Inter-Process Communication (IPC)

Marwadi University

Department of Computer Engineering

Unit - 3
Concurrency Control(IPC)

Operating System - 01CE1401

Dr. Krunal Vaghela

Topics Covered

Process synchronization

Critical- section problem.

Classic problems of Synchronization

Software Solutions for synchronization problem.

Hardware Solutions for synchronization problem.

Synchronization and their applications.

Interprocess Communication: Introduction

Processes in system can be independent or cooperating.

1. Independent processes: cannot affect or be affected by the execution of another process.
2. Cooperating processes: can affect or be affected by the execution of another process.

Interprocess Communication: Introduction

Interprocess 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 the transferring of data from one process to another.

Processes may be running on one or more computers connected by a network.

IPC may also be referred to as **inter-thread communication and inter-application communication.**

IPC, why is it there?

There are several reasons for providing an environment that allows process cooperation:

Information sharing

Computational speedup

Modularity

Convenience

Issues of process cooperation

Data corruption

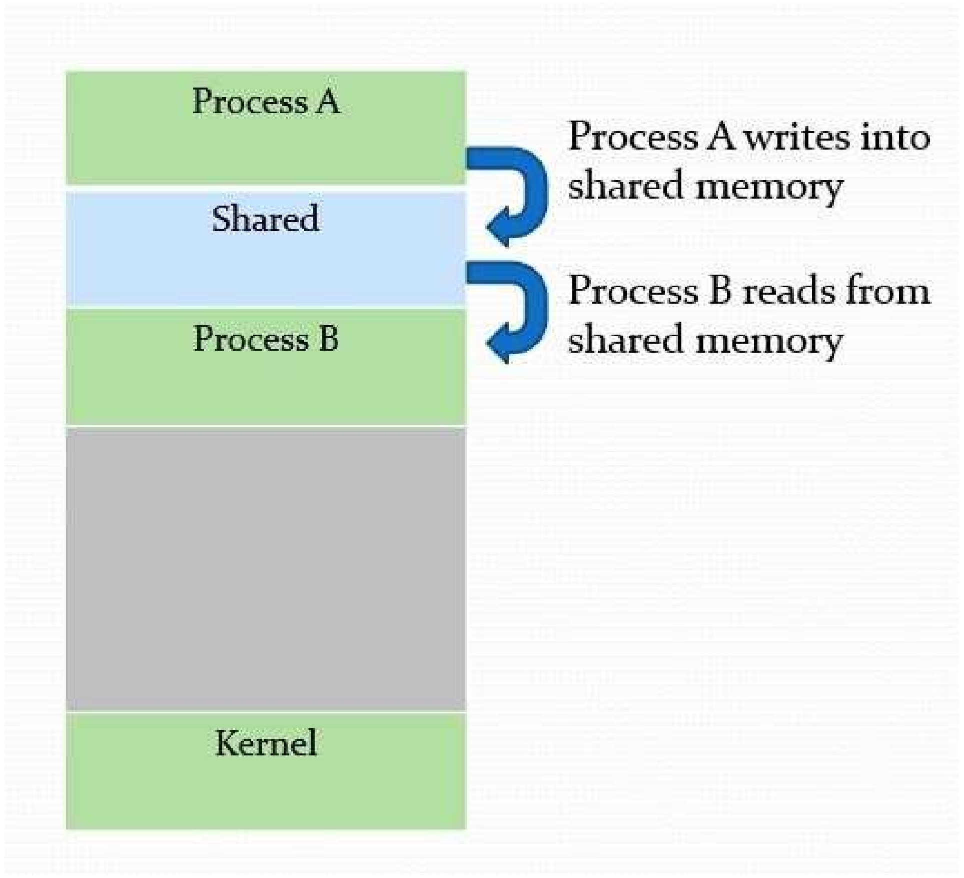
Deadlock

Increased complexity

Requires processes to synchronize their processing

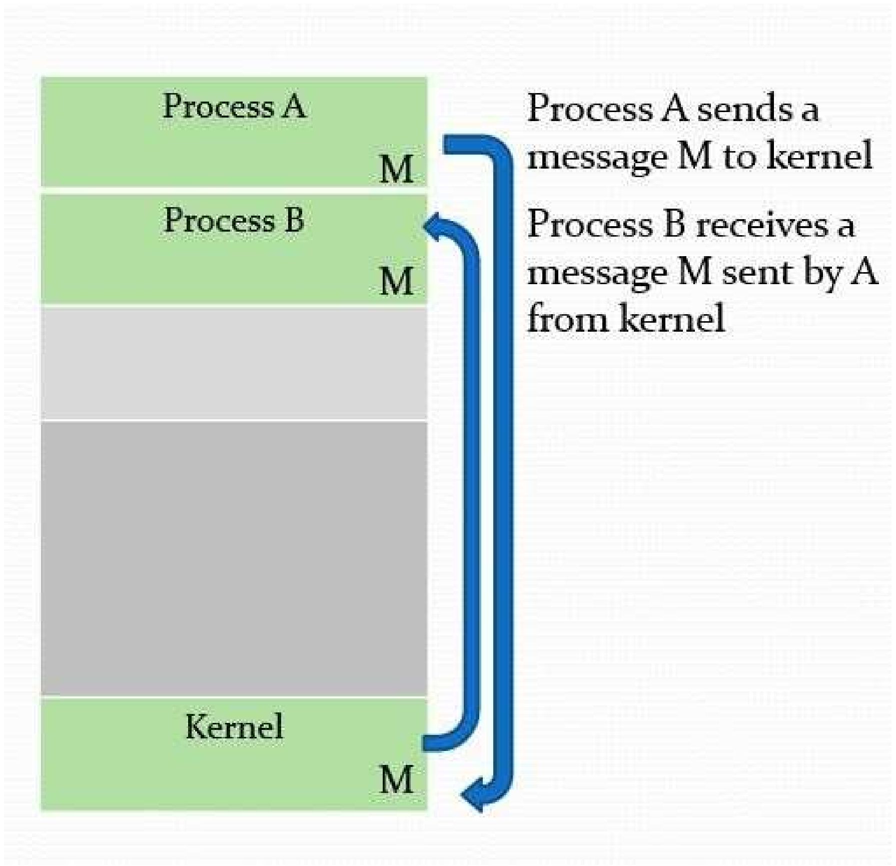
Models for IPC

Shared Memory


Process A
Shared
Process B
Kernel
Process A writes into shared memory
Process B reads from shared memory

Models for IPC

Message Passing


Process A
M
Process B
M
Kernel
M
Process A sends a message M to kernel
Process B receives a message M sent by A from kernel

Race Condition

Race conditions arise in software when separate processes or threads of execution depend on some shared state.

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.

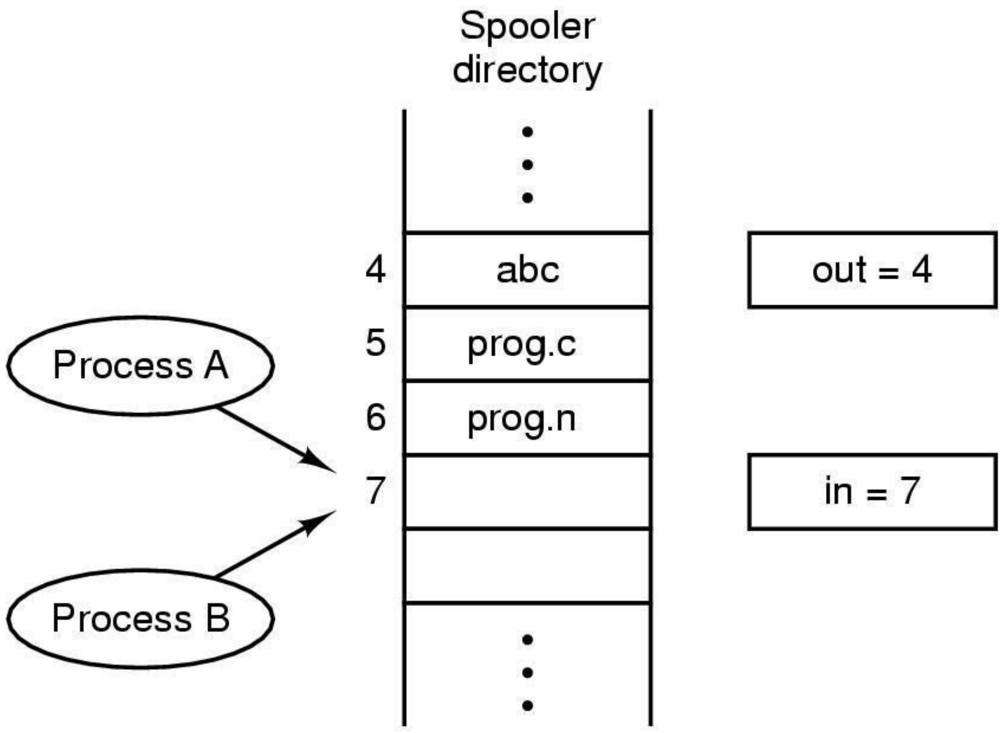
Operations upon shared states are critical sections that must be mutually exclusive. Failure to do so opens up the possibility of corrupting the shared state.

