priority inversion means the **execution of a high priority process/thread is blocked by a lower priority process/thread**.
- Consider a computer with two processes, H having high priority and L having low priority.
- The scheduling rules are such that H runs first then L will run.

Priority inversion problem

- At a certain moment, L is in critical region and H becomes ready to run (e.g. I/O operation complete).
- H now begins busy waiting and waits until L will exit from critical region.
- But H has highest priority than L so CPU is switched from L to H.
- Now L will never be scheduled (get CPU) until H is running so L will never get chance to leave the critical region so H loops forever. This situation is called priority inversion problem.

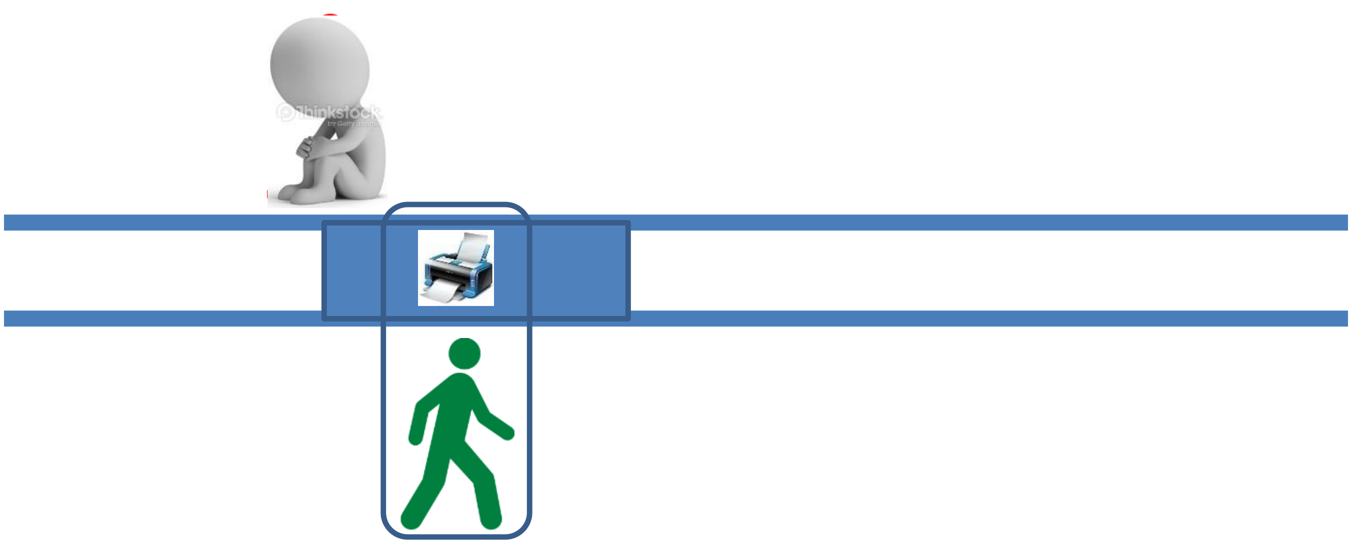


Mutual Exclusion with Busy Waiting

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

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**.

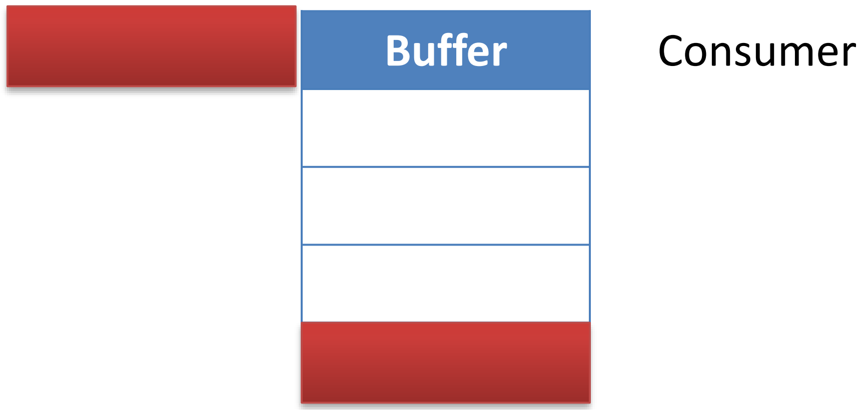


Sleep and Wakeup

- 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'.

