CHAPTER 2 PROCESS SYNCHRONIZATION

Process Synchronization

- **"Process Synchronization is a mechanism to ensure a systematic sharing of resource among concurrent processes."**
- Process Synchronization means sharing system resources by processes in a way that, concurrent access to shared data is handled thereby minimizing the chance of inconsistent data.
- Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.
- Process Synchronization was introduced to handle problems that arose while multiple process executions. Some of the problems are discussed below.
- There are two types of process:
    - Co-operative Process
    - Independent Process

What is a race condition in operating system?

- 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
- A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.
- For Example, two processes Process P1 and Process P2 both sharing one common Variable called **shared**.
- The value for shared variable = 5.
- For Execution, we have 2 Possibilities as follows:

Possibility 1:

In first possibility CPU will Select **Process P1 first** for Execution,

Process P1 and P2 share common variable called int shared = 5	
Process P1	**Process P2**
1. int x=shared; 2. x++; 3. sleep(3); 4. shared=x;	1. int y=shared; 2. y - -; 3. sleep(3); 4. shared=y;

- First Instruction of process P1 is executed which is int x=Shared, Variable x is initialized as Shared so now x = 5;
- In next instruction is X++, so x will be increment by 1 so value of x is now x=6;
- In the 3rd instruction, Process P1 is interrupted and will sleep for 3 Milliseconds so that CPU can context switch to Process P2 and start its execution.
- First Instruction for process P2 will be executed which is int y=Shared, variable y is initialized as Shared so now y=5.
- Next instruction is y- -, so y will be decremented by 1 and after decrementing value of y = 4;
- In the 3rd instruction, Process P2 is interrupted and will sleep for 3 Milliseconds so that CPU can context switch to Process P1 and resume its execution from 4th instruction.
- The 4th Instruction of Process P1 is shared=x; which means the value for shared is now 6. (Logically incorrect). P1 is completed so it is terminated.
- CPU will again switch the context to process p2 to resume its execution with the 4th instruction which is shared = y; which means the value for shared is now 4. (Logically incorrect) P2 is completed so it is terminated.
- The **Final Value of shared variable is 4** after completion of two processes P1 and P2. But when we perform 1 increment and 1 decrement then the final value of shared should be 5. But due to lack of process synchronization, answer we receive is incorrect and this situation is called **Race Condition.**

Possibility 2:

- Here, CPU Select **Process P2 first** for Execution,

Process P1 and P2 share common variable called int shared = 5	
Process P2	**Process P1**
1. int y=shared; 2. y - -; 3. sleep(3); 4. shared=y;	1. int x=shared; 2. x++; 3. sleep(3); 4. shared=x;

- First Instruction of process P2 is executed which is int y=shared, variable y is initialized as shared so y=5;

- Next instruction for process P2 is y- -, y will be decremented by 1 so value of y is now y=4;
- In 3rd instruction Process P2 is interrupted and will sleep for 3 Milliseconds so CPU will context switch and select Process P1 and start its execution.
- For Process P1, 1st instruction will be executed which is int x=shared, Variable x is initialized as shared so now x=5;
- Next instruction is x++, x will be increment by 1 so value of x is 6.
- In 3rd instruction Process P1 is interrupted and will sleep for 3 Milliseconds so CPU will select Process P2 to resume its execution from 4th instruction.
- The 4th Instruction of Process P2 is shared=y; which means the value for shared is now 4. (Logically incorrect) P2 is completed so it will be terminated
- CPU will context switch to resume execution of process P1, the 4th Instruction of Process P1 is shared=x, which means **the value for shared is now 6**. (Logically incorrect) P1 is completed so it will be terminated.
- Here so Final Value of Shared = 6 after completion of two processes P1 and P2. But when we perform 1 increment and 1 decrement then the final value of shared should be 5 only. But because of lack of process synchronization, we receive the incorrect answer here, this situation is said to be **Race Condition.**

Here if we select Process P2 first and then Process P1 then The Final Value of shared = 6 after completion of two processes P1 and P2. And if we select Process P1 first and then Process P2 then The Final Value of shared = 4 after completion of two processes P1 and P2, which is logically incorrect this situation is said to be **Race Condition.**

Two processes, P1 and P2, share the global variable called shared. At some point in its execution, P1 updates a to the value 1, and at some point, in its execution, P2 updates a to the value 2. Thus, the two tasks are in a race to increment and decrement variable shared. In this example the "loser" of the race (the process that updates last) determines the final value of shared.