Reasons:

- Exact instruction execution order cannot be predicted.
- Resource(file, memory, data) sharing.

Race Condition Example

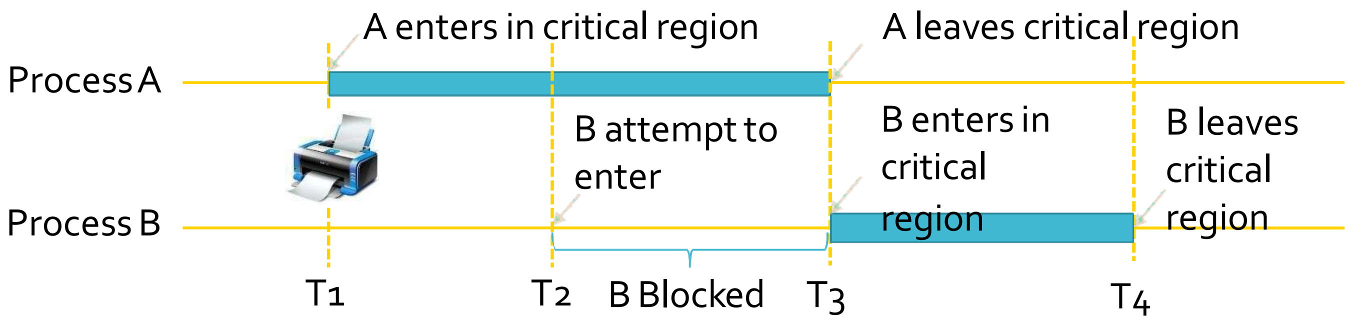
Print Spooler directory; where 2 processes wants to access shared memory at same time.



Critical Section

Critical Section

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



Solving Critical-Section Problem

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

Mutual Exclusion

- **No two processes** may be **simultaneously inside** the same critical section.
- **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.

Mutual exclusion with busy waiting

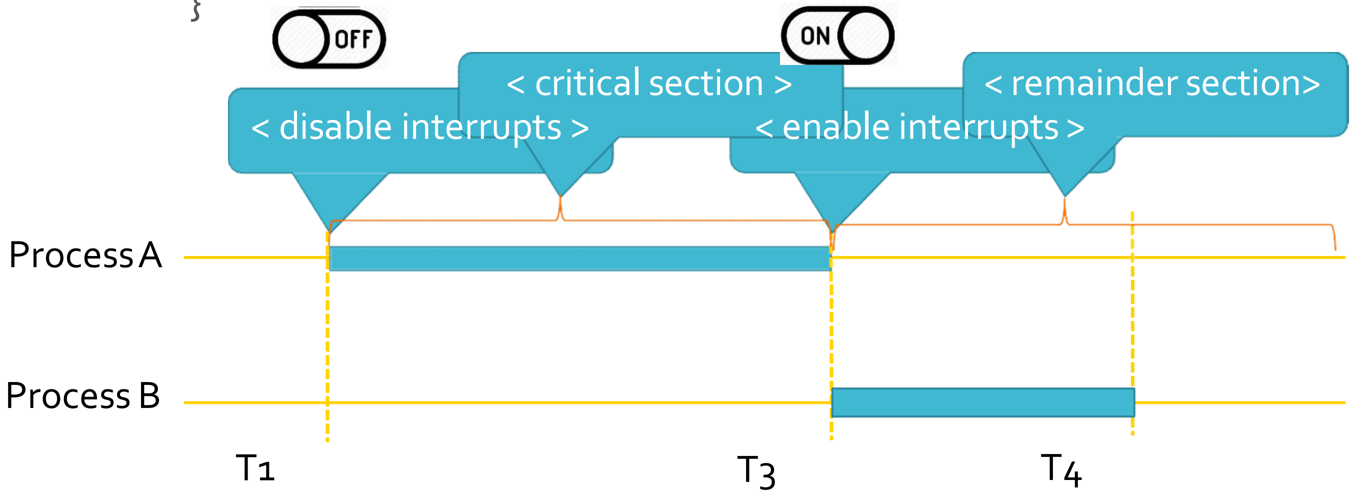
- Disabling interrupts (Hardware approach)
- Shared lock variable (Software approach)
- Strict alteration (Software approach)
- TSL (Test and Set Lock) instruction (Hardware approach)
- Exchange instruction (Hardware approach)
- Dekker's solution (Software approach)
- Peterson's solution (Software approach)

Mutual exclusion with busy waiting

Disabling interrupts (Hardware approach)

```
while (true)
        {
                < disable interrupts >;
                < critical section >;
                < enable interrupts >;
                < remainder section>;

        }
```



Mutual exclusion with busy waiting

Disabling interrupts (Hardware approach)

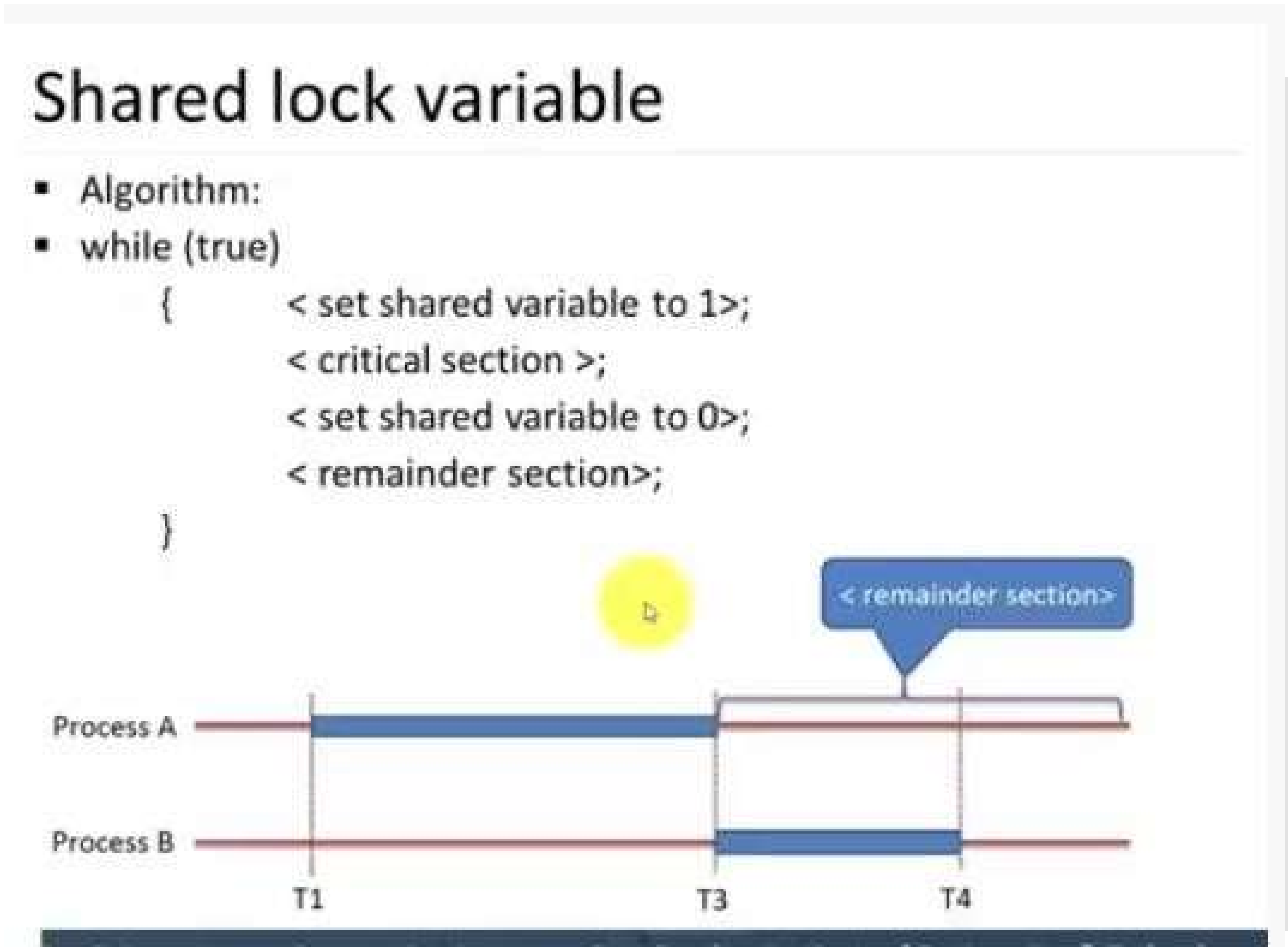
- **Problems**:
  - Unattractive or **unwise to give user processes the power to turn off interrupts**.
  - What if one of them did it (disable interrupt) and never turned them on (enable interrupt) again? That could be the **end of the system**.
  - If the system is a multiprocessor, with two or more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