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)

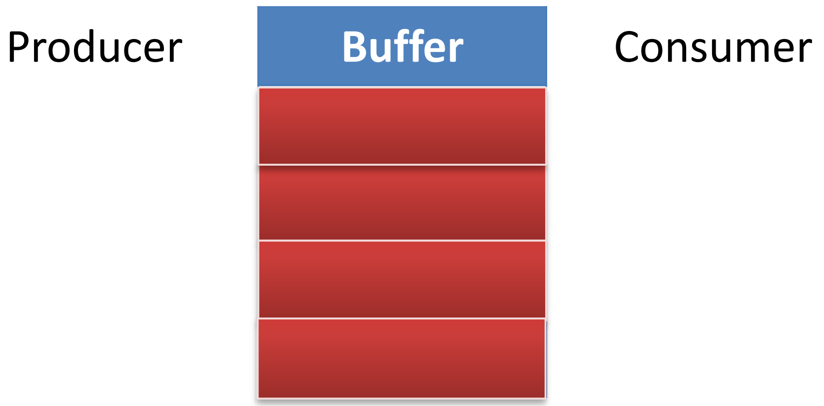


What Producer Consumer problem is?

- The problem is to make sure that the **producer won't try to add data** (information) into the buffer **if it is full** and **consumer won't try to remove data** (information) from the an **empty buffer**.
- Solution for producer:
  - **Producer either go to sleep** or discard data if the **buffer is full**.
  - **Once the consumer removes an item** from the buffer, it **notifies (wakeups) the producer** to put the data into buffer.
- Solution for consumer:
  - **Consumer can go to sleep** if the **buffer is empty**.
  - **Once the producer puts data into buffer**, it **notifies (wakeups) the consumer** to remove (use) data from buffer.

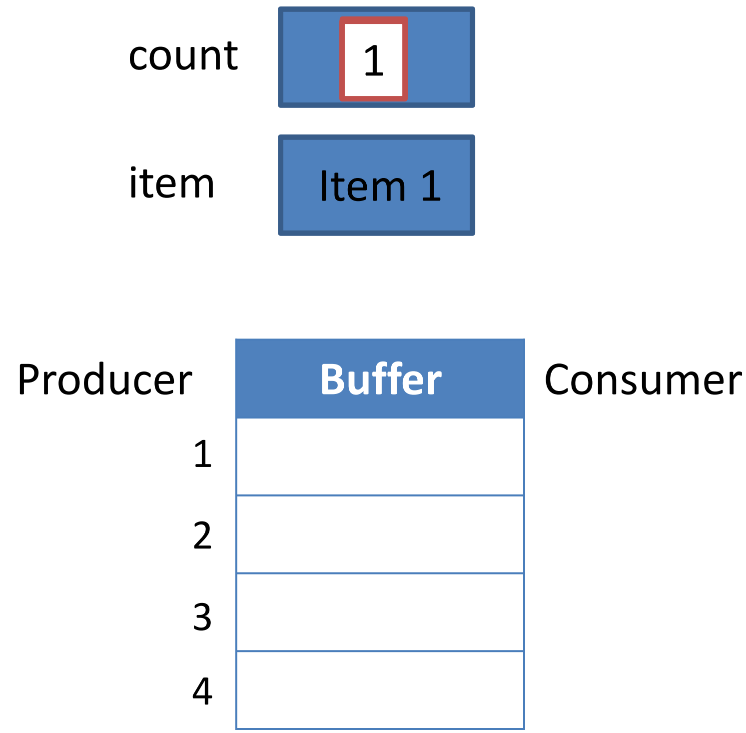
What Producer Consumer problem is?

- 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 √



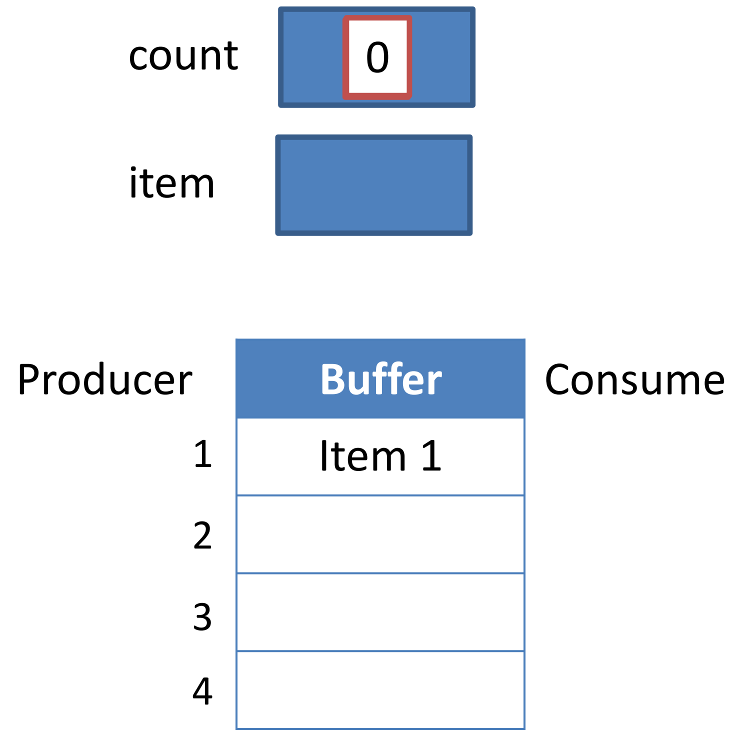
Producer Consumer problem using Sleep & Wakeup

```
#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???)**

Problem in Sleep & Wakeup

- The **consumer** has just **read** the variable **count**, noticed it's **zero** and is just about to move inside the if block.
- Just **before calling sleep**, the **consumer** is **suspended** and the **producer** is **resumed**.
- The **producer creates an item**, puts it into the buffer, and increases **count**.
- Because the **buffer was empty** prior to the last addition, the **producer tries to wake up the consumer**.

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

Problem in Sleep & Wakeup

- Unfortunately the **consumer wasn't yet sleeping**, and the **wakeup call is lost**.
- When the **consumer resumes**, it **goes to sleep** and will **never be awakened again**. This is because the consumer is only awakened by the producer when **count** is equal to 1.
- The **producer will loop until the buffer is full**, after which it will also go to sleep.
- **Finally, both the processes will sleep forever.** This solution therefore is unsatisfactory.