Here, two processes are increment and decrement common variable 'shared'. The final value of 'shared' depends on the relative execution order of Process P1 and P2. Such situation is called **race condition or racing problem.**

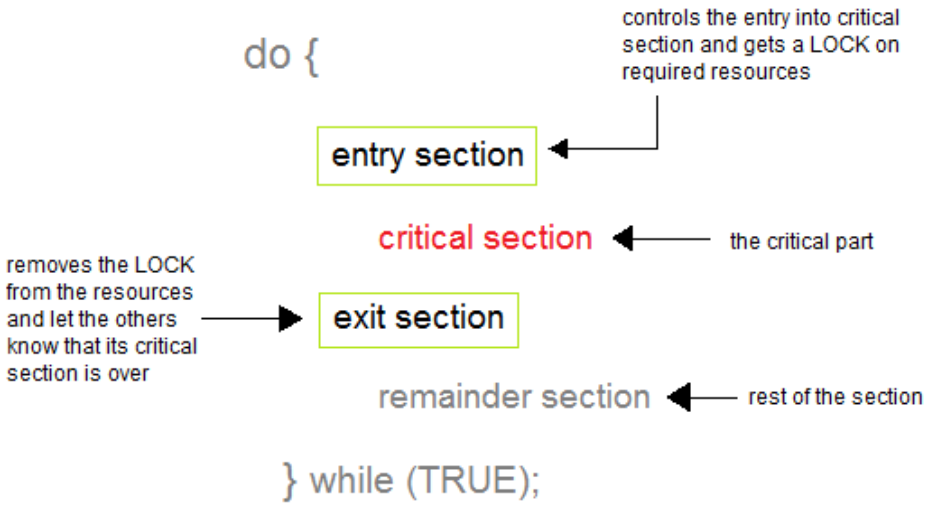
Avoiding Race Conditions:

- **Critical Section:** To avoid race condition we need **Mutual Exclusion. Mutual Exclusion** is some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same things.
- The difficulty above in the two processes occurs because process P1 started using one of the shared variables before process P2 was finished with it.
- That part of the program where the shared memory is accessed is called the **critical region or critical section.**
- If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions.

- Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.

2.1 Critical Section Problem

- A Critical Section is a code segment that accesses shared variables and must be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.



Solution to Critical Section Problem

(Rules for avoiding Race Condition) Solution to Critical section problem:

1. No two processes may be simultaneously inside their critical regions. (Mutual Exclusion)

2. No assumptions may be made about speeds or the number of CPUs.

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

4. No process should have to wait forever to enter its critical region.

A **solution** to the critical section problem must satisfy the following three conditions:

1. **Mutual Exclusion** Out of a group of cooperating processes, only one process at a time can be executing in their critical section

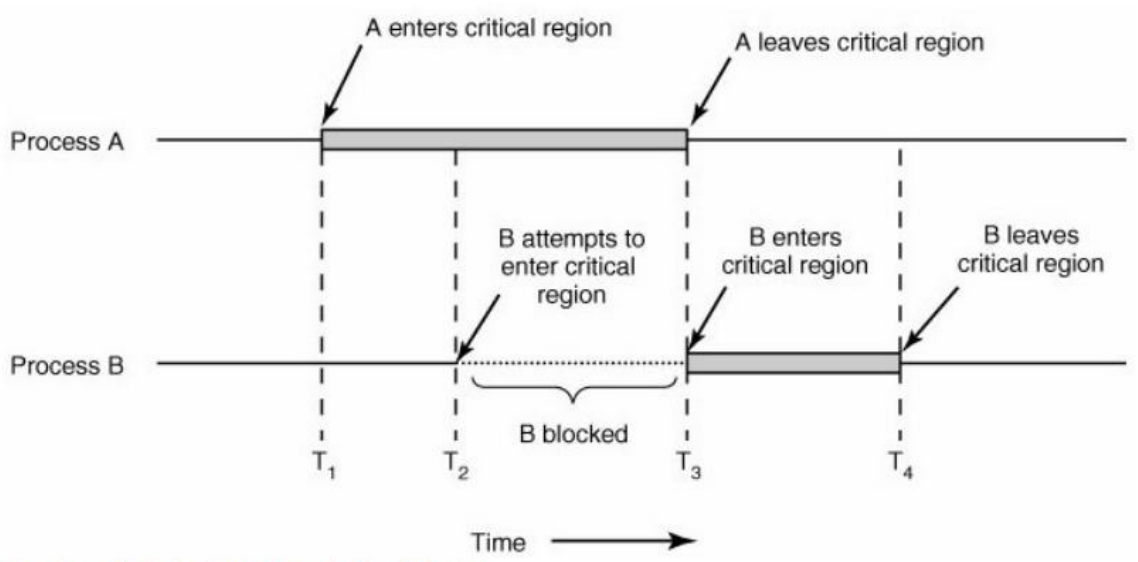


Fig:Mutual Exclusion using Critical Region

2. Progress

If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the processes not in their remainder sections can participate in the decision, and the decision cannot be postponed indefinitely. (For example, processes cannot be blocked forever waiting to get into their critical sections.)

