



## Outline

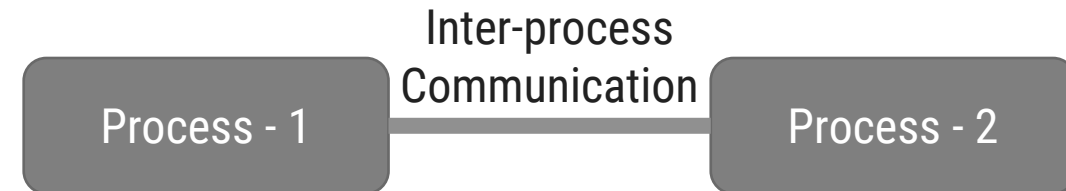
- IPC, Race Conditions, Critical Section, Mutual Exclusion
- Mutual exclusion with busy waiting
  - Disabling interrupts (Hardware approach)
  - Shared lock variable (Software approach)
  - Strict alteration (Software approach)
  - TSL (Test and Set Lock) instruction (Hardware approach)
  - Exchange instruction (Hardware approach)
  - Dekker's solution (Software approach)
  - Peterson's solution (Software approach)
- The Producer Consumer Problem
- Semaphores
- Classical IPC Problems
  - Readers & Writer Problem and Dining Philosopher Problem
- Monitor
- Mutex
- Pipes and Message Passing
- Barrier and Signal

# IPC, Race Conditions, Critical Section, Mutual Exclusion

- Section - 1

# 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**

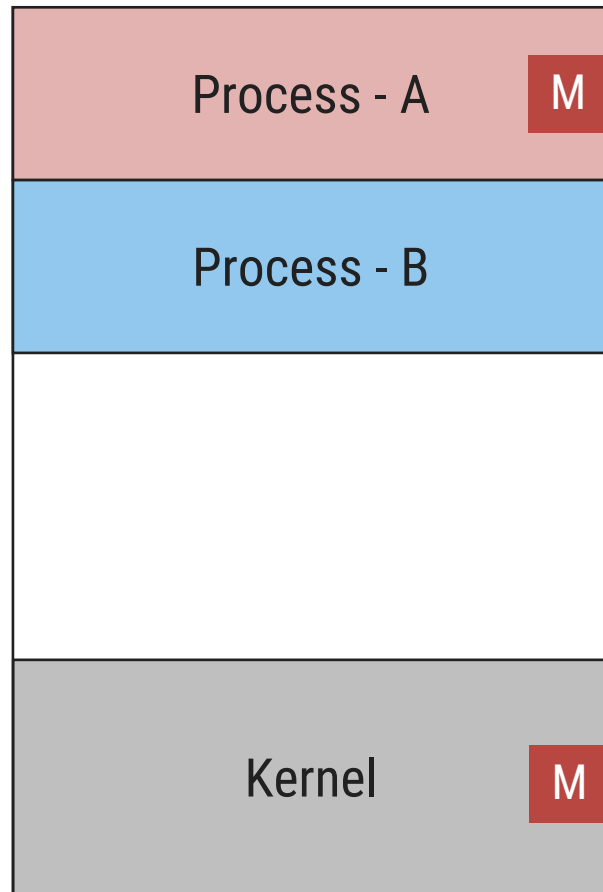
# Inter-process communication (IPC)

- 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)

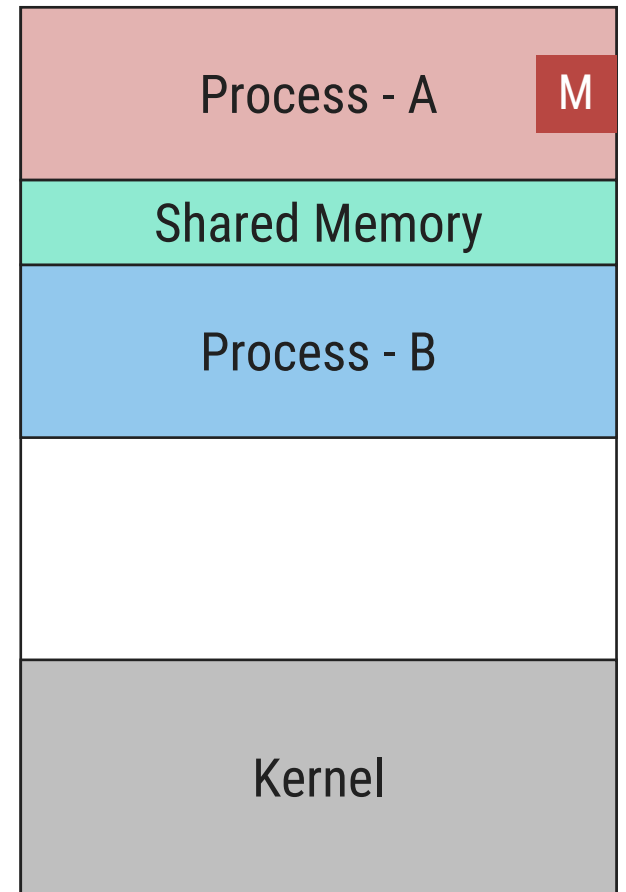
## □ Message Passing

- 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

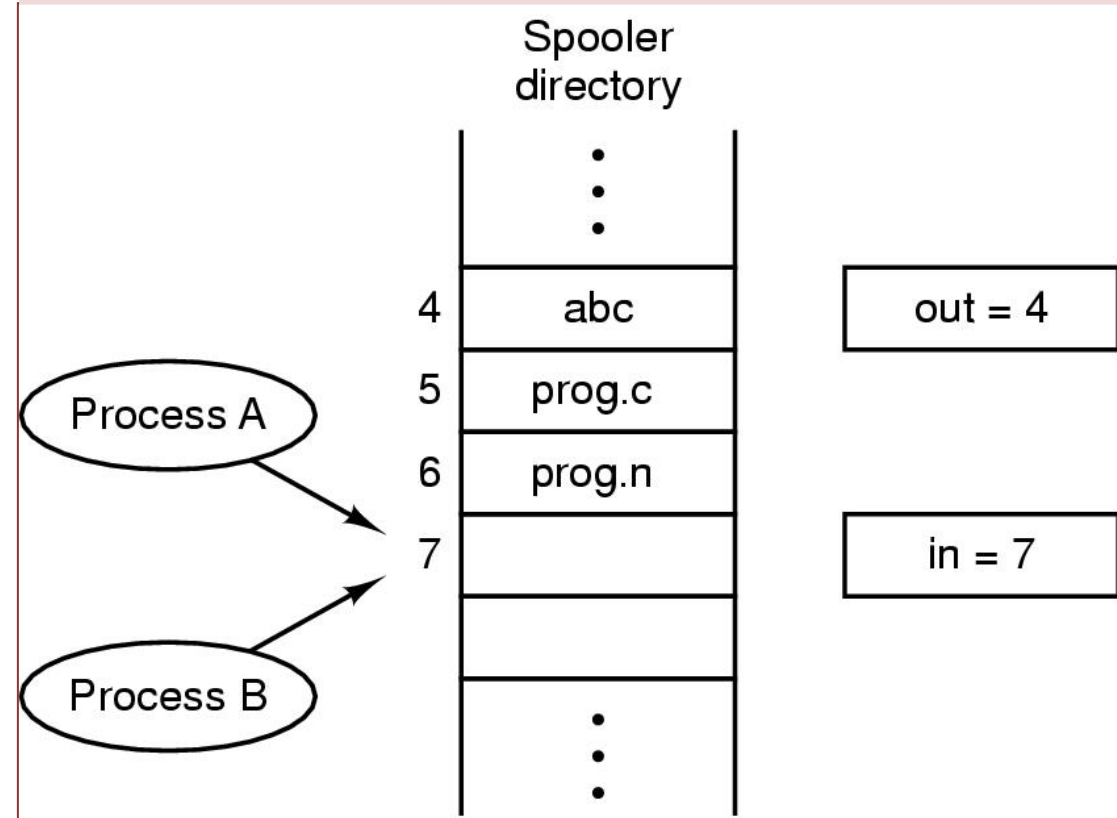


# Race Condition

- 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**.

Print spooler directory example :

Two processes want to access shared memory at the same time.