```
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);
        }
}
```

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:

1. **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.
2. **Counting semaphores** :-
    - Counting semaphore can have possible **values more than two**.

Semaphore (cont…)

- We want functions **insert _item** and **remove_item** such that the following hold:
  1. Mutually exclusive access to buffer: **At any time only one process should be executing** (either insert_item or remove_item).
  2. 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).
  3. 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).

Semaphore (cont…)

- We want functions **insert _item** and **remove_item** such that the following hold:

  4. No busy waiting.

  5. No producer starvation: A **process does not wait forever** at **insert_item()** provided the buffer repeatedly becomes full.

  6. No consumer starvation: A **process does not wait forever at remove_item()** provided the buffer repeatedly becomes empty.

Semaphores

Two operations on semaphores are defined.

1. **Down** Operation
   - The down operation on a semaphore **checks** to see if the **value is greater than 0**.
   - If so, it **decrements the value** and just continues.
   - If the **value is 0**, the process is **put to sleep** without completing the down for the moment.
   - **Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible atomic action**.
   - It is **guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked**.

Semaphores

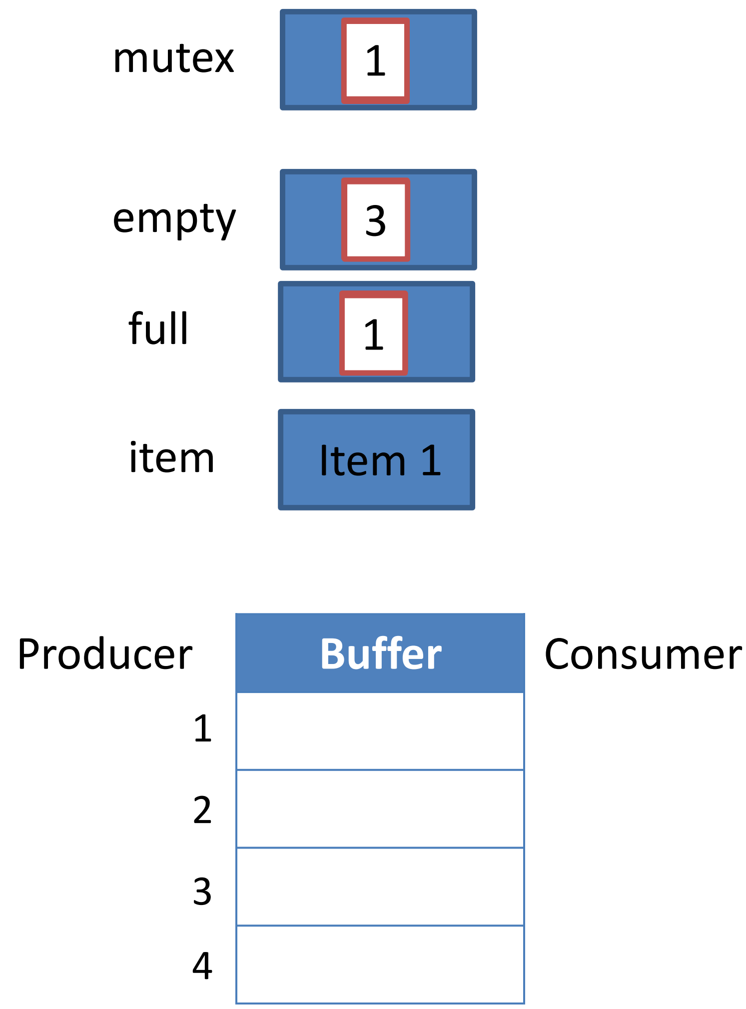
Two operations on semaphores are defined.

2. **Up** Operation

- The up operation **increments the value** of the semaphore addressed.
- If **one or more processes were sleeping** on that semaphore, unable to complete an earlier down operation, **one of them is chosen by the system (e.g., at random) and is allowed to complete its down**.
- The operation of **incrementing the semaphore and waking up one process is also indivisible**.
- **No process ever blocks doing an up, just as no process ever blocks doing a wakeup in the earlier model.**

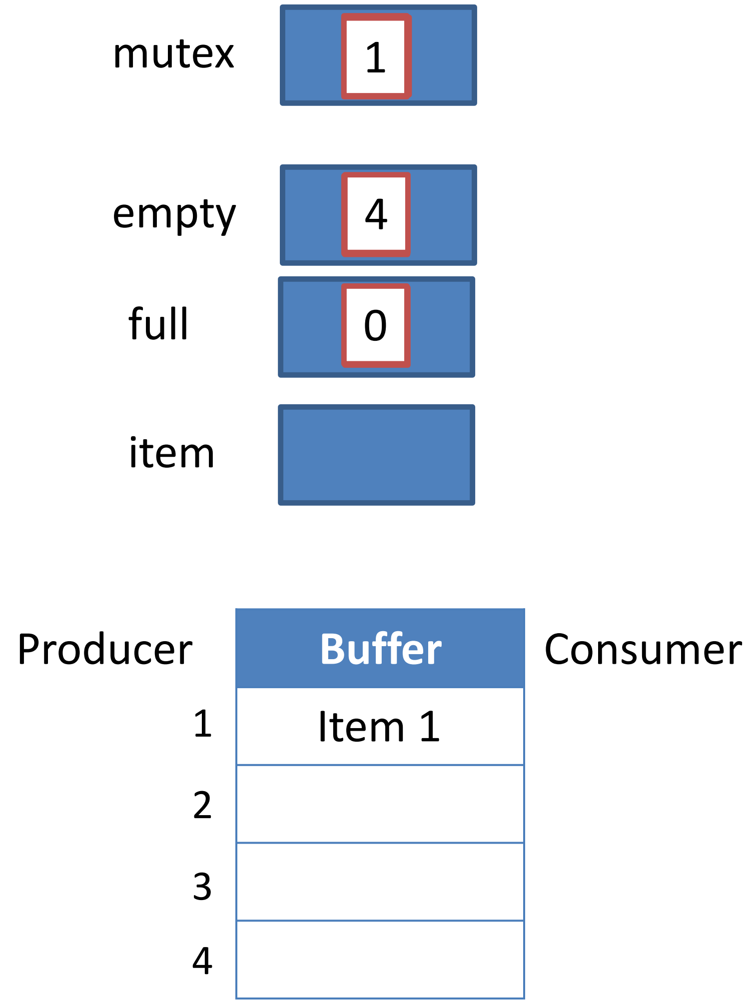
Producer Consumer problem using Semaphore

