Marwadi University

Operating System

Unit – 3 & 4 Concurrency & InterProcess Communication

Subject Faculty: Prof. Suhag Baldaniya

Disclaimer

- ***It is hereby declared that the production of the said content is meant for non-commercial, scholastic and research purposes only.***

- ***We admit that some of the content or the images provided in this channel's videos may be obtained through the routine Google image searches and few of them may be under copyright protection. Such usage is completely inadvertent.***

- ***It is quite possible that we overlooked to give full scholarly credit to the Copyright Owners. We believe that the non-commercial, only-for-educational use of the material may allow the video in question fall under fair use of such content. However we honor the copyright holder's rights and the video shall be deleted from our channel in case of any such claim received by us or reported to us.***

Topics to be covered

- Introduction to IPC
- Methods for IPC
- Process Synchronization - Race Conditions ,Critical Section, Mutual Exclusion
- Synchronization Methods - Disabling Interrupts, Hardware Solution, Strict Alternation and Peterson's Solution
- The Producer Consumer Problem
- Semaphores
- Event Counters, Monitors, Message Passing
- Classical IPC Problems: Reader's & Writer Problem, Dinning Philosopher Problem and Sleeper Barber Problem

Introduction to IPC

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.

Introduction to IPC

- **Inter-process communication** (**IPC**) is a set of methods for the exchange of data among multiple **threads** in one or more **processes**.

- 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.***

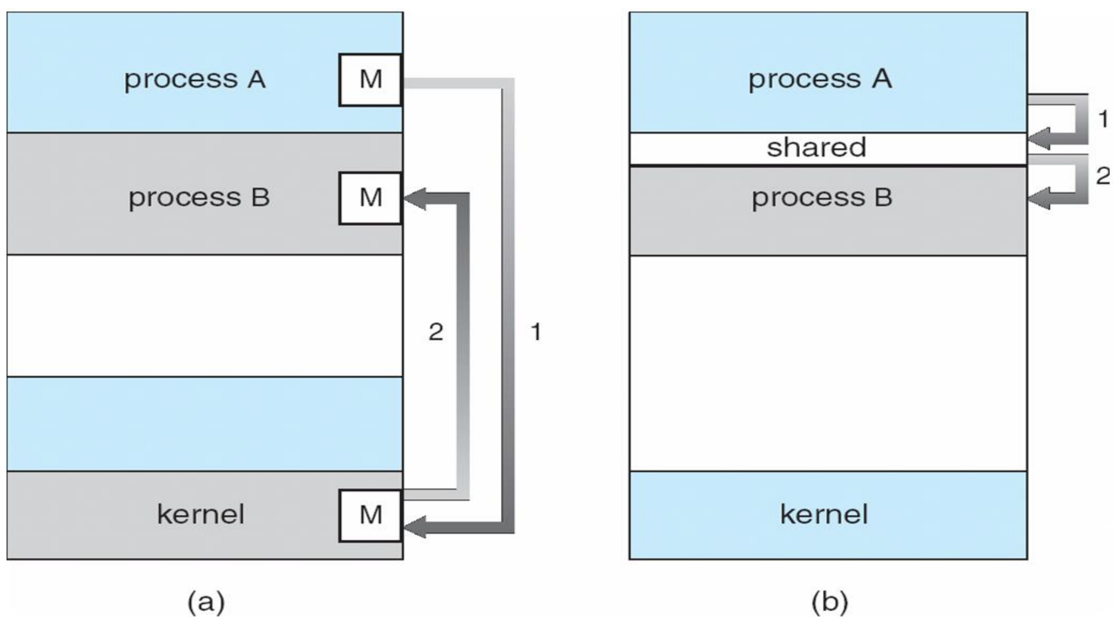
Why IPC ?

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

- Information sharing
- Computational speedup
- Modularity
- Convenience

Methods of communication

- The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:
  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

- 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.

- **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 race condition.