# Example of Race Condition

- Process A

next\_free\_slot = in

Write file name at slot (7)

next\_free\_slot += 1

in = next\_free\_slot (8)

## Context Switch

- Process B

next\_free\_slot = in

Write file name 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)

next\_free\_slot += 1

## Context Switch

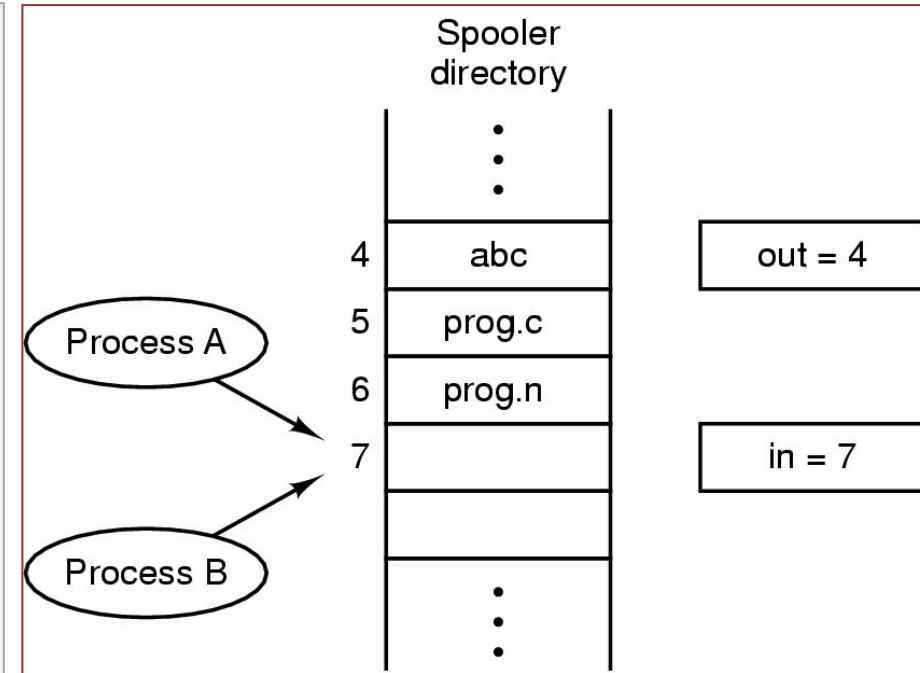
- Process A

Write file name at slot (7)

next\_free\_slot += 1

in = next\_free\_slot (8)

Race Condition

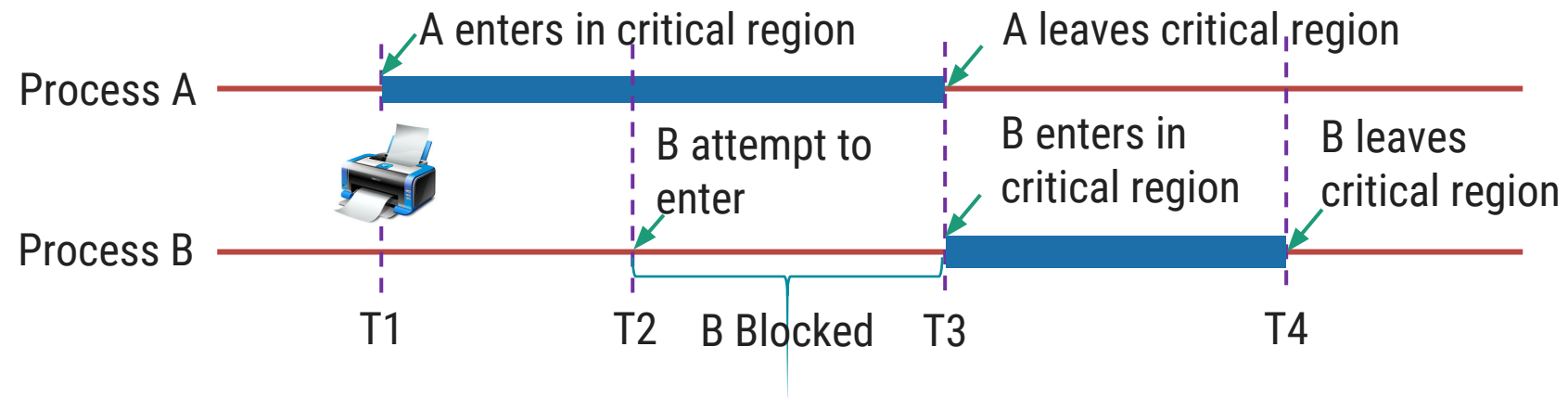




# Critical Section



**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).

# Mutual exclusion with busy waiting

- Section - 2

# Mutual exclusion with busy waiting

- Mechanisms for achieving mutual exclusion with busy waiting
  - Disabling interrupts ([Hardware approach](#))
  - Shared lock variable ([Software approach](#))
  - Strict alteration ([Software approach](#))
  - TSL (Test and Set Lock) instruction ([Hardware approach](#))
  - Exchange instruction ([Hardware approach](#))
  - Dekker's solution ([Software approach](#))
  - Peterson's solution ([Software approach](#))

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



# Disabling interrupts (Hardware approach)

- Section – 2.1

# Disabling interrupts (Hardware approach)

- while (true)

- {

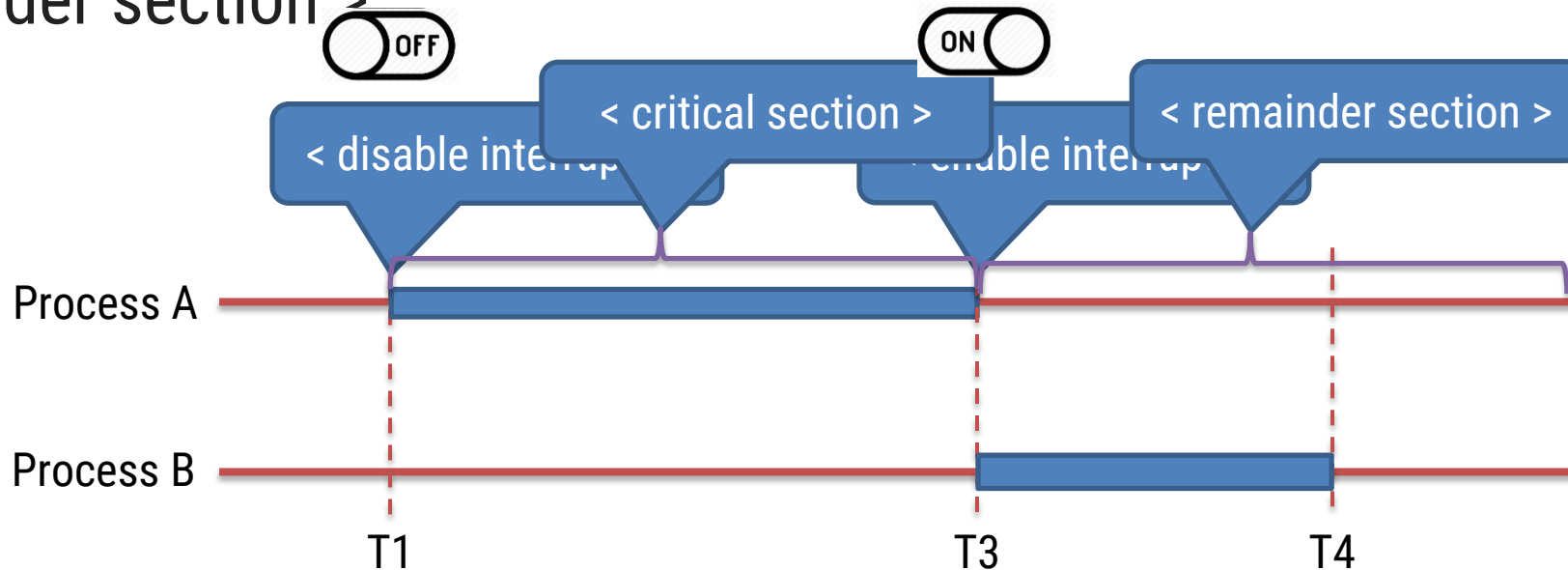
- < disable interrupts >;

- < 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)