```
#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);
        }
}
```



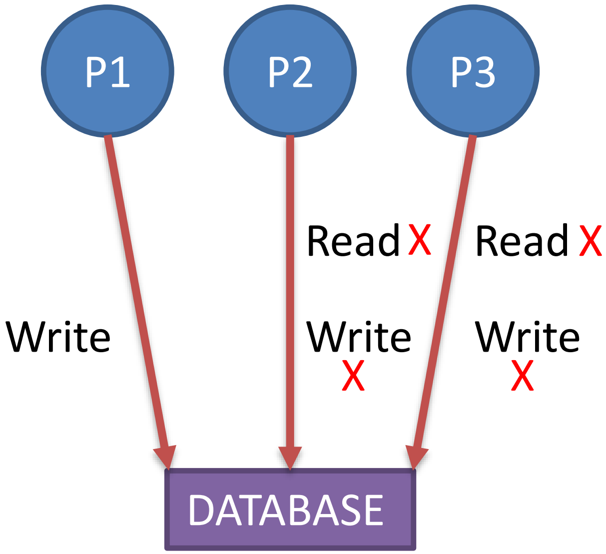
Producer Consumer problem using Semaphore

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



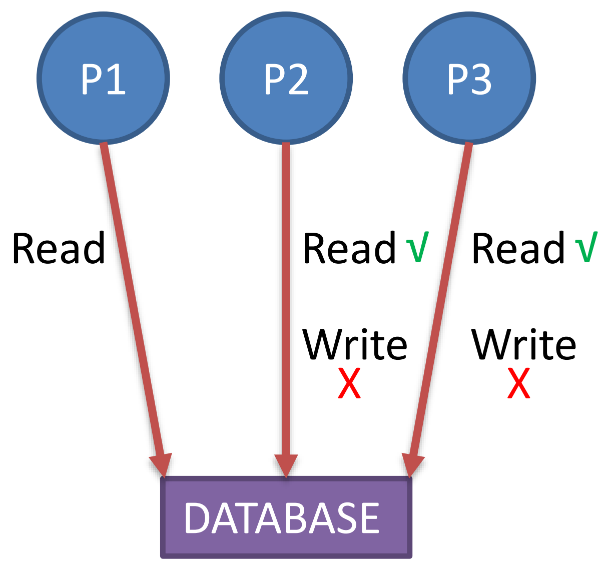
Readers Writer problem

- In the readers and writers problem, many competing processes are wishing to perform reading and writing operations in a database.
- It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers.



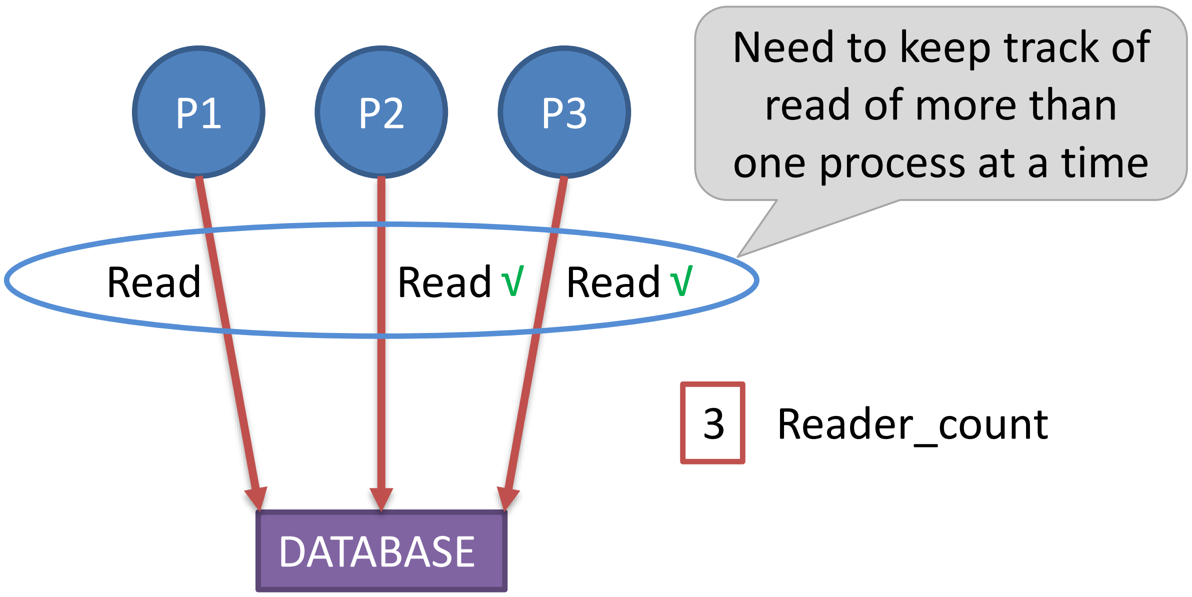
Readers Writer problem

- In the readers and writers problem, many competing processes are wishing to perform reading and writing operations in a database.
- It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers.



Readers Writer problem

- In the readers and writers problem, many competing processes are wishing to perform reading and writing operations in a database.
- It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers.



Readers Writer problem using Semaphore