Basic Definitions

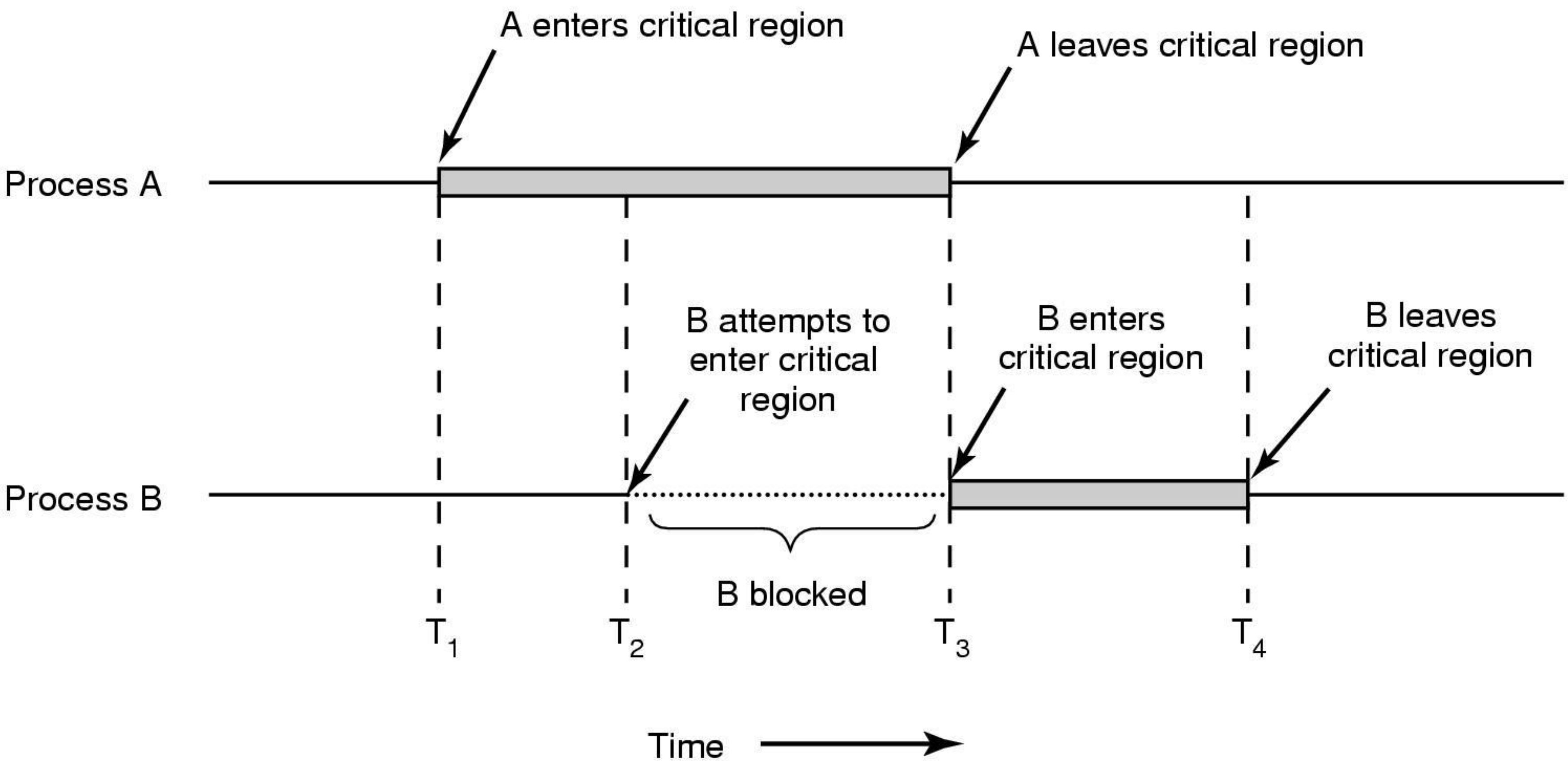
- Situation where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when.
- **Mutual Exclusion and Critical Section**
- **Critical Section**: The **part of program where the shared resource is accessed** is called critical section or critical region.
- **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.

Critical Section (Critical Region)

- Sometimes a process has to access shared memory or files, or do other critical things that can lead to races. This type of **part of program** where shared memory is accessed is called the **critical region** or **critical section.**

- If no two processes are ever in their critical regions at the same time then we could avoid race.

What we are trying to do???



Solving Critical-Section Problem

Any solution to the CS problem must satisfy following criteria:

1. **Mutual Exclusion**
   - Only one process at an instant of time can enter into critical section.
   - Also, critical section must be accessed in mutual exclusion fashion.
2. **Progress**
   - Progress says, only consider those processes should compete which actually wants to access or enter into the critical section.
   - Also, progress is your mandatory criteria.
3. **Bounded Wait**
   - There should be a maximum bound up to which a process can wait.
   - No process should have to wait forever to enter a critical section.

Synchronization Methods

1. **Busy Waiting:** wastage of time, falling in infinite loop, not doing anything but busy only in knocking.

2. **Without Busy Waiting:** instead of knocking again and again process will simply goes to rest mode and whenever the process in critical section gets free, will awake another process and notify it to use the critical section.

Busy Waiting Solutions for CS Problem

A list of proposals to achieve mutual exclusion

1. Disabling interrupts
2. Lock variables
3. Strict alternation
4. Flag Variable
5. Peterson's solution
6. The TSL instruction (Hardware Solution)

Solution-1.Disabling Interrupts

- **Idea:** process disables interrupts, enters critical region, enables interrupts when it leaves critical region
- **Problems**
  - Process might never enable interrupts, crashing system
  - Won’t work on multi-core (multiprocessor/with two or more CPUs)chips as disabling interrupts only effects only the CPU that executed the disabling instruction.
  - the other ones will continue and can access the shared memory.