- Section – 2.2

# Shared lock variable (Software approach)

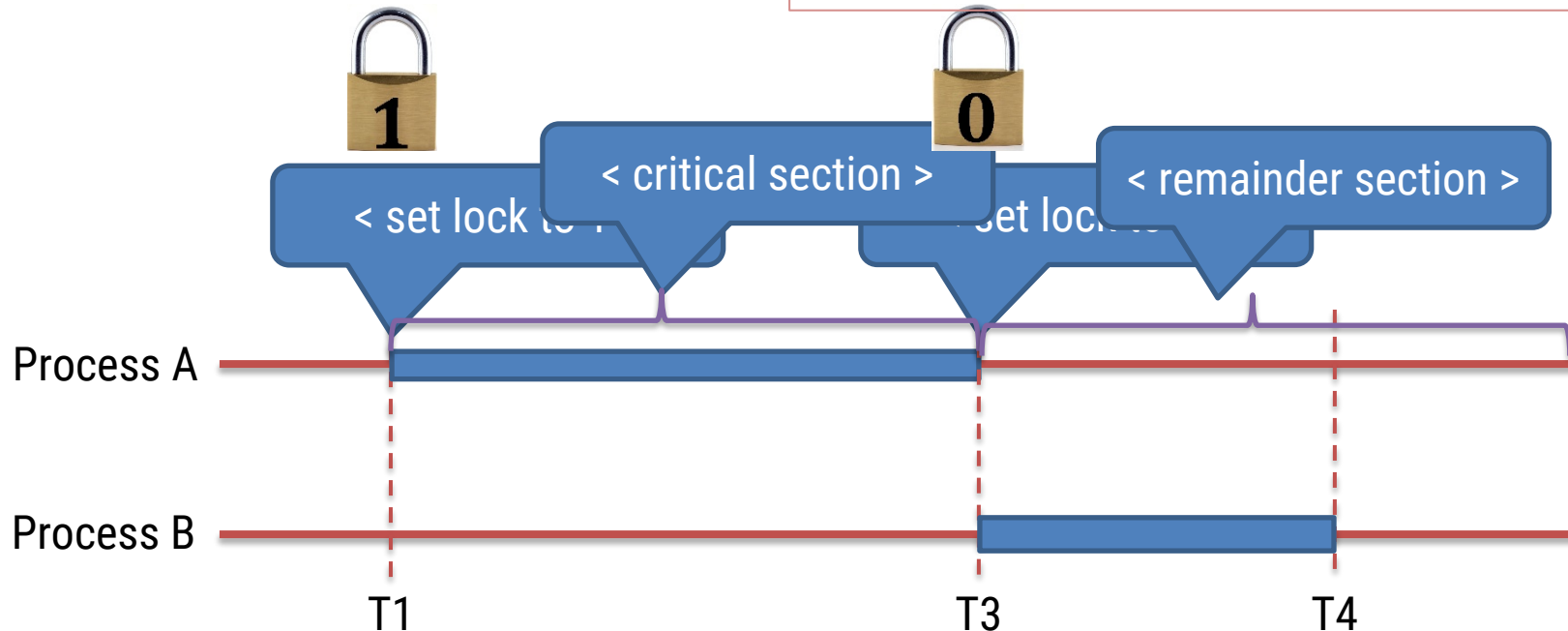
- A shared variable lock **having value 0 or 1**.
- Before entering into critical region a process checks a shared variable lock's value.
  - **If the value of lock is 0 then set it to 1** before entering the critical section and **enters into critical section** and **set it to 0** immediately after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

# 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.



# Strict alteration (Software approach)

- Section – 2.3

# Strict alteration (Software approach)

- **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.

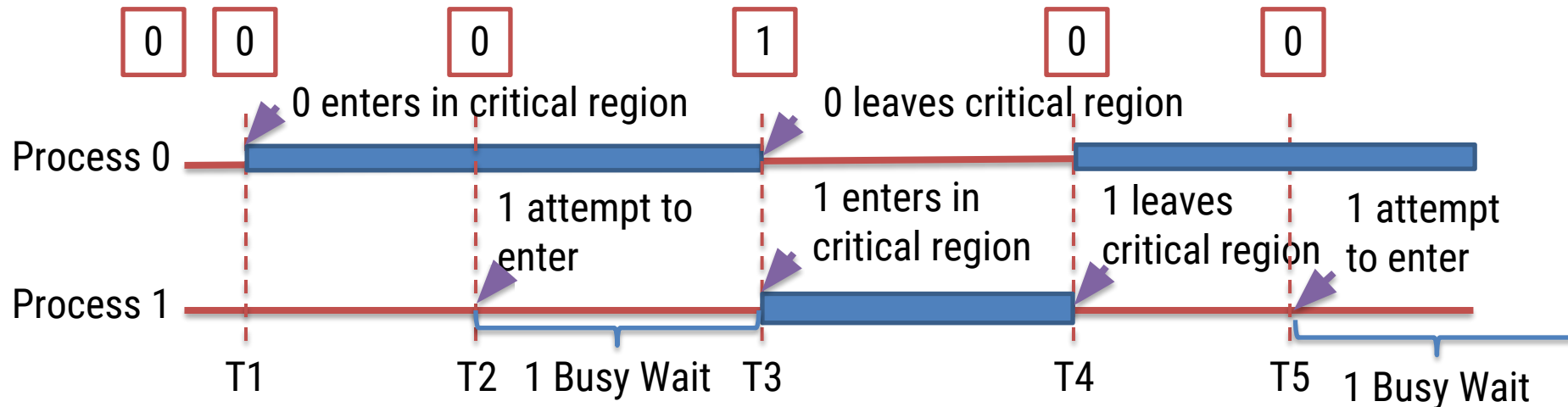
# Strict alteration (Software approach)