```
typedef int semaphore;
semaphore mutex=1;          //control access to reader count
semaphore db=1;             //control access to database
int reader_count=0;         //number of processes reading database
```

Readers Writer problem using Semaphore

```
void Reader (void)
{     while (true){
      down(&mutex);                       //gain access to reader count
      reader_count=reader_count+1;   //increment reader counter
      if(reader_count==1)                 //if this is first process to read DB
            down(&db)                     //prevent writer process to access DB
      up(&mutex)                          //allow other process to access reader_count
      read_database();
      down(&mutex);                        //gain access to reader count
       reader_count=reader_count-1;    //decrement reader counter
      if(reader_count==0)                  //if this is last process to read DB
            up(&db)                       //leave the control of DB, allow writer process
      up(&mutex)                          //allow other process to access reader_count
      use_read_data();}                   //use data read from DB (non-critical)
}
```

Readers Writer problem using Semaphore

```
void Writer (void)
{         while (true){
          create_data();          //create data to enter into DB (non-critical)
          down(&db);              //gain access to DB
          write_db();             //write information to DB
          up(&db);}               //release exclusive access to DB
}
```

Mutex

- Mutex is the short form for '**Mutual Exclusion Object**'.
- A Mutex and the binary semaphore are essentially the same.
- Both Mutex and the binary semaphore can take values: 0 or 1.