Solution-2.Lock variable

- **A software solution - everyone shares a lock**
  - When lock is 0, process turns it to 1 and enters in critical region
  - When exit critical region, turn lock to 0
- **Problem - Race condition**
  - suppose that one process reads the lock and sees that it is 0, Before it can set the lock to 1,another process is scheduled, runs and sets the lock to 1.
  - when the first process runs again it will also set the lock to 1 and two processes will be in their critical regions at the same time.

Lock variable

1. While (lock!=0);
2. Lock=1;
3. Enter in Critical Section
4. Lock=0;
5. Exit CS

Solution-3 Strict Alternation

- 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 critical section.
- Process 1 also finds it to be 0 and therefore sits in loop continually testing ‘turn’ to see when it becomes 1.
- countinusoly testing a variable until some value appears is called 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 critical region.**

Strict Alternation

For Process P0

```
While (TRUE)
{
while ( turn != 0); /* trap */
Critical_region();
turn =1; /*a process exit CS*/
noncritical_region();
}
```

For Process P1

```
While (TRUE)
{
while ( turn != 1); /* trap */
Critical_region();
turn =0; /*a process exit CS*/
noncritical_region();
}
```

Solution 4. Flag Variable

For Process P0

```
While(True)
{
Flag[0]=T; /*process is interested
    to enter into CS*/
while ( Flag[1]); /* trap*/
Critical_region();
Flag[0] =F; /*a process exit CS*/
noncritical_region();
}
```

For Process P1

```
While(True)
{
Flag[1]=T; /*process is
    interested to enter into CS*/
while ( Flag[0]); /* trap*/
Critical_region();
Flag[1] =F; /*a process exit CS*/
noncritical_region();
}
```

Solution-5 Peterson's Solution

For Process P0

```
While(True)
{
Flag[0]=T; /*process is
    interested to enter into CS*/
Turn = 1;
while ( turn==1 && Flag[1]==T);
    /* trap*/
Critical_region();
Flag[0] =F; /*a process exit CS*/
noncritical_region();
}
```

For Process P1

```
While(True)
{
Flag[1]=T; /*process is
    interested to enter into CS*/
Turn = 0;
while ( turn==0 && Flag[0]==T);
    /* trap*/
Critical_region();
Flag[1] =F; /*a process exit CS*/
noncritical_region();
}
```

Peterson’s Solution

- Process 0 & 1 try to get in simultaneously.
- Last one in sets turn: say it is process 1.
- Process 0 enters (turn= = process is False).
- First written value of variable will be overwritten & lost.

6. Hardware Solution - TSL (Test & Set Lock)

- It is hardware based mechanism to solve cs problem.
- There is one shared variable whose value is 0 or 1.
- If the value is =0 , which indicates unlock.
- If the value is =1, which indicates lock.
- **Testing part :**
- Here process check the status of the lock, it is 0 or 1.
- If the lock value is 0 than it is available.
- If the lock value is 1 than it should be wait.
- Set lock p

Setlock part

Boolean TestAndSet(Boolean *target) //*target is used to store the value of lock variable

{

Boolean rv=*target;

*target=true;

Return rv;

}

While(TestAndSet (&lock));// Passing the value of lock

Enter into CS

Lock=false;

Remainder section;

}

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.
- The scheduling rules are such that *H* is run whenever it is in ready state.
- At a certain moment, with *L* in its critical region, *H* becomes ready to run (e.g., an I/O operation completes).
- *H* now begins busy waiting, but since *L* is never scheduled while *H* is running, *L* never gets the chance to leave its critical region, so *H* loops forever.
- This situation is sometimes referred to as the **priority inversion problem.**

What’s wrong with Peterson, TSL ?

- 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.
- **Limitations:**
    1. **Busy Waiting**: this approach waste CPU time
    2. **Priority Inversion Problem**: a low-priority process blocks a higher-priority one

Sleep and Wake up

- 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, be suspended until another process wakes it up.
  - **Wakeup** (unblocks process): The wakeup call has one parameter, the process to be awakened.

The Producer-Consumer Problem

- It is also known as ***Bounded Buffer Problem*** **in (multi-process synchronization problem).**
- Consider two processes Producer and Consumer , who **share common, fixed size buffer.**
- Producer **puts information** into the buffer
- Consumer **consume** this **information** from buffer.
- Trouble arises when the producer **wants to put a new item in the buffer, but it is already full.**
- And consumer **wants to remove a item from buffer, but it is already empty.**

The Producer-Consumer Problem

Consumer

```
Void consumer()
{
int item;
While(True)
{
    while(count==0); /*buffer
    empty*/
     item= buffer(out);
     out=(out+1)mod n;/*out is a local
         variable
     count = count – 1;
     Consume_item(item)
}
}
```

Producer

```
int count = 0;/*count indicates
    how many items in buffer
void producer()
{
     While(True)
     {
     Produce_item(item);
     while(count==n); /*buffer full*/
     Buffer[in]=item;/*in is a local
         variable and in indicates index
         of item
     in=(in+1) mod n;
     count = count+1;
}
```