## Process 0

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

## Process 1

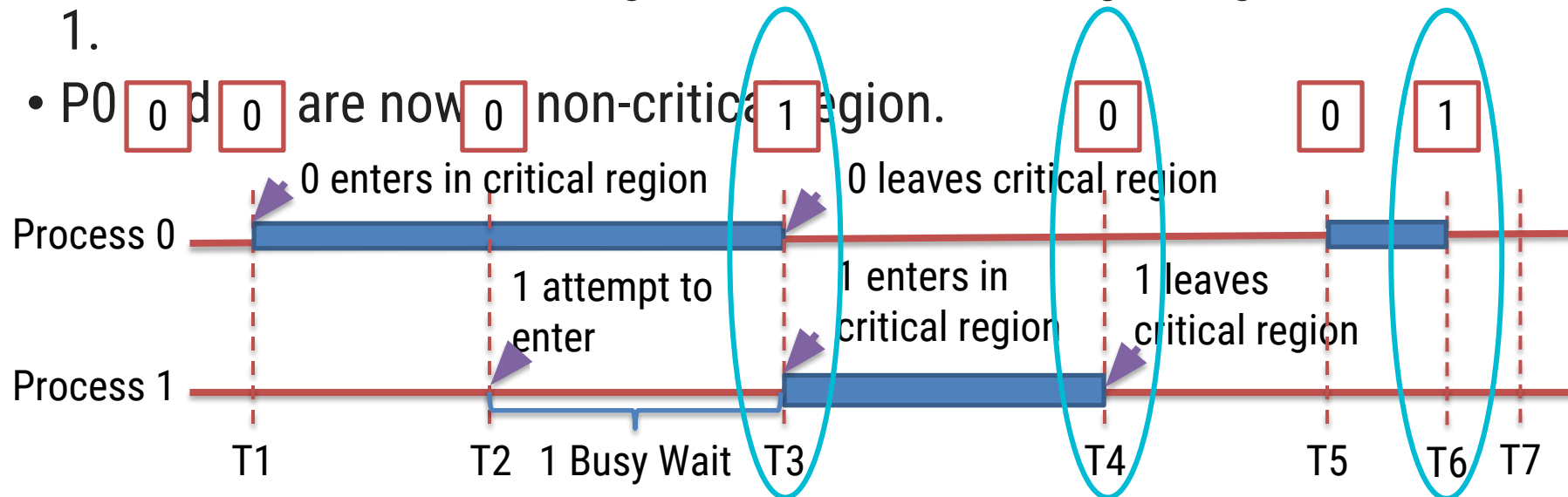
```
while (TRUE)
{
    while (turn != 1) /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```





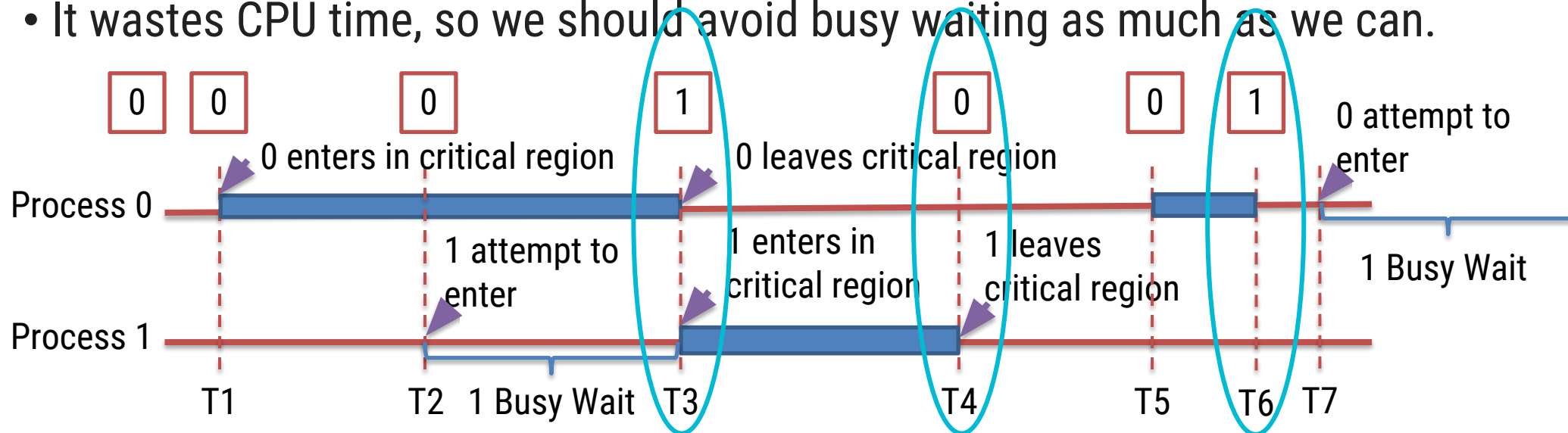
# Disadvantages of Strict alteration (Software approach)

- **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 0 and 0 are now 0 non-critical 1 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.



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

- Section – 2.4

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

- Algorithm

enter\_region: (Before entering its critical region, process calls enter\_region)

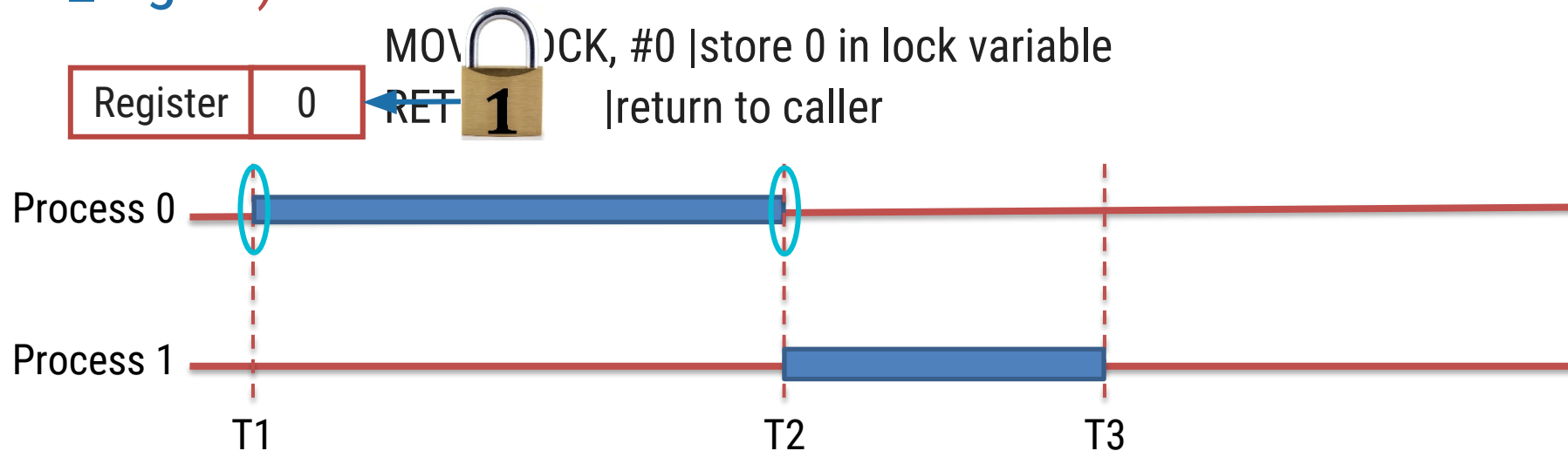
TSL REGISTER, LOCK | copy lock variable to register set lock to 1

CMP REGISTER, #0 | was register variable 0?

JNE enter\_region | if it was nonzero, lock was set, so loop

RET | return to caller: critical region entered

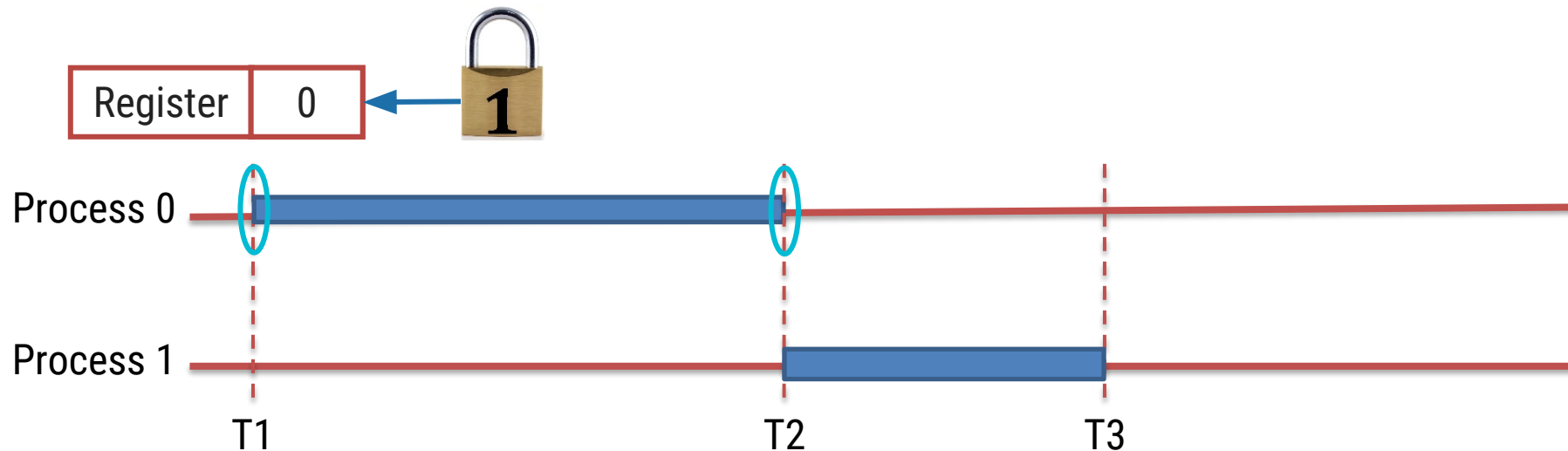
leave\_region: (When process wants to leave critical region, process calls leave\_region)



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

- 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 (Hardware approach)

- Section – 2.5

# Exchange instruction (Hardware approach)

- 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 register 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, process calls leave\_region)

MOVE LOCK, #0 |store 0 in lock variable

RET |return to caller



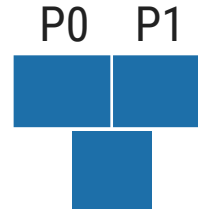
# Dekker's solution (Software approach)

- Section – 2.6

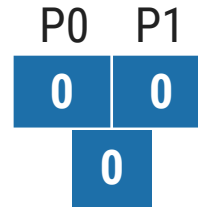
# Dekker's solution (Software approach)

- Algorithm variables

wants\_to\_enter [2] //array of 2 booleans  
turn //integer



wants\_to\_enter[0]  $\leftarrow$  false  
wants\_to\_enter[1]  $\leftarrow$  false  
turn  $\leftarrow$  0 // or 1



# Dekker's solution (Software approach)

## Process 0