```
mutex_lock:
        TSL REGISTER,MUTEX        |copy Mutex to register and set Mutex to 1
        CMP REGISTERS, #0         |was Mutex zero?
        JZE ok                    |if it was zero, Mutex was unlocked, so return
        CALL thread_yield         |Mutex is busy; schedule another thread
        JMP mutex_lock            |try again later
ok:     RET                       |return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0             |store a 0 in Mutex
        RET                       |return to caller
```

Monitor

- A **higher-level synchronization primitive**.
- A monitor is a **collection of procedures, variables, and data structures** that are all **grouped together in a special kind of module or package**.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

Monitor

- Monitors have an **important property for achieving mutual exclusion: only one process can be active in a monitor at any instant**.
- When a **process calls a monitor procedure**, the **first few instructions of the procedure will check to see if any other process is currently active within the monitor**.
- If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

Producer Consumer problem using Monitor

- The solution proposes condition variables, along with two operations on them, **wait and signal**.
- When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a wait on some condition variable, **full**.
- This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now.

Producer Consumer problem using Monitor

- This other process the consumer, can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on.
- To avoid having two active processes in the monitor at the same time a signal statement may appear only as the final statement in a monitor procedure.
- If a signal is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived.

Producer Consumer problem using Monitor

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert (item:integer);
        begin
                if count=N then wait (full);
                insert_item(item);
                count=count+1;
                if count=1 then signal (empty);
        end;
```

Producer Consumer problem using Monitor

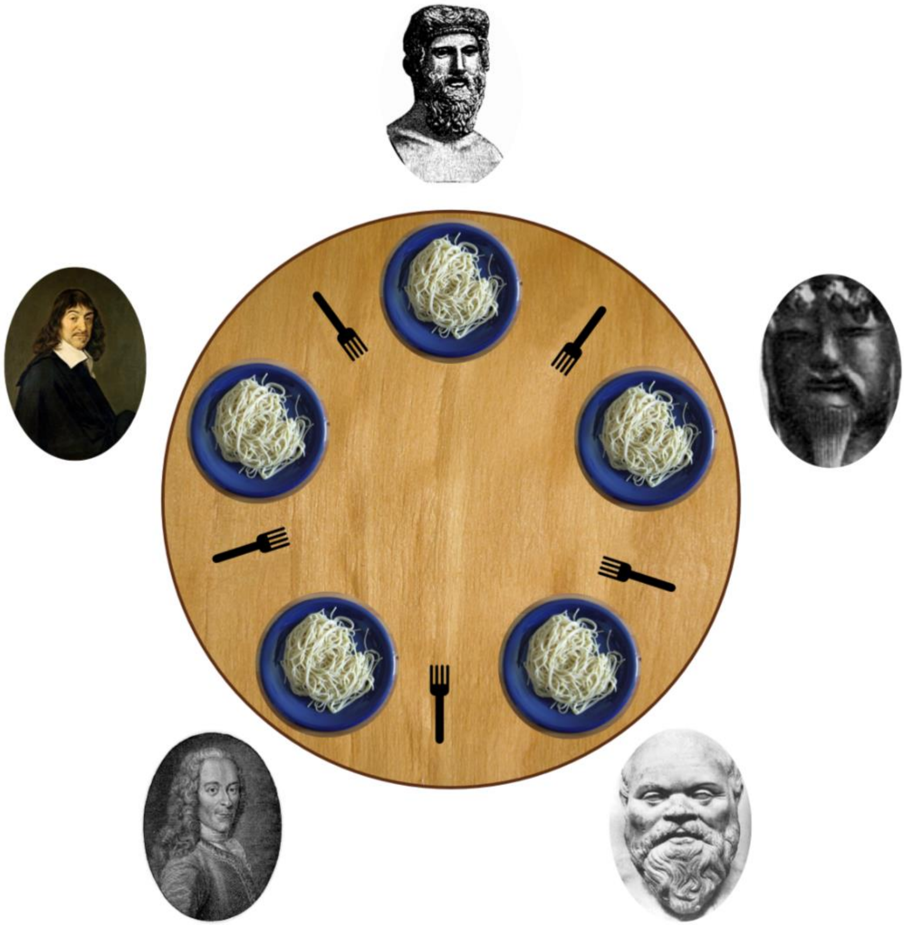
```
    function remove:integer;
    begin
         if count=0 then wait (empty);
        remove=remove_item;
        count=count-1;
        if count=N-1 then signal (full);
    end;
    count=0;
end monitor;
```

Producer Consumer problem using Monitor

```
procedure producer;
begin
        while true do
        begin
                item=produce_item;
                ProducerConsumer.insert(item);
        end;
end;

procedure consumer;
begin
        while true do
        begin
                item=ProducerConsumer.remove;
                Consume_insert(item);
        end;
end;
```

Dinning Philosopher Problem



- In this problem 5 philosophers sitting at a round table doing 2 things eating and thinking.
- While eating they are not thinking and while thinking they are not eating.
- Each philosopher has plates that is total of 5 plates.
- And there is a fork place between each pair of adjacent philosophers that is total of 5 forks.
- Each philosopher needs 2 forks to eat and each philosopher can only use the forks on his immediate left and immediate right.