The Producer-Consumer Problem

Consumer

```
Do{

Wait(full);// wait until full>0 than decrement full.
Wait(mutex);// acquire lock
Signal(mutex); //release lock
Signal(empty); // increment empty
}
While(true);
```

Producer

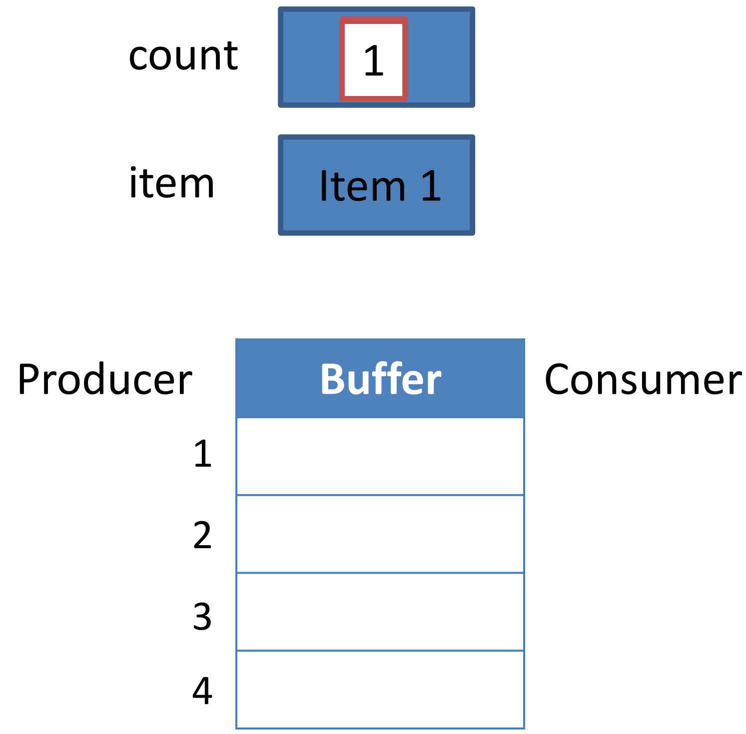
```
Do{

Wait(empty);// wait until empty>0 than decrement empty.
Wait(mutex);// acquire lock and add data to buffer
Signal(mutex); //release lock
Signal(full); // increment full
}
While(true);
```

Solution - Producer-Consumer Problem

- **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.

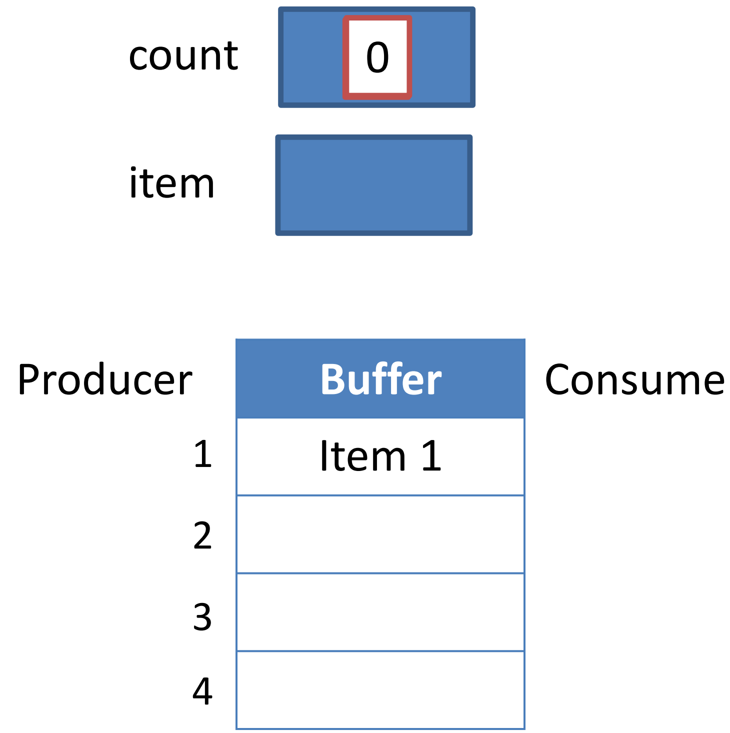
Producer Consumer problem using Sleep & Wakeup

```
#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);
      }
}
```



Semaphore

- Semaphore is a non-negative integer value shared between processes.
- Introduced by Edsger Dijkstra, used to solve cs problem.
- A semaphore is a special integer variable that apart from initialization, is accessed only through two standard operations **i.e. wait() and signal().**

Wait() operation

```
wait(s)
{
    while(s<=0);
    s = s – 1;
}
```

Signal() operation

```
signal(s)
{
    s = s + 1;
}
```

Wait() & Signal() Operation

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

- **wait():** Also denoted as P(Proberen).(Test)
- 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.