```
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

P0 P1



## Process 1

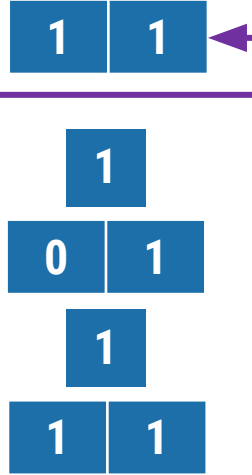
```
wants_to_enter[1] ← true
while (wants_to_enter[0])
  {if (turn == 0)
    {wants_to_enter[1] ← false
     while (turn == 0)
      {// busy wait}
     wants_to_enter[1] ← true
    }
  }
// critical section
...
turn ← 0
wants_to_enter[1] ← false
// remainder section
```

# Dekker's solution (Software approach)

## Process 0

```
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



## Process 1

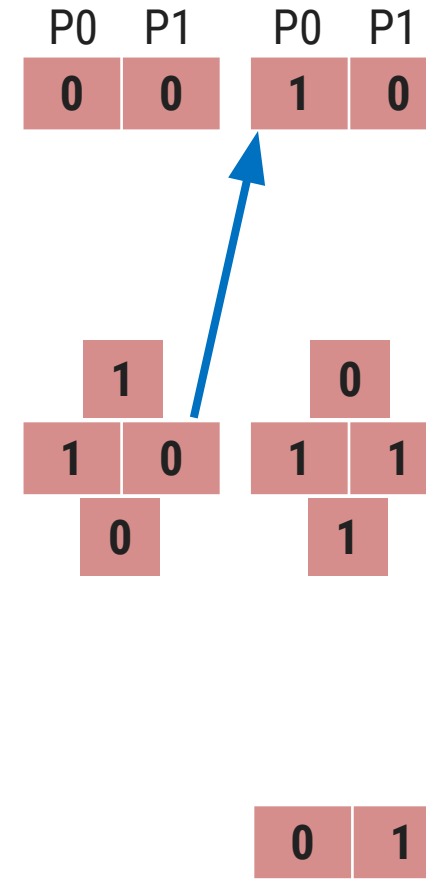
```
wants_to_enter[1] ← true
while (wants_to_enter[0])
{if (turn == 0)
  {wants_to_enter[1] ← false
   while (turn == 0)
    {// busy wait}
  wants_to_enter[1] ← true
}
}
// critical section
...
turn ← 0
wants_to_enter[1] ← false
// remainder section
```

# Peterson's solution (Software approach)

- Section – 2.7

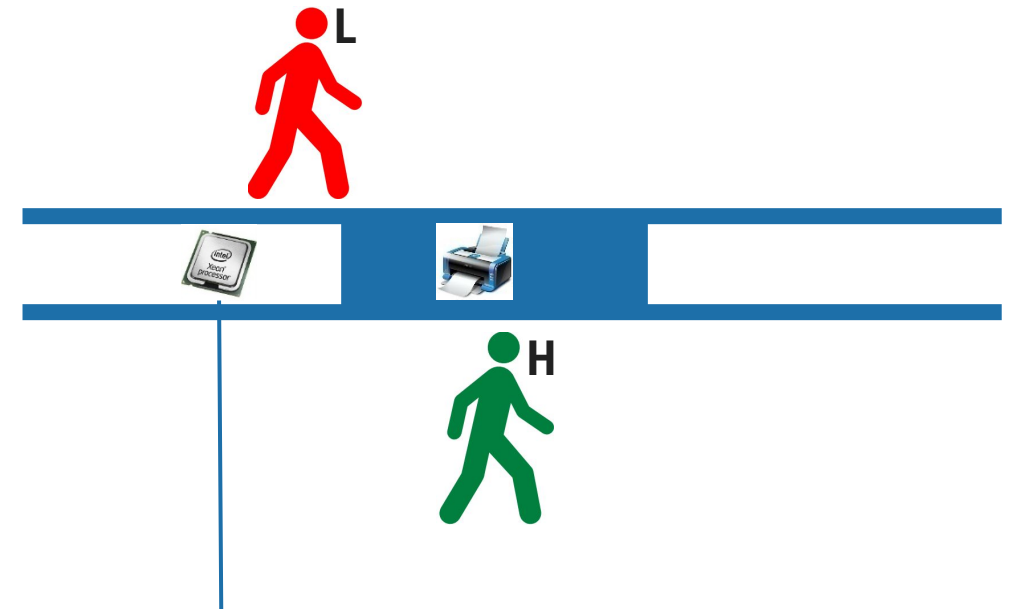
# Peterson's solution (Software approach)

```
#define FALSE 0
#define TRUE 1
#define N 2 //number of processes
int turn;    //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**.
- 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

- 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'.