Mutual exclusion with busy waiting

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.

Mutual exclusion with busy waiting



Shared lock variable

- Algorithm:
- while (true)

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

Mutual exclusion with busy waiting

Shared lock variable (Software approach)

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**.

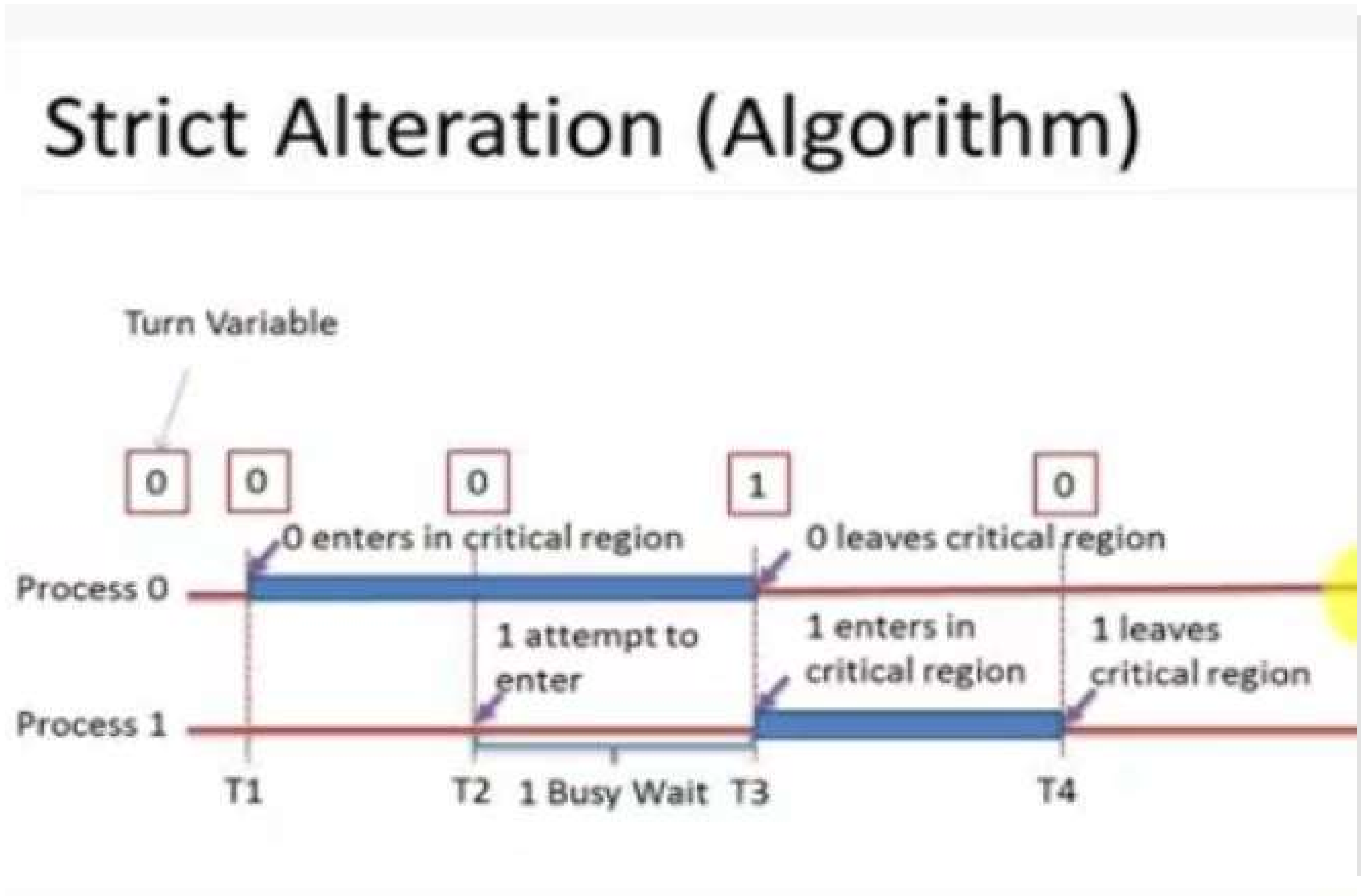
Now **two processes are in their critical regions** accessing the same shared memory, which violates the mutual exclusion condition.

Mutual exclusion with busy waiting

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.

Mutual exclusion with busy waiting



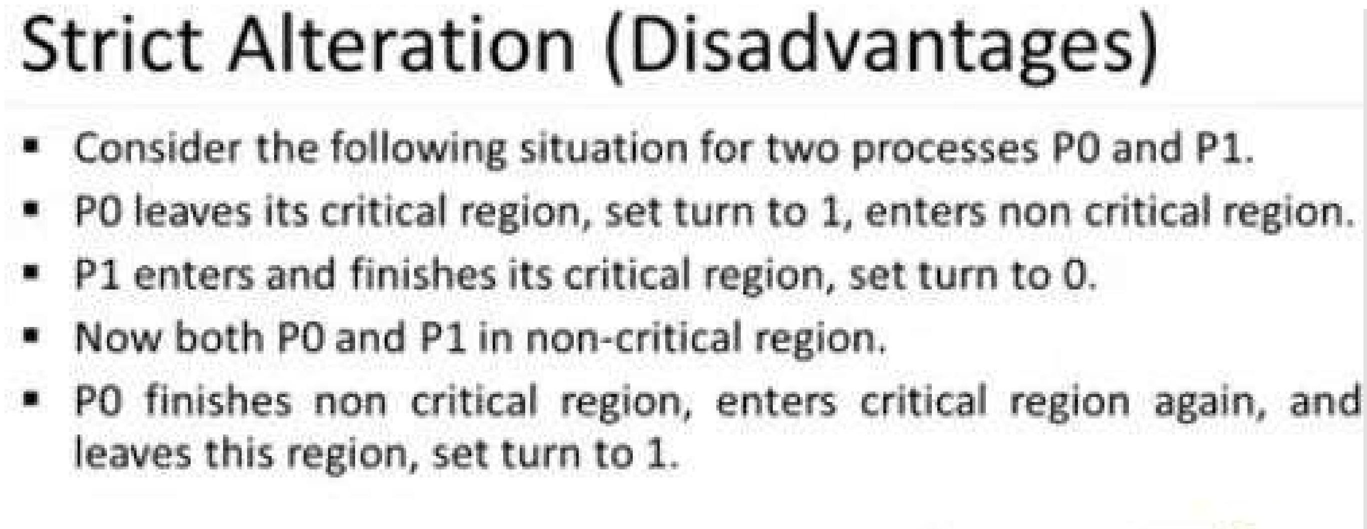
Strict Alteration (Algorithm)