- **signal(): Also denoted as V(Vehrogen).(Increment).**
- 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)

Usage of Semaphore

- **Semaphore is used for –**

1. For solving CS problem (Initial Value = 1)
2. For deciding the order of execution among processes
3. For managing resources

Types of Semaphores

- There are two types of semaphores as follows:

1. **Binary Semaphore**
2. **Counting Semaphore**

- Semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available, up/down) are called **binary semaphores** (same functionality that **mutexes** have).
- Semaphores which allow an arbitrary resource count are called **counting semaphores**(can have possible values more than two)

Producer Consumer with Semaphores

Semaphore S = 1 //Take care of critical section

Semaphore E = n //Empty Slots /*n is the size of buffer*/

Semaphore F = 0 //Full Slots

```
void producer()
{
while(T){
    produce();
    wait(E)  //check empty slots
    wait(S)              /*Critical
    insert_item();               Section*/
    signal(S)
    signal(F) //incr one in buffer
    }
}
```

```
void consumer
{
while(T) {
    wait(F)   //check buffer is full
    wait(S)                  /*Critical
    remove_item();                   Section*/
    signal(S)
    signal(E)   //incr one in buffer
    use();
    }
}
```

Classical IPC Problems

- Readers and Writers Problem
- Dinning Philosopher Problem
- Sleeping Barber Problem

Reader's Writer's Problem

- Multiple readers can concurrently read from the data base.
- But when updating the DB, there can only be one writer (i.e., no other writers and no readers either)
- Writers should have exclusive access to write data.

Case	Process 1	Process 2	Allowed / Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Reading	Writing	Not Allowed
Case 3	Writing	Reading	Not Allowed
Case 4	Reading	Reading	Allowed

Readers-Writers

Semaphore wrt = 1 **// for writer**

Semaphore mutex = 1 **// for reader**

Int readcount = 0 **/*readcount is a shared variable, which contains the count of readers in the CS*/**

For writer

```
{
    wait(wrt) // it will check if there is no other writer is
    prossesing or not
    write operation;
    signal(wrt)
}
```

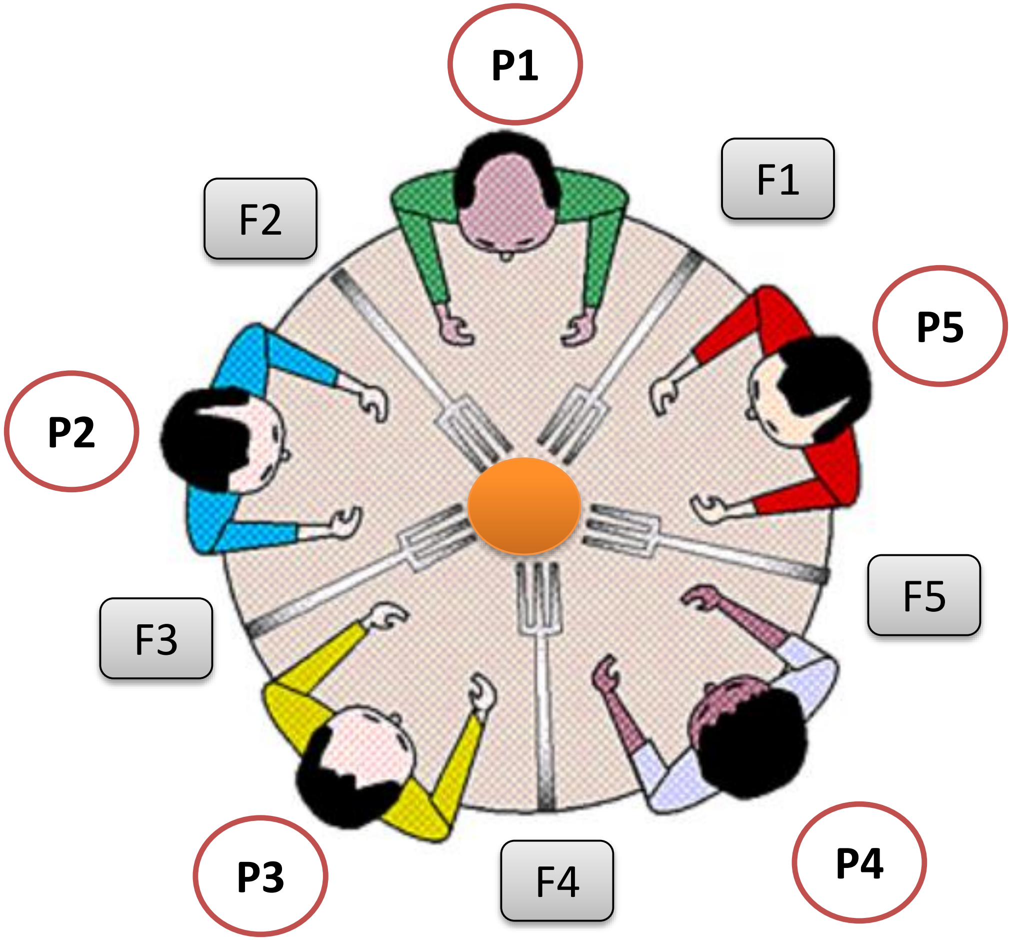
For reader

```
{
    wait(mutex)
    readcount ++;
    if(readcount==1)    /*Actually, here we are checking  first reader
                          or not?*/
    wait(wrt)           // no writer can enter in CS
    signal(mutex) //other reader can enter into cs
    //read operation;
    wait(mutex)
    readcount--;  //reader wants to leave
    if(readcount==0)    //Last reader or No reader
    signal(wrt)         //Now writer can enter in CS
    signal(mutex)
}
```