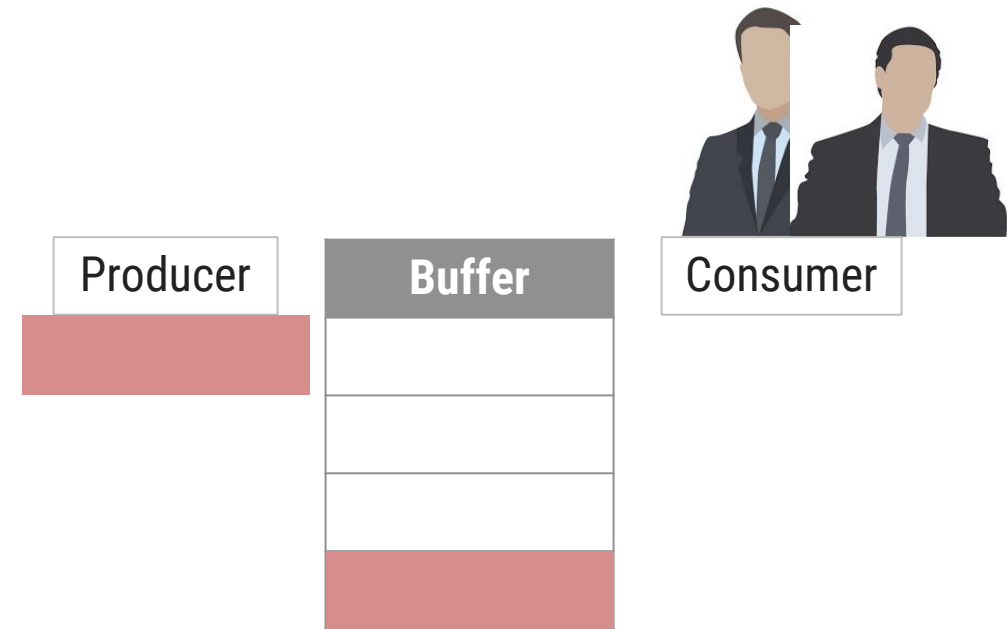


# The Producer Consumer Problem

- Section - 3

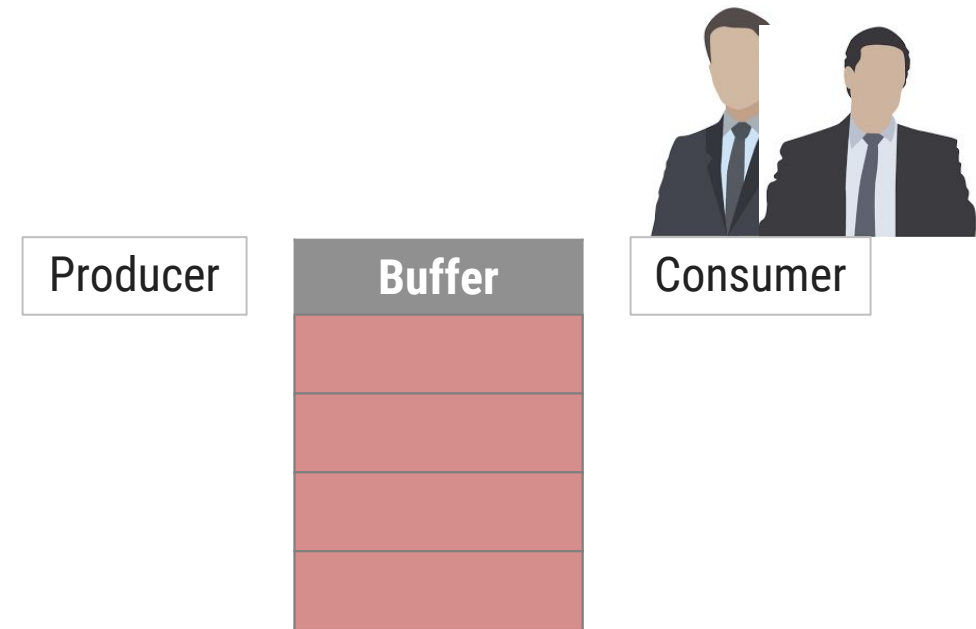
# The Producer Consumer Problem

- ❑ 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** ✓

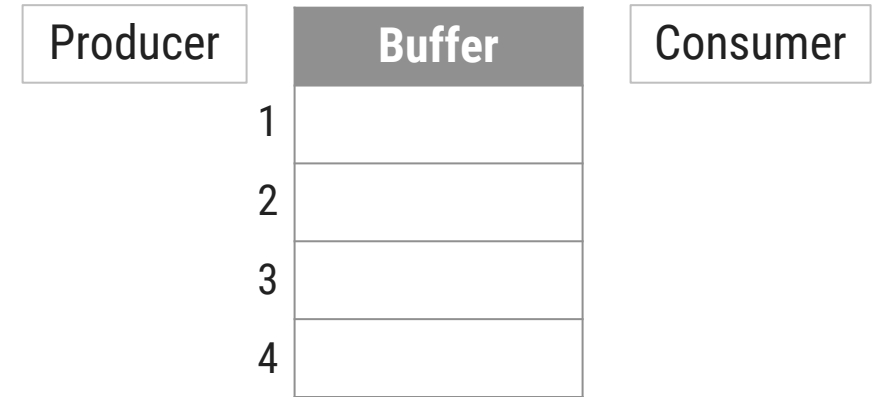


# Producer Consumer problem using Sleep & Wakeup

```
#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);
    }
}
```


count 1


item Item 1

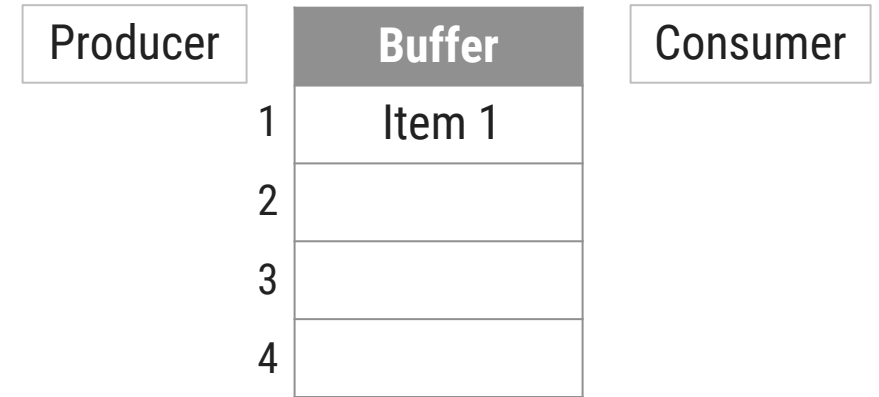


# Producer Consumer problem using Sleep & Wakeup

```
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);
    }
}
```

count 

item 



# Problem in Sleep & Wakeup

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

```
#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



count

1

item

Item 1

Buffer

1

2

3

4

Item 2

Item 3

Item 4

Wakeup  
Signal

```
void consumer (void)
```

```
{  int item;
```

```
  while (true)
```

```
  {  if (count==0) sleep();
```

```
      item=remove_item();
```

```
      count-1;
```

```
      if(count==N-1)
```

```
        wakeup(producer);
```

```
        consume_item(item);
```

```
  }
```

```
}
```

Consumer

Context  
Switching





# 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.
- 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.

# 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.

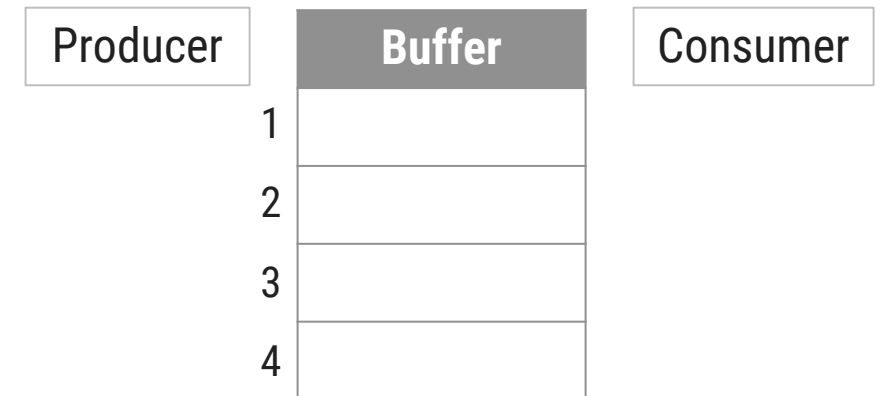
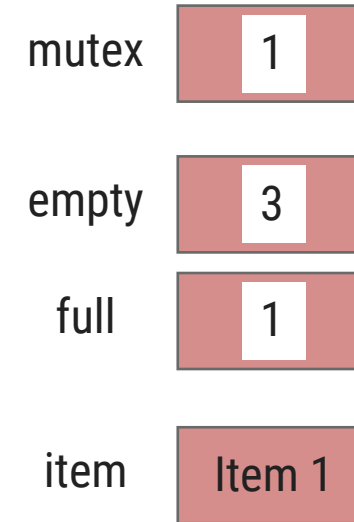
# Producer Consumer problem using Semaphore

```
#define N 4

typedef int semaphore;

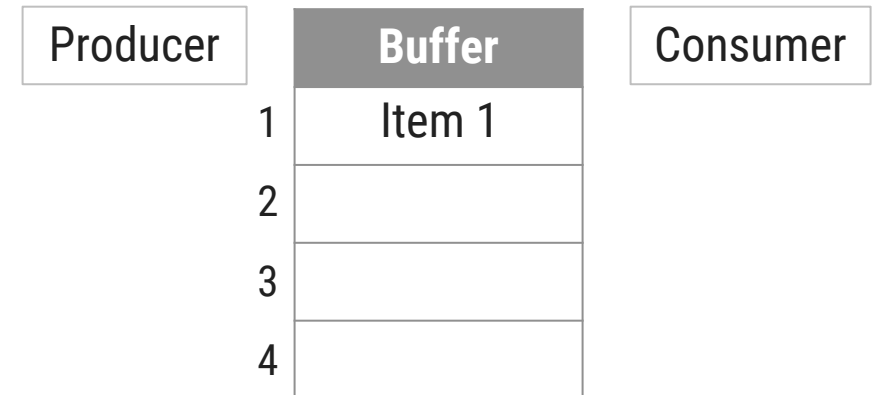
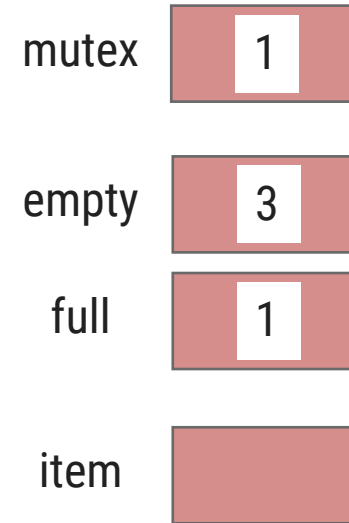
semaphore mutex=1;
semaphore empty=N;
semaphore full=0;