Mutual exclusion with busy waiting

Strict alteration (Software approach)

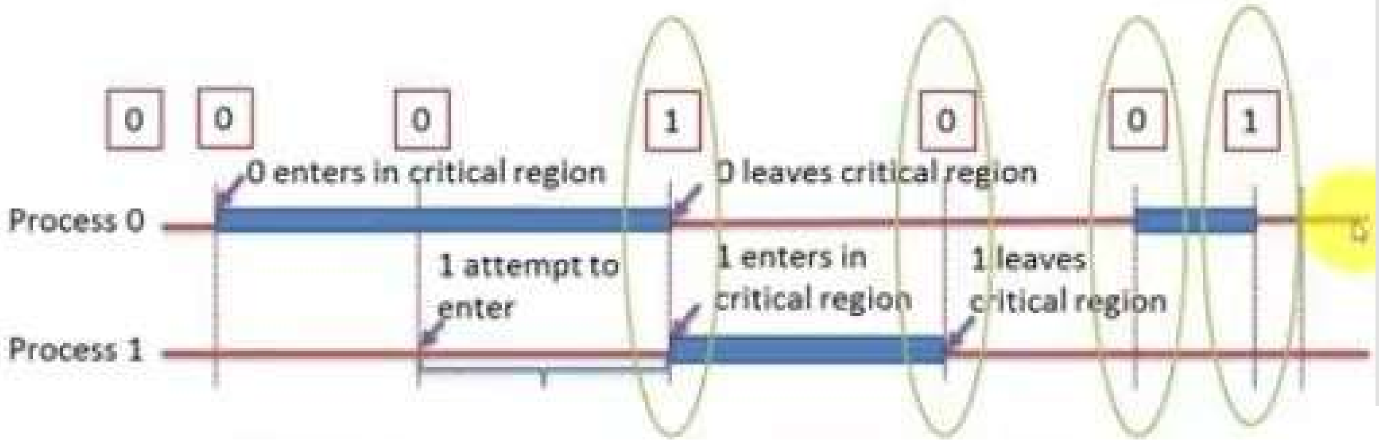
Process 0	**Process 1**
while (TRUE)	while (TRUE)
{	{
while (turn != 0); /* loop */ ;	while (turn != 1); /* loop */ ;
critical_region();	critical_region();
turn = 1;	turn = 0;
noncritical_region();	noncritical_region();
}	}

Mutual exclusion with busy waiting



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.

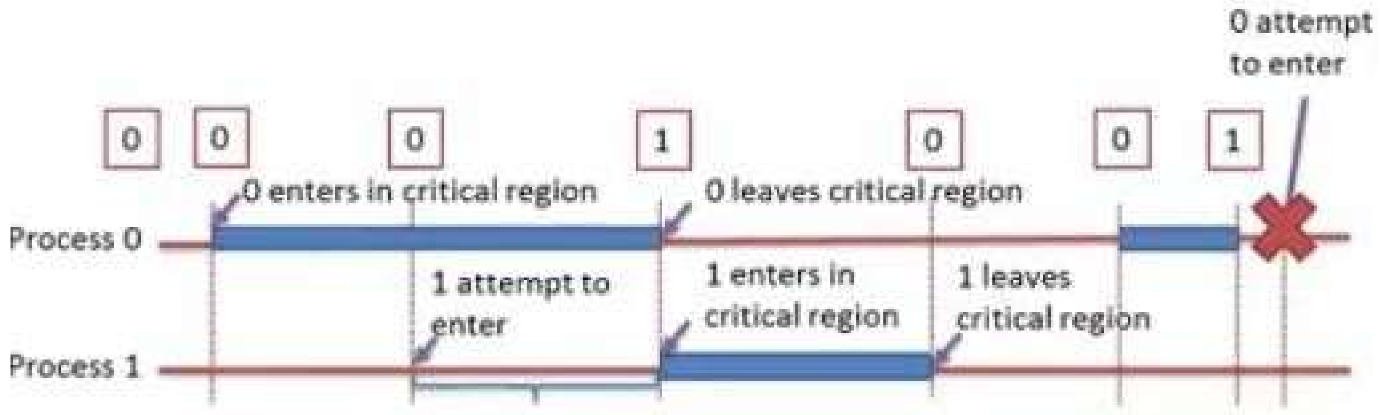


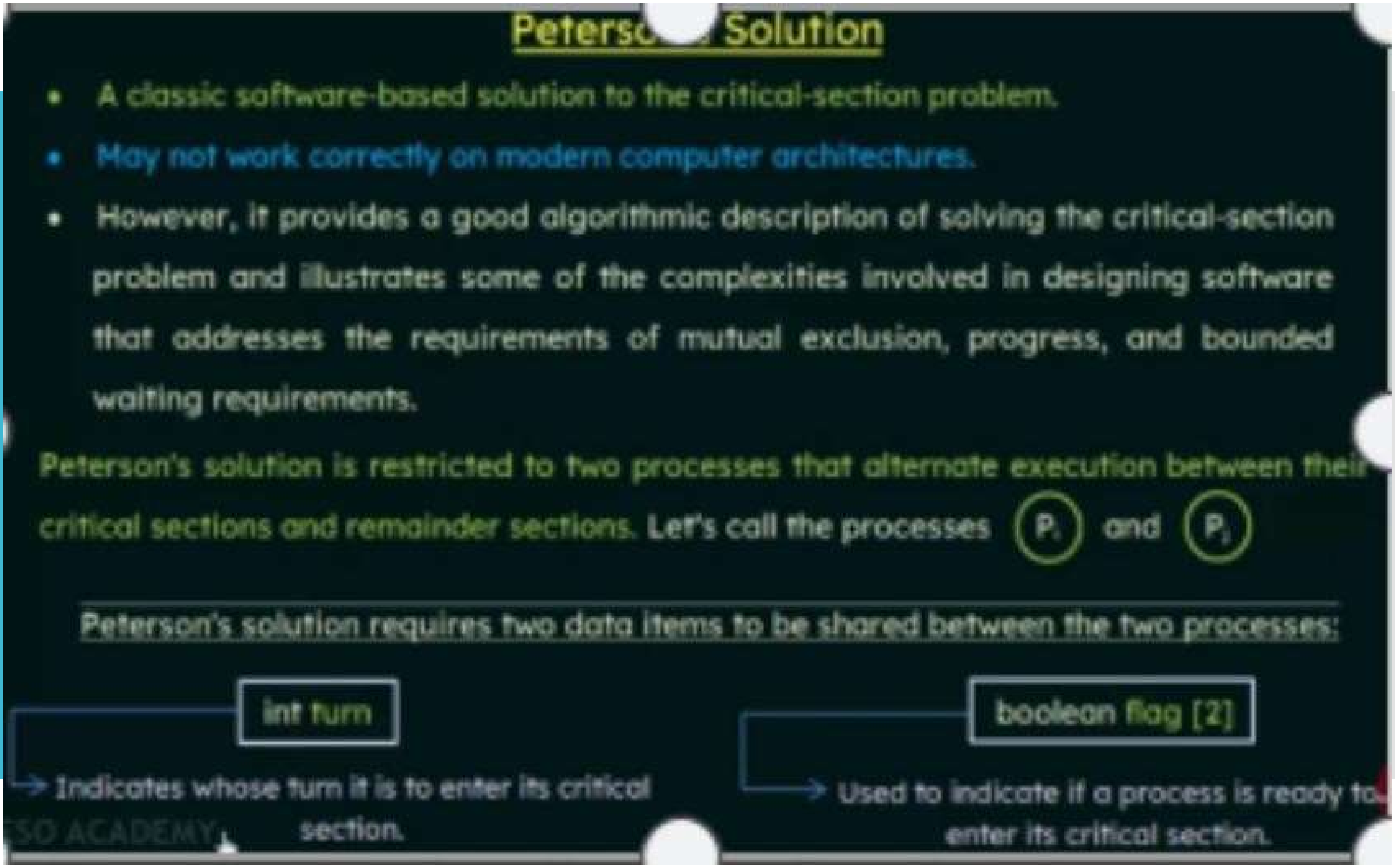
Mutual exclusion with busy waiting

Strict Alteration (Disadvantage)

Taking turn is not a good idea when one of the processes is much slower then the other. For Example

- 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.