Solution to Dinning Philosopher Problem

```
#define N 5                      //no. of philosophers
#define LEFT (i+N-1)%5           //no. of i's left neighbor
#define RIGHT (i+1)%5            //no. of i's right neighbor
#define THINKING 0               //Philosopher is thinking
#define HUNGRY 1                 //Philosopher is trying to get forks
#define EATING 2                 //Philosopher is eating
typedef int semaphore;           //semaphore is special kind of int
int state[N];                    //array to keep track of everyone's state
semaphore mutex=1;               //mutual exclusion for critical region
semaphore s[N];                  //one semaphore per philosopher
```

Solution to Dinning Philosopher Problem

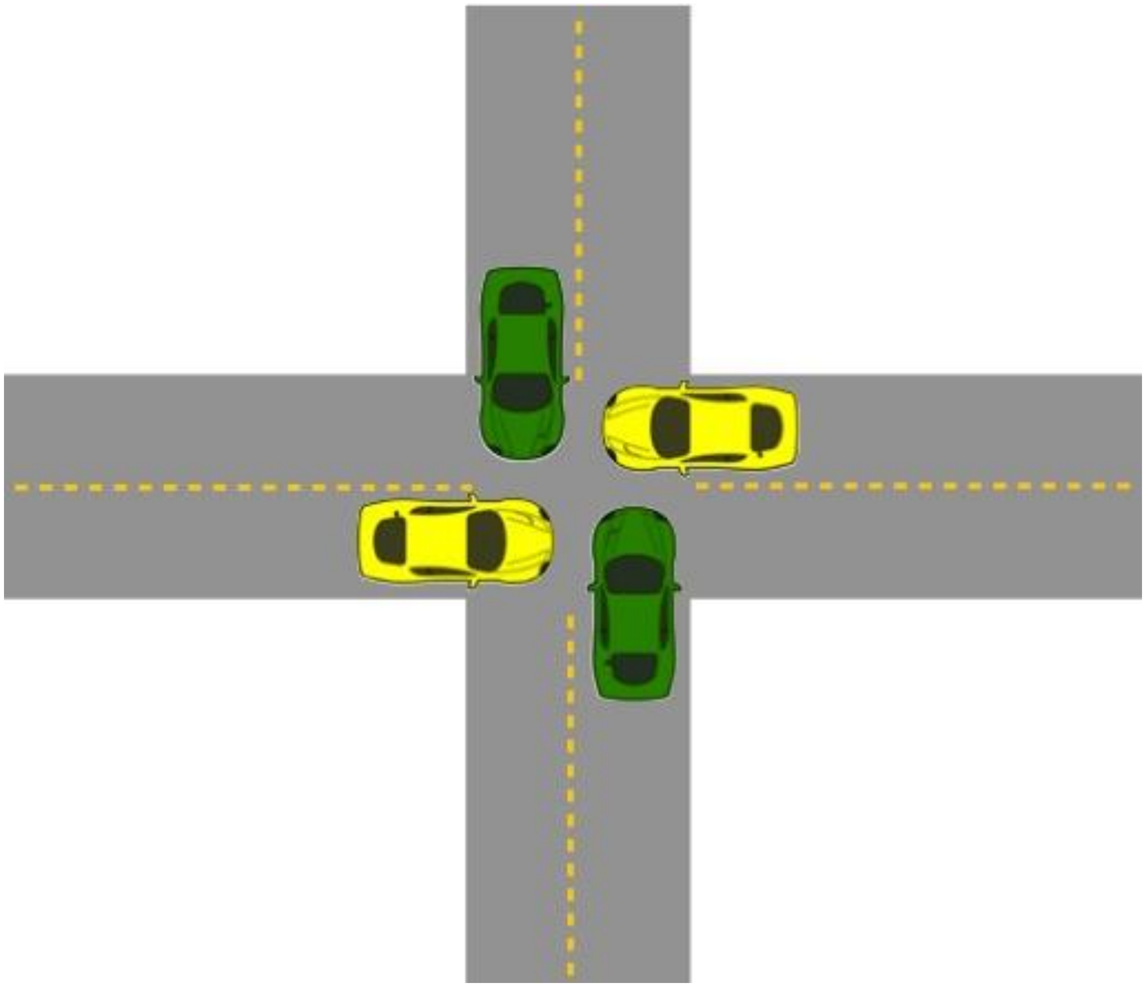
```
void take_forks (int i)            //i: philosopher no, from 0 to N-1
{
        down(&mutex);              //enter critical region
        state[i]=HUNGRY;           //record fact that philosopher i is hungry
        test(i);                   //try to acquire 2 forks
        up(&mutex);                //exit critical region
        down(&s[i]);               //block if forks were not acquired
}

void put_forks (int i)             //i: philosopher no, from 0 to N-1
{
        down(&mutex);              //enter critical region
        state[i]=THINKING;         //philosopher has finished eating
        test(LEFT);                //see if left neighbor can now eat
        test(RIGHT);               //see if right neighbor can now eat
        up(&mutex);                //exit critical region
}
```