Dinning Philosopher Problem

- Philosophers eat and think.

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

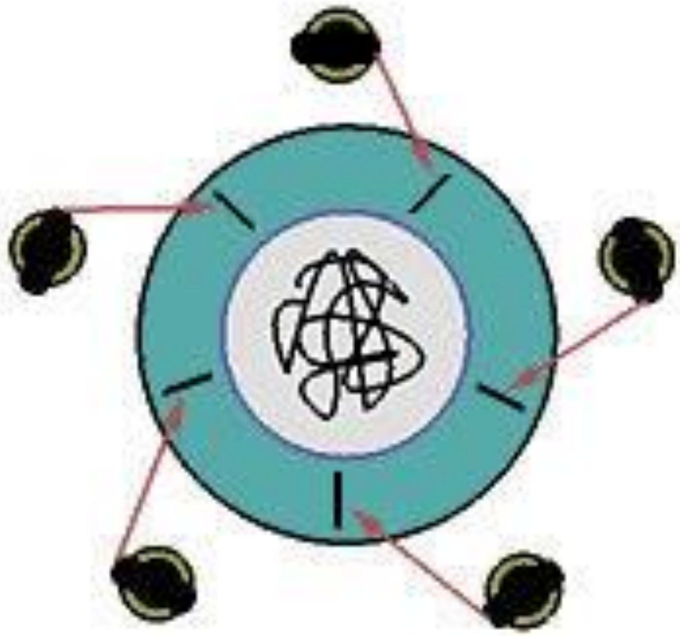


What is a Problem?

- The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table
- Their job is to think and eat alternatively.
- A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers.
- To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available.
- In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

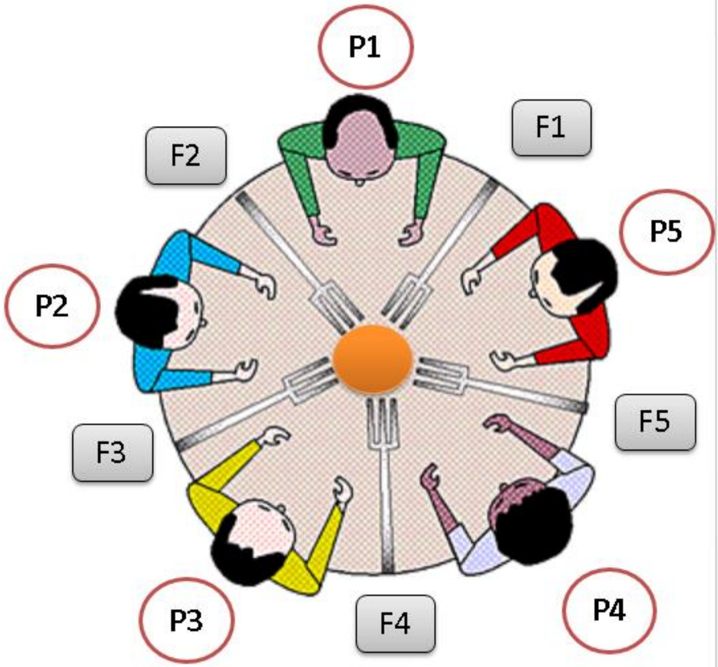
- Problems:

Deadlock



Dinning Philosopher Problem

```
Void philosopher()
{
    while(T)
    {
        think();
        take_fork(i);          //Left fork
        take_fork((i+1)%n);    //Right fork
        eat();
        put_fork(i);           //Left fork
        put_fork((i+1)%n);     //Right fork
    }
}
```



Solution (Not correct)

- Solution of **Dining Philosopher** problem using **Semaphore**

Void philosopher()

```
{
    while(T) {
        think();
        take_fork(Si);          //Left fork
        take_fork((Si+1)%n);    //Right fork
        eat();
        put_fork(Si);           //Left fork
        put_fork((Si+1)%n);     //Right fork
    }
}
```

S[i] five semaphores (i.e. equal to number of forks)
S1, S2, S3, S4, S5

P1 →	S1	S2
P2 →	S2	S3
P3 →	S3	S4
P4 →	S4	S5
P5 →	S5	S1

Solution

P1 →	**S1**	**S2**
P2 →	**S2**	**S3**
P3 →	**S3**	**S4**
P4 →	**S4**	**S5**
P5 →	**S1**	**S5**

Sleeping Barber 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.