Peterso[illegible] Solution

- A classic software-based solution to the critical-section problem.
- May not work correctly on modern computer architectures.
- However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Let's call the processes P_i and P_j

Peterson's solution requires two data items to be shared between the two processes:

int turn
→ Indicates whose turn it is to enter its critical section.

boolean flag [2]
→ Used to indicate if a process is ready to enter its critical section.

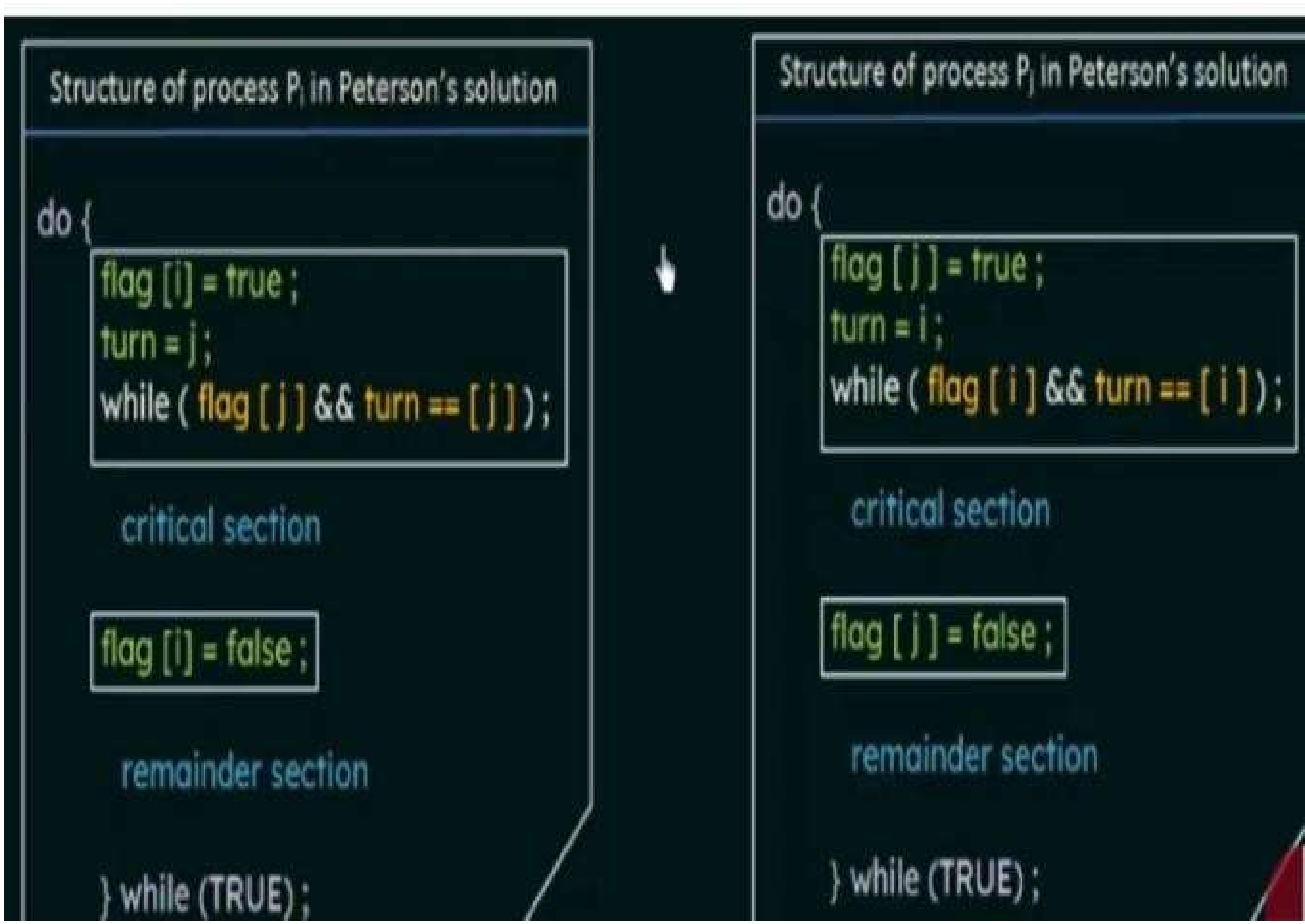
Mutual exclusion with busy waiting

Peterson's solution (Software approach)

- Peterson's solution is restricted to 2 processes that alters it's execution.
- Peterson's solution requires the 2 processes to share two data items:

```
        int turn;              ← Indicates whose turn it is!
        boolean flag[2];       ← Indicates if process "is ready" to enter C.S
```

Mutual exclusion with busy waiting



Structure of process P_i in Peterson's solution

```
do {
        flag [i] = true ;
        turn = j ;
        while ( flag [ j ] && turn == [ j ] ) ;

          critical section

        flag [i] = false ;

          remainder section

} while (TRUE) ;
```

Structure of process P_j in Peterson's solution

```
do {
        flag [ j ] = true ;
        turn = i ;
        while ( flag [ i ] && turn == [ i ] ) ;

          critical section

        flag [ j ] = false ;

          remainder section

} while (TRUE) ;
```

What's wrong with Peterson ?

When a process wants to enter its critical region, it checks to see if the entry is allowed.

- If it is not, the process just sits in a tight loop waiting until it is allowed to enter. **Problem** : Busy waiting-waste of CPU time!

So to avoid above problem system call can be provide like

- Sleep

- wake up

What's wrong with Peterson, TSL ?

(To avoid busy waiting we have IPC primitives - pair of sleep and wakeup)

Solution : Replace busy waiting by blocking calls

Sleep (blocks process): it is a system call that causes the caller to block, that is- process can be suspended until another process wakes it up.

Wakeup (unblocks process): The wakeup call has one parameter, the process to be awakened.

This concept is used in producer-consumer problems.

Producer Consumer Problem: Introduction

It is multi-process synchronization problem.

- It is also known as bounded buffer problem.

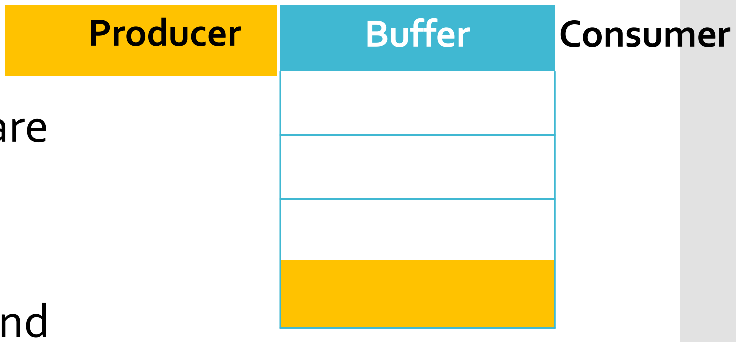
This problem describes two processes **producer** and **consumer**, who share common, fixed size buffer.

Producer process

Produce some information and put it into buffer

Consumer process

- **Consume this information** (remove it from the buffer)



Producer Consumer Problem: Approch

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.