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);
    }
}
```



# Classical IPC Problems

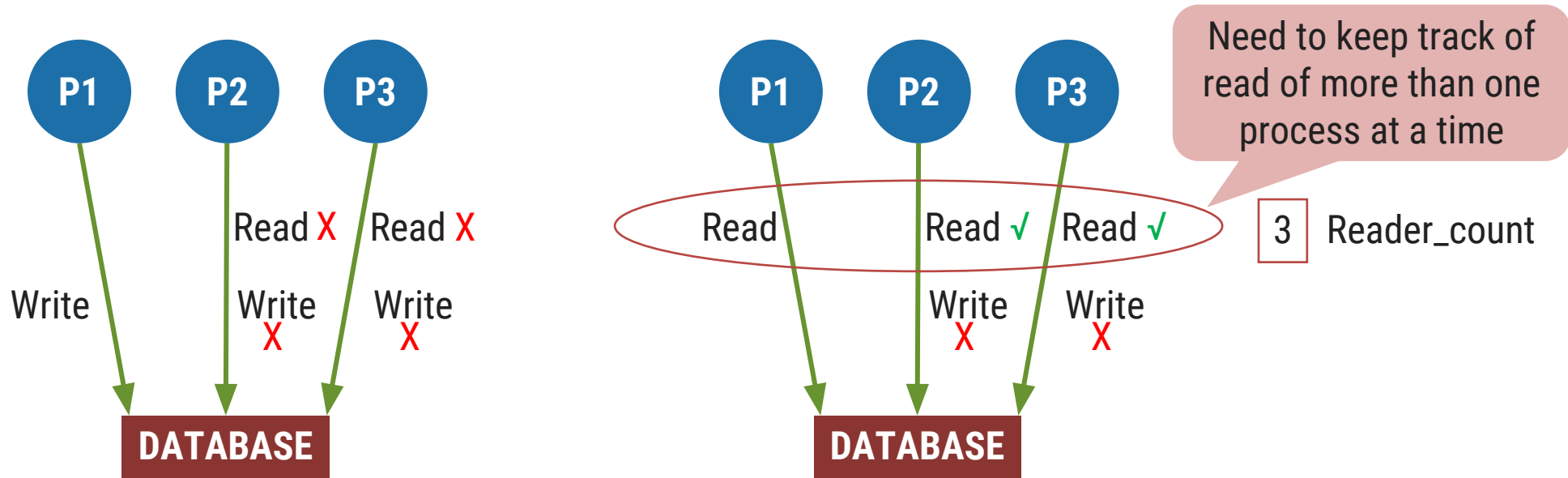
## Readers & Writer Problem

- Section – 5.1



# 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

```
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

```
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

```
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
}
```

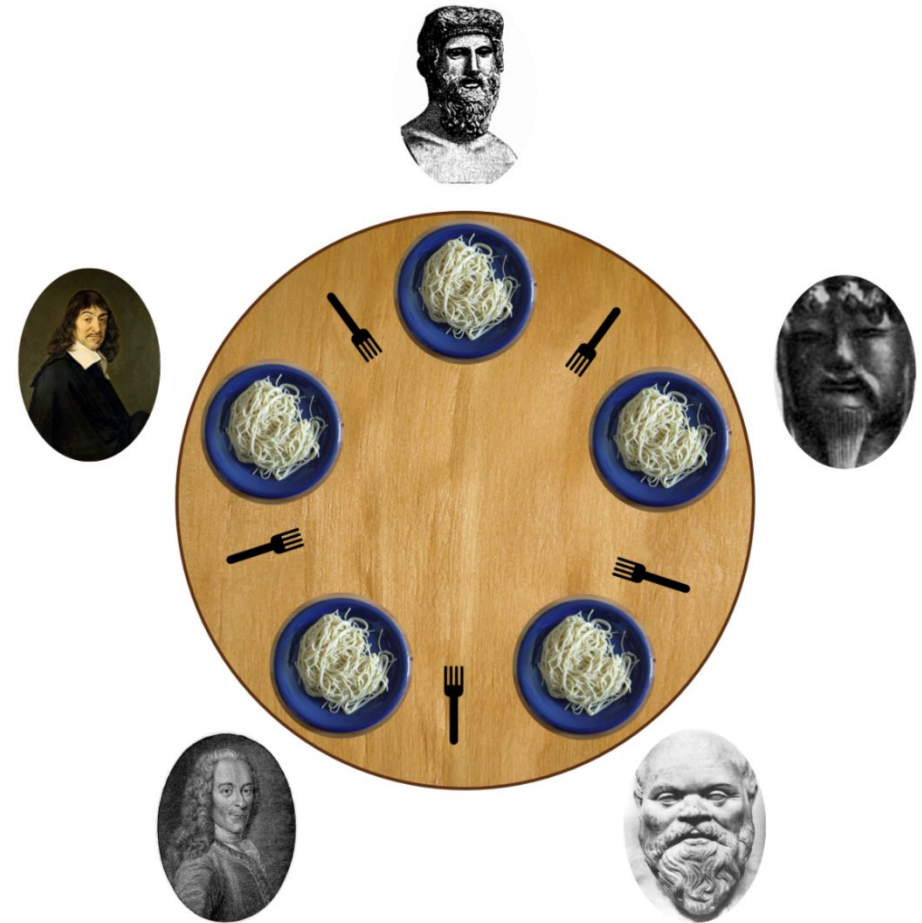
# Classical IPC Problems

## Dinning Philosopher Problem

- Section – 5.2

# 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.



# 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
```

# Dinning Philosopher Problem

```
void philosopher (int i)  //i: philosopher no, from 0 to N-1
{
    while (true)
    {
        think();          //philosopher is thinking
        take_forks(i);    //acquire two forks or block
        eat();             //eating spaghetti
        put_forks(i);     //put both forks back on table
    }
}
```

```
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 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]); }
}
```

```
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
}
```



# Monitor

- Section – 6

# 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.
- 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.
- 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** producer;

**begin**

**while** true **do**

**begin**

item=produce\_item;

ProducerConsumer.insert(item);

**end;**

**end;**

**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

```
procedure consumer;
```

```
begin
```

```
  while true do
```

```
    begin
```

```
      item=ProducerConsumer.remove;
```

```
      Consume_insert(item);
```

```
    end;
```

```
end;
```

```
  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;
```

# Mutex

- Section – 7

# 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 REGISTER, #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

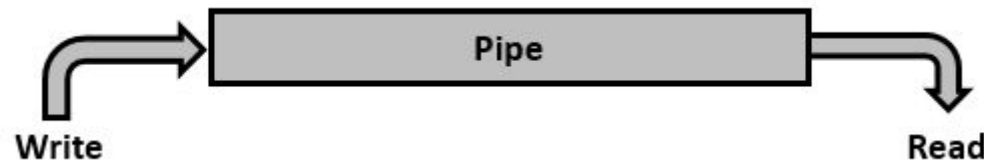
# Pipes and Message Passing

- Section – 8



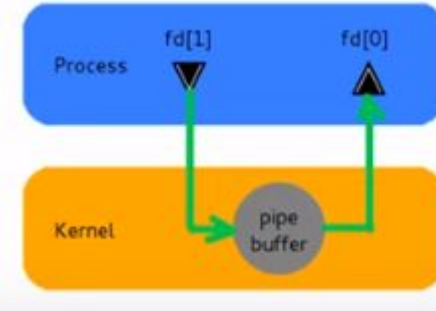
# Pipes

- Pipe is a **communication medium between two or more related or interrelated processes** usually between parent and child process.
- Communication is achieved by **one process write into the pipe and other process reads from the pipe**.
- It **performs one-way communication** only means we can use a pipe such that one process write into the pipe, and other process reads from the pipe.
- It opens a pipe, which is an **area of main memory that is treated as a “virtual file”**.
- It is **bounded buffer** means we can send only limited data through pipe.



# Pipes

- Accessed by two associated file descriptors
  - fd[0] for reading from pipe
  - fd[1] for writing into the pipe



```
#include<unistd.h>
int pipe(int pidedes[2]);
```

- This system call would **create a pipe for one-way communication** i.e., it **creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.**
- Descriptor **pipedes[0]** is for reading and **pipedes[1]** is for writing. **Whatever is written into pipedes[1] can be read from pipedes[0].**
- This call would **return zero on success and -1 in case of failure.**

# Pipes

```
#include<unistd.h>
```

```
#include<fcntl.h>
```

```
#include<stdio.h>
```

Header files

```
char *message = "Hey how are you"    /* Message created */
```

```
main ()
```

```
{
```

```
    char buffer[1024];    /* Size of buffer is 1024 */
```

```
    int fd[2];            /* Created array of size 2 for 2 process (file descriptor) */
```

```
    pipe(fd);             /* Created pipe; passed array (fd) in pipe system call */
```

```
    if (fork() != 0)      /* Parent code */
```

```
        { write (fd[1], message, strlen (message) + 1); }
```

```
    else                  /* Child code */
```

```
        { read (fd[0], buffer, 1024);
```

```
            printf ("Received from Parent %s\n", buffer); }
```

```
}
```

# Pipes

```
#include<unistd.h>
#include<fcntl.h>
#include<stdio.h>
char *message = "Hey how are you"
main ()
{
char buffer[1024];
int fd[2];
pipe(fd);
if ( 3000 != 0) /* Parent code */
{ write (fd[1], message,
        strlen (message) + 1); }
else /* Child code */
{ read (fd[0], buffer, 1024);
  printf ("Received from Parent %s\n",
        buffer); }
}
```

Parent process

```
#include<unistd.h>
#include<fcntl.h>
#include<stdio.h>
char *message = "Hey how are you"
main ()
{
char buffer[1024];
int fd[2];
pipe(fd);
if ( 0 != 0) /* Parent code */
{ write (fd[1], message,
        strlen (message) + 1); }
else /* Child code */
{ read (fd[0], buffer, 1024);
  printf ("Received from Parent %s\n",
        buffer); }
}
```

Child process

pid = 3000

# Message Passing

- This method will use **two primitives**

1. **Send**: It is used to **send message**.