Sleeping Barber Problem

```
Semaphore customer = 0          // num of customers waiting for haircut
Semaphore barber = 0            // barber waiting for customers(sleeping)
Mutex = 1                       // for Mutual exclusion among chairs available
int NumofEmptyChairs = N;       /*total num of seats available at
                                  barber shop*/
For Barber {
    while(T) {
        wait(customer); /* waits for a customer (sleeps). */
        wait(mutex);    /* mutex to protect the number of available
                           seats.*/
        NumofEmptyChairs++; /* a chair gets free.*/
        signal(barber); /* bring customer for haircut.*/
        signal(mutex); /* barber is cutting hair.*/
} }
```

For Customer

```
{
    while(T) {
    wait(mutex);
    if(NumofEmptyChairs>0)
    {
        NumofEmptyChairs--; /* sitting down.*/
        signal(customer); /* notify the barber. */
        signal(mutex); /* release the lock */
        wait(barber); /* wait in the waiting room if barber is busy. */
    }
    else
        signal(mutex); /* release the lock customer leaves  */
}
```

Mutex

- When semaphore ability is not needed a simplified version of the semaphore, called a mutex is used.
- Mutexes are good only for managing mutual exclusion to some shared resource.
- A mutex is a variable that can be in one of the two states (only one bit is required to represent it) with 0 meaning unlocked and 1 meaning locked:
  - Unlocked
  - Locked
- Two procedures are used for Mutex:
  - Mutex_Lock //to enter into CS
  - Mutex_Unlock //to exit from CS

Monitors

- Tony Hoare (in 1974) and Per Brinch Hansen (in 1975) proposed a new synchronization structure called a monitor.
- Monitors are features to be included in high level programming languages.
- **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

- Monitors have an **important property** that makes them useful 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.

Monitors

- 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, say**, 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.