Producer Consumer Problem: Approch

Buffer is empty

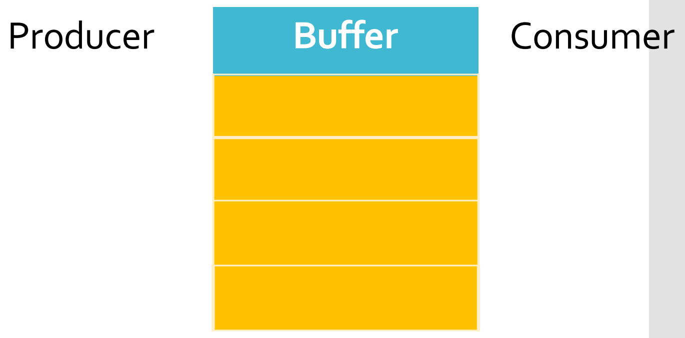
- Producer want to produce
- Consumer want to consume

Buffer is full

- Producer want to produce
- Consumer want to consume

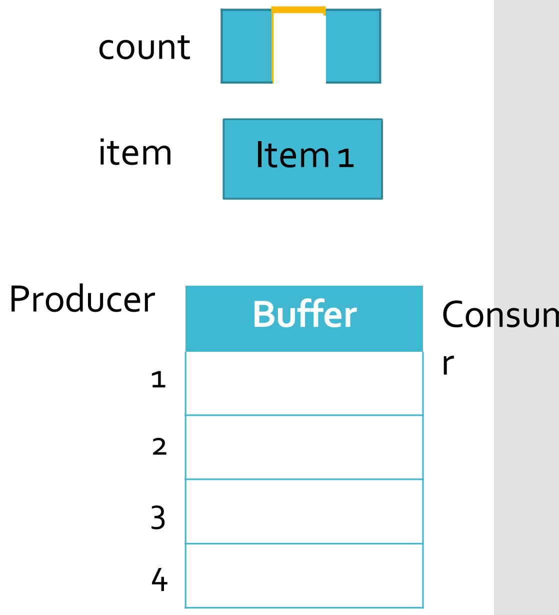
Buffer is partial filled

- Producer want to produce
- Consumer want to consume



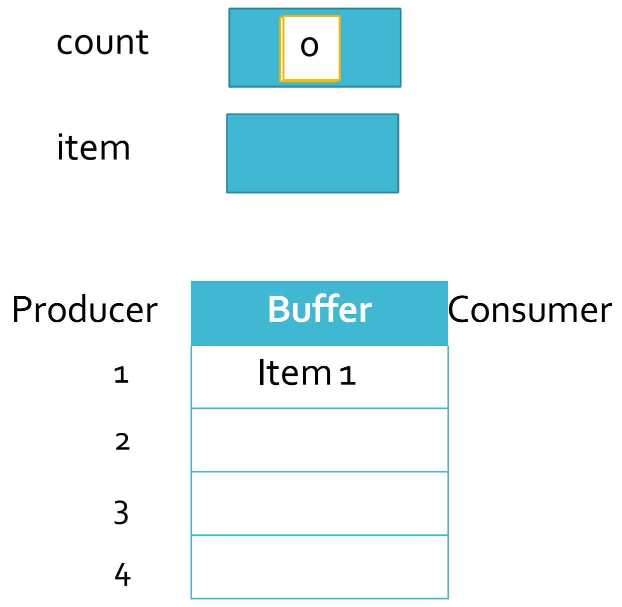
Producer Consumer Problem

```
#define N 100
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);
      }
}
```

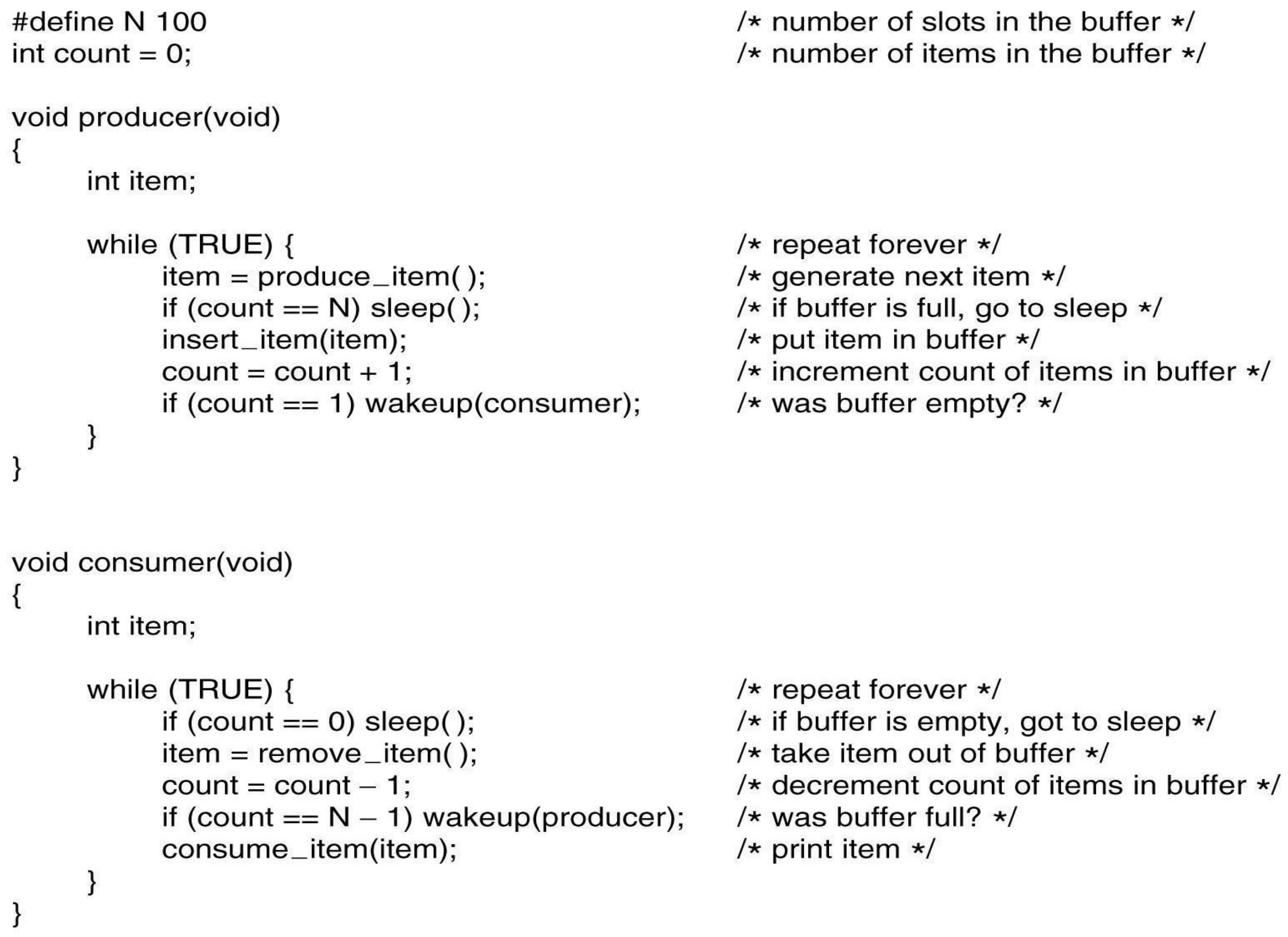


Producer Consumer Problem

```
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);
      }
}
```



Producer Consumer Problem



```
#define N 100                                    /* number of slots in the buffer */
int count = 0;                                   /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                           /* repeat forever */
                item = produce_item( );          /* generate next item */
                if (count == N) sleep( );        /* if buffer is full, go to sleep */
                insert_item(item);               /* put item in buffer */
                count = count + 1;               /* increment count of items in buffer */
                if (count == 1) wakeup(consumer);  /* was buffer empty? */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                           /* repeat forever */
                if (count == 0) sleep( );        /* if buffer is empty, got to sleep */
                item = remove_item( );           /* take item out of buffer */
                count = count − 1;               /* decrement count of items in buffer */
                if (count == N − 1) wakeup(producer);  /* was buffer full? */
                consume_item(item);              /* print item */
        }
}
```

Producer Consumer Problem: Issues

- 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);
        }
}
```

Producer Consumer Problem: Issues

- 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);
        }
}
```

Semaphores

Definition :-

A **semaphore is a variable** that **provides a simple but useful abstraction for controlling access** (of shared resource) by multiple processes to a common resource in a parallel programming or multi user environment.

So, Semaphore is nothing but a synchronized variable, that contain integer value.

A useful way to think of a semaphore is as a record of how many units of a particular resource are available.

Semaphores: Types

Semaphores which allow an arbitrary resource count are called **counting semaphores**(can have possible values more than two)

While semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available, up/down) are called **binary semaphores**.

Counting semaphore is having value of count and Binary semaphore is having value 0/1.

Semaphores (Up Down Operations)

Counting semaphores are equipped with two operations:

- 1. **V** (known as signal() OR up)
- 2. **P** (Known as wait() OR down)

Operations **V** increments the semaphore **S** and **P** decrements the semaphore **S**.