- Send (destination, &message)
- In above syntax **destination** is the process to which sender want to send message and **message** is what the sender wants to send.

2. **Receive**: It is used to receive message.

- Receive (source, &message)
- In above syntax **source** is the process that has send message and **message** is what the sender has sent.

# Producer Consumer problem using message passing

## Producer

```
#define N 100 //number of slots in buffer
void producer (void)
{
    int item;
    message m; //message buffer
    while (true)
    {
        item=produce_item(); //generate something to put in buffer
        receive(consumer, &m); //wait for an empty to arrive
        build_message(&m, item); //construct a message to send
        send(consumer, &m); //send item to consumer
    }
}
```

## Consumer

```
void consumer (void)
{
    int item, i;
    message m;
    for (i=0; i<N; i++) send (producer, &m); //send N empties
    while (true)
    {
        receive (producer, &m); //get message containing item
        item=extract_item(&m); //extract item from message
        send (producer, &m); //send back empty reply
        consume_item (item); //do something with the item
    }
}
```

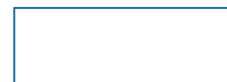


Producer

Message 1

Message 2

Message 3



Consumer

# Producer Consumer problem using message passing

- In this problem, an **actual buffer does not exist**. Instead, the **producer and consumer pass messages** to each other.
- These **messages can contain the items** which, in the previous examples, were placed in a buffer.
- In this example, a buffer size of three has been chosen.
- The **consumer begins the process by sending four empty messages to the producer**.
- The **producer creates a new item for each empty message it receives, then it goes to sleep**.
- The **producer will not create a new item unless it first receives an empty message**.
- The **consumer waits for messages from the producer which contain the items**.
- **Once consumer consumes an item, it sends an empty message to the producer**.
- Again, there is no real buffer, only a buffer size which dictates the number of items allowed to be produced.

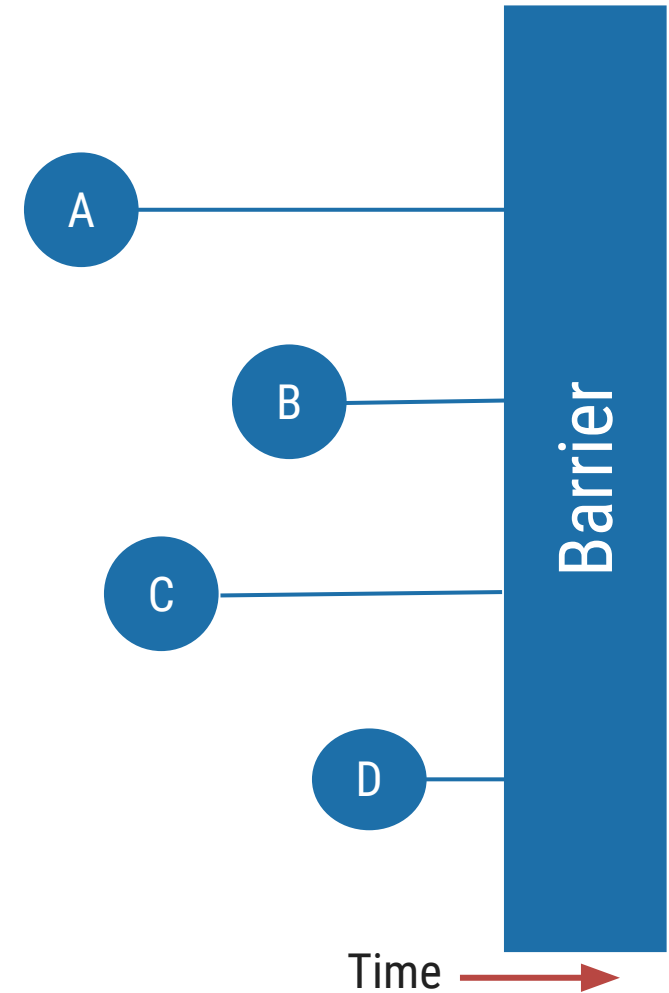
# Barrier and Signal

- Section – 9



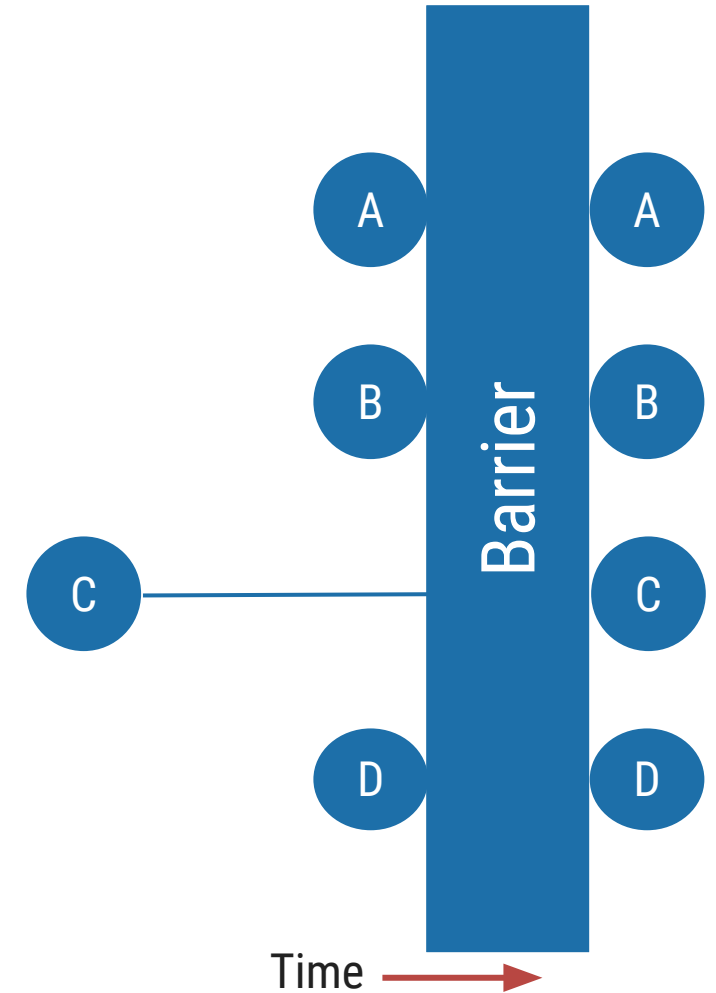
# Barrier

- In this mechanism **some application are divided into phases** and have that rule that **no process may proceed into the next phase until all process completed** in this phase and are **ready to proceed to the next phase**.
- This behavior is achieved by placing barrier at the end of each phase.
- When a **process reaches the barrier**, it is **blocked until all processes have reach at the barrier**.
- There are four processes with a barrier.



# Barrier

- Processes **A, B and D are completed** and **process C is still not completed** so **until process C will complete, process A, B and D will not start in next phase.**
- Once **process C completes** all the process start **in next phase.**



# Signal

- Signals are **software interrupts sent to a program to indicate that an important event has occurred.**
- When a signal is delivered to a process, the **process will stop what its doing, either handle or ignore the signal.**
- Signals are **similar to interrupts**, the difference being that **interrupts are mediated by the processor and handled by the kernel** while **signals are mediated by the kernel (possibly via system calls) and handled by processes.**
- The kernel may pass an interrupt as a signal to the process that caused it (examples are SIGSEGV, SIGBUS, SIGILL and SIGFPE).

# Questions

1. Define following Terms: Mutual Exclusion, Critical Section, Race Condition
2. What is Semaphore? Give the implementation of Readers-Writers Problem using Semaphore
3. What is Semaphore? Give the implementation of Bounded Buffer Producer Consumer Problem using Semaphore.
4. Explain Dining philosopher solution using Semaphore.
5. What Critical section Problem and list the requirements to solve it. Write Peterson's Solution for the same.
6. Write short note on message passing.
7. Explain monitor with algorithm.