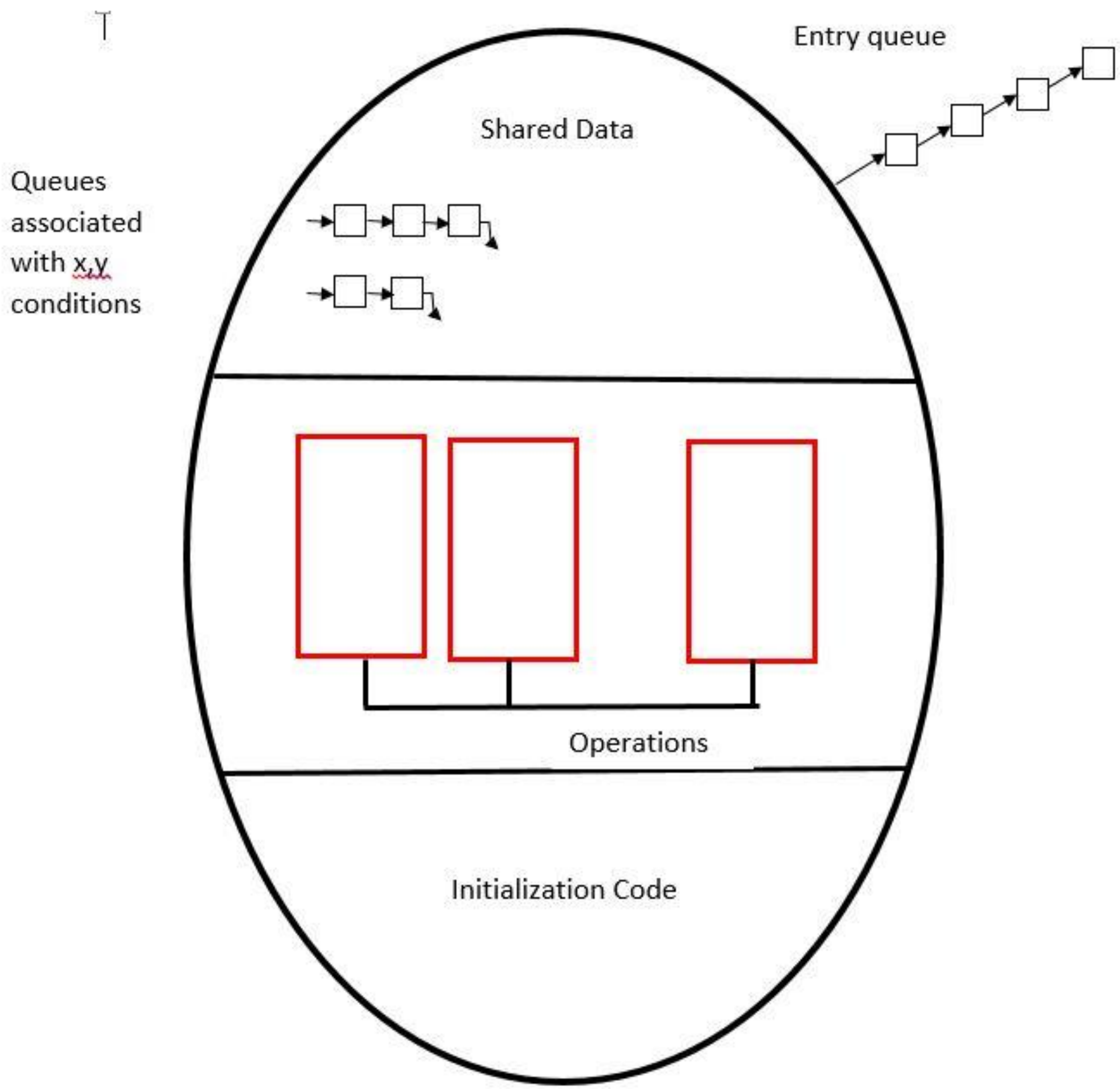
Monitors

- This other process, for example, 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(restore).

Monitors



Entry queue
Shared Data
Queues associated with x,y conditions
Operations
Initialization Code

Monitor Syntax

```
monitor Demo                    // Name of monitor
    Variables;
    condition variables;

    procedure p1();
    .....
    end;
    procedure p2();
    .....
    end;
end monitor;
```

Producer Consumer Problem using Monitors

```
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 Monitors

```
function remove (item : 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 ;
```

Event Counters

- An **event counter** is another **data structure** that can be **used for process synchronization.**
- Like a semaphore, it has an **integer count** and a **set of waiting process identifications**.

- Unlike semaphores, the count variable only increases. It is similar to the **“next customer number”** used in systems where each customer takes a sequentially numbered ticket and waits for that number to be called.

Event Counters

- **Read(E):** return the count associated with event counter E.

- **Advance(E):** atomically increment the count associated with event counter E.

- **Await(E, v):** if E.count <= v, then continue.

  Otherwise, block until E.count > v.

Producer-Consumer with Event Counters

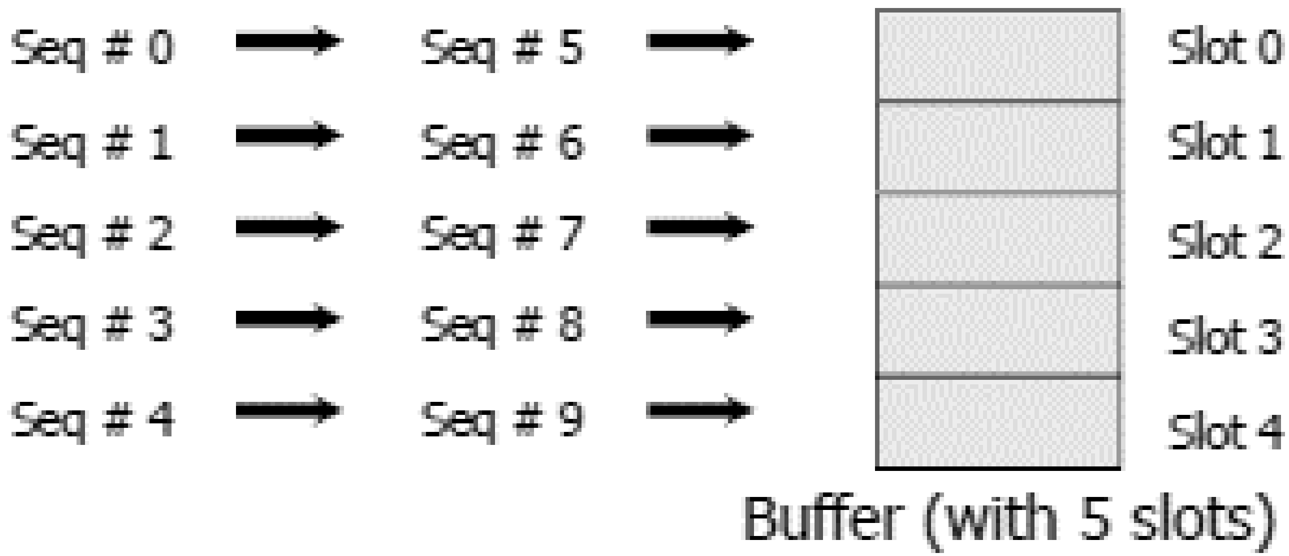
- Two event counters are used, **in and out**, each of which has an **initial count of zero.**

- Each process includes a **private sequence variable (v)** which **indicates the sequence number** of the **last item** it has produced or will consume**. Items are sequentially produced and consumed.**

- **Each item** has a **unique location** in the buffer (based on its sequence number), so **mutually exclusive access to the buffer is not required.**

Mapping Sequence Numbers to Buffer Locations

- Note that items with sequence numbers 0 and 5 cannot both be in the buffer at the same time, as this would imply that there are six items in the buffer.




THANK
YOU