Semaphores (Up Down Operations)

A simple way to understand wait() and signal() operations is:

signal():

Increments the value of semaphore variable by 1.

After the increment, if the pre-increment value was negative (meaning there are processes waiting for a resource), it transfers a blocked process from the semaphore's waiting queue to the ready queue. (like Wake up)

wait():

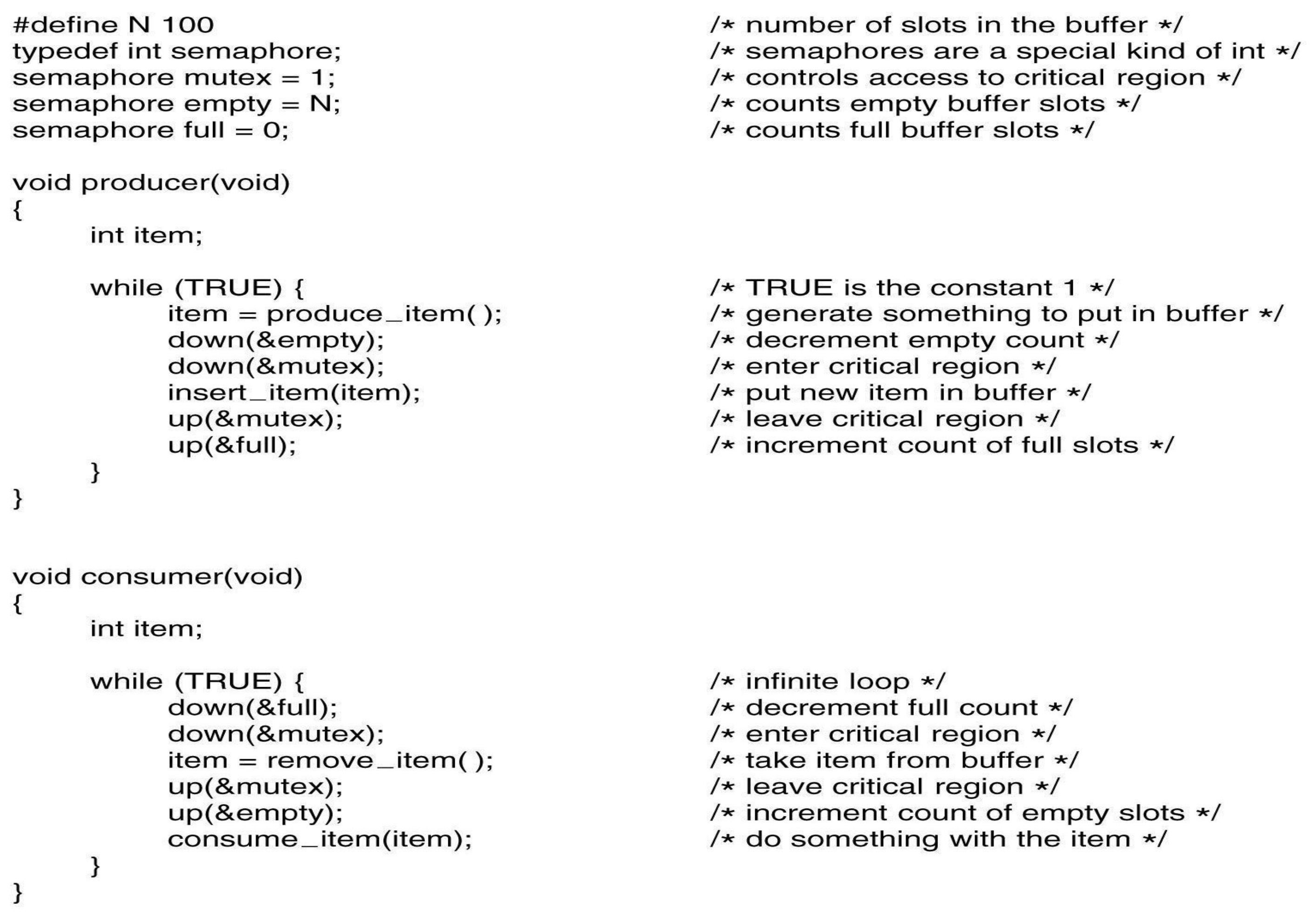
Decrements the value of semaphore variable by 1.

If the value becomes negative or 0, the process executing wait() is blocked (like sleep), i.e., added to the semaphore's queue.

Producer Consumer Problem with Semaphore

- 3 semaphores: full, empty and Mutex
- Full counts- number of full(filled) slots (initially 0)
- Empty counts- number of empty slots (initially N)
- Mutex protects variable which contains the items produced and consumed

Producer Consumer Problem with Semaphore



```
#define N 100                               /* number of slots in the buffer */
typedef int semaphore;                      /* semaphores are a special kind of int */
semaphore mutex = 1;                        /* controls access to critical region */
semaphore empty = N;                        /* counts empty buffer slots */
semaphore full = 0;                         /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                      /* TRUE is the constant 1 */
                item = produce_item( );     /* generate something to put in buffer */
                down(&empty);               /* decrement empty count */
                down(&mutex);               /* enter critical region */
                insert_item(item);          /* put new item in buffer */
                up(&mutex);                 /* leave critical region */
                up(&full);                  /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                      /* infinite loop */
                down(&full);                /* decrement full count */
                down(&mutex);               /* enter critical region */
                item = remove_item( );      /* take item from buffer */
                up(&mutex);                 /* leave critical region */
                up(&empty);                 /* increment count of empty slots */
                consume_item(item);         /* do something with the item */
        }
}
```

Classical IPC Problems: Dinning Philosopher Problem

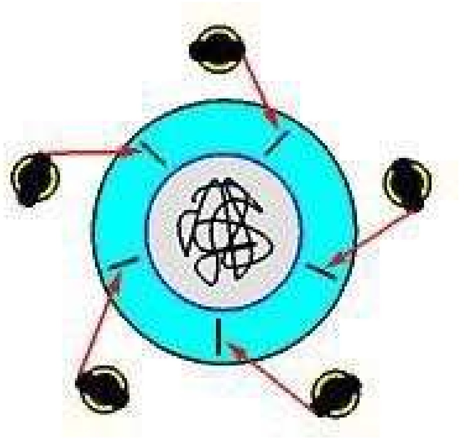
- Philosophers eat and think.

1. To eat, they must first acquire a left fork and then a right fork (or vice versa).
2. Then they eat.
3. Then they put down the forks.
4. Then they think.
5. Go to 1.

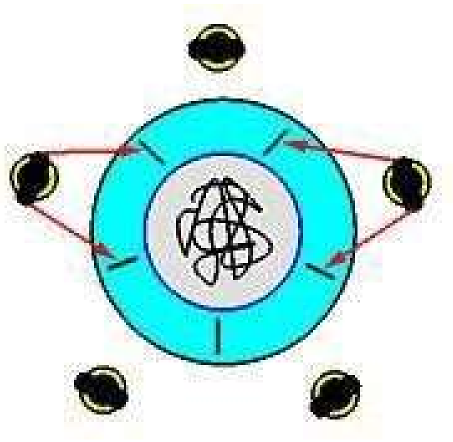
Classical IPC Problems: Dinning Philosopher Problem