Solution to Dinning Philosopher Problem

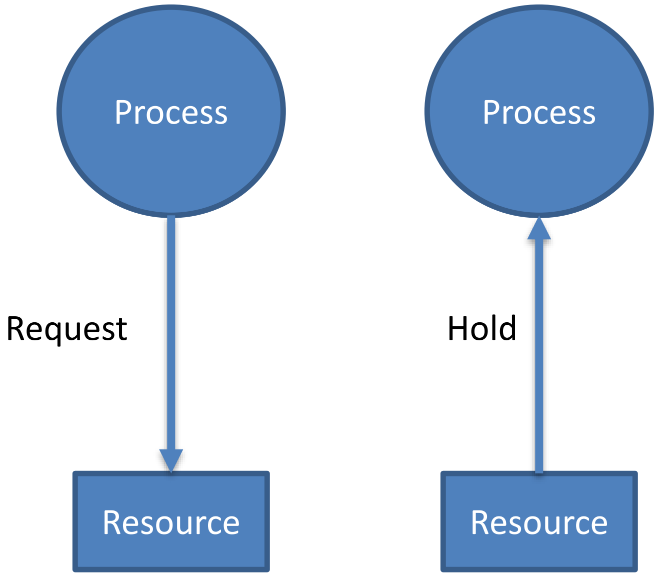
```
void test (i)                    //i: philosopher no, from 0 to N-1
{
        if (state[i]==HUNGRY &&
         state[LEFT]!=EATING &&
         state[RIGHT]!=EATING)
        {
                state[i]=EATING;
                up (&s[i]);
        }
}
```

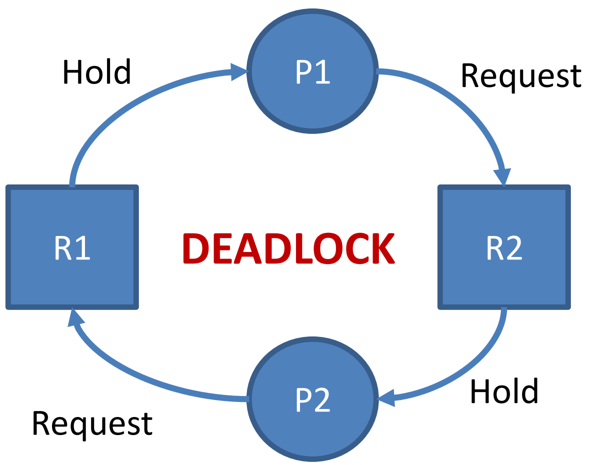
What is Deadlock?



What is Deadlock?

- A set of processes is deadlocked **if each process in the set is waiting for an event** that **only another process in the set can cause**.
- Deadlocks are a **set of blocked processes** each **holding a resource and waiting to acquire a resource held by another process**.





Preemptable and non-preemptable resource

- **Preemptable**:- Preemptive resources are those which **can be taken away from a process without causing any ill effects** to the process.
  - Example:- Memory.
- **Non-preemptable**:- Non-pre-emptive resources are those which **cannot be taken away from the process without causing any ill effects** to the process.
  - Example:- CD-ROM (CD recorder), Printer.

Deadlock v/s Starvation

Deadlock	Starvation
All processes keep waiting for each other to complete and none get executed.	High priority process keep executing and low priority process are blocked.
Resources are blocked by the process.	Resources are continuously utilized by the higher priority process.
Necessary conditions are mutual exclusion, hold and wait, no preemption, circular wait.	Priorities are assigned to the process.
Also known as circular wait.	Also known as lived lock.
It can be prevented by avoiding the necessary conditions for deadlock.	It can be prevented by Aging.

Conditions that lead to deadlock

1. Mutual exclusion
   - Each resource is either currently **assigned to exactly one process or is available**.
2. Hold and wait
   - Process currently holding resources granted earlier can **request more resources**.
3. No preemption
   - Previously granted resources **cannot be forcibly taken away** from process.
4. Circular wait
   - There must be a **circular chain of 2 or more processes**. Each process is waiting for resource that is held by next member of the chain.

- **All four of these conditions must be present for a deadlock to occur.**

Strategies for dealing with deadlock

1. Just **ignore** the problem.
2. **Detection** and **recovery**.
    - Let deadlocks occur, detect them and take action.
3. Dynamic **avoidance** by careful resource allocation.
4. **Prevention**, by structurally negating (killing) one of the four required conditions.