- **Problems:**

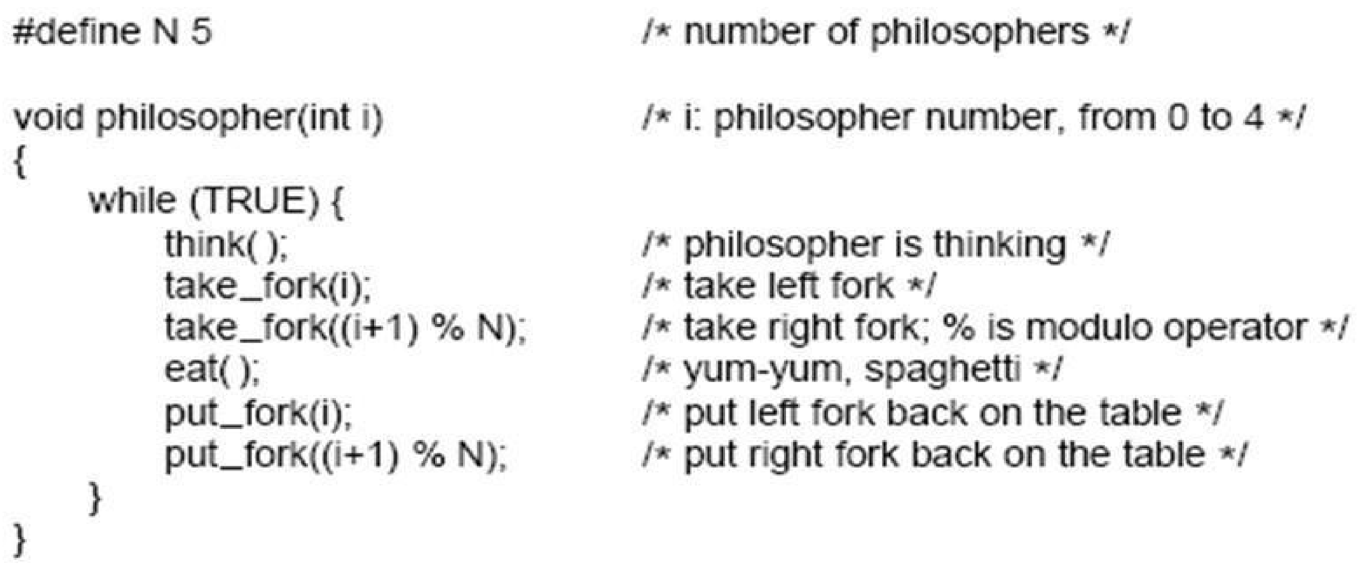
1. Deadlock



2. Starvation

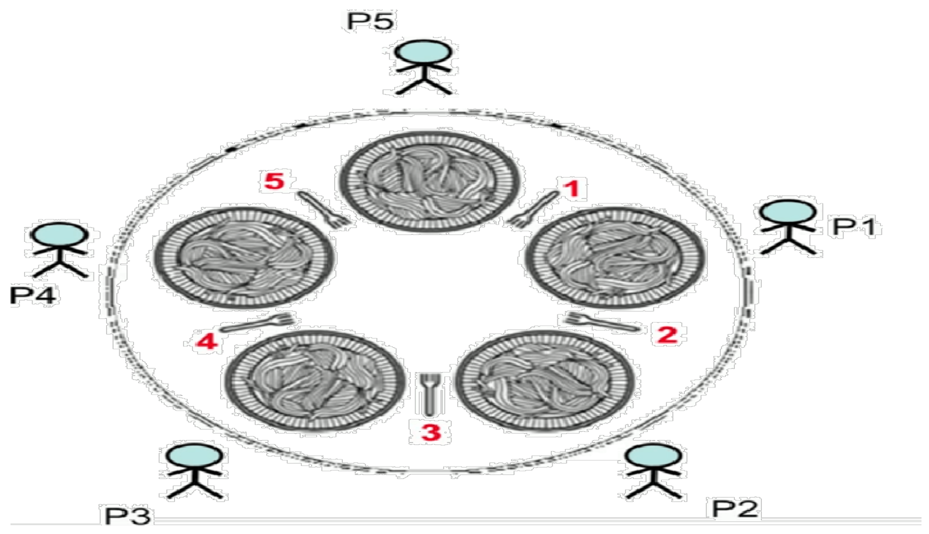


Classical IPC Problems: Dining Philosophers (naïve solution- first try)



```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
      while (TRUE) {
            think( );                /* philosopher is thinking */
            take_fork(i);            /* take left fork */
            take_fork((i+1) % N);    /* take right fork; % is modulo operator */
            eat( );                  /* yum-yum, spaghetti */
            put_fork(i);             /* put left fork back on the table */
            put_fork((i+1) % N);     /* put right fork back on the table */
      }
}
```



Classical IPC Problems: Sleeping Barbar Problem

One barber, one barber chair, and N seats for waiting.

No customers so barber sleeps.

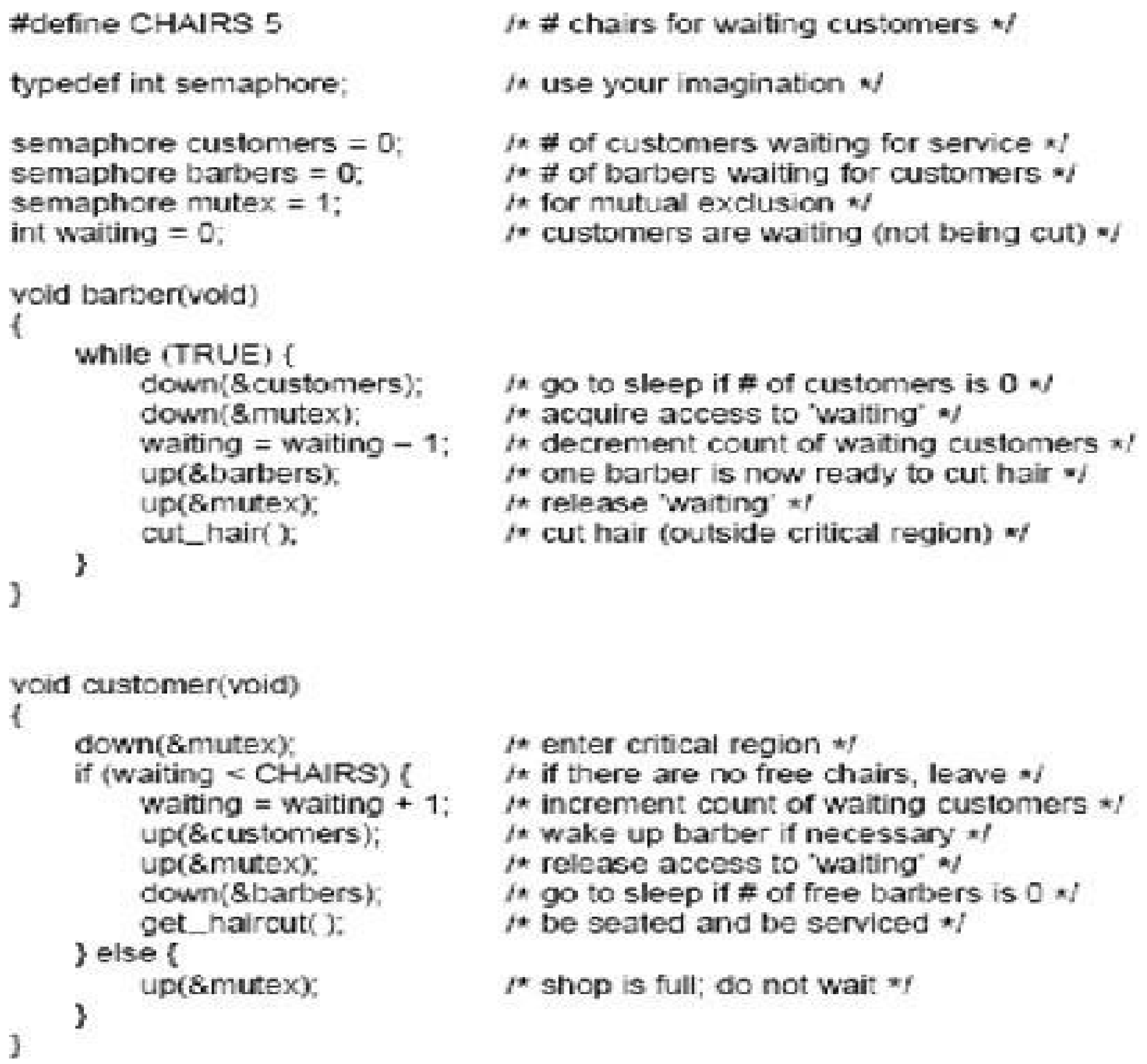
Customer comes in & wakes up barber.

More customers come in.

- If there is an empty seat, they take a seat.
- Otherwise, they leave.

Classical IPC Problems: Sleeping Barbar Problem

Classical IPC Problems: Sleeping Barbar Problem’s Solution



```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting – 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
          cut_hair( );              /* cut hair (outside critical region) */
     }
}

void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

THANK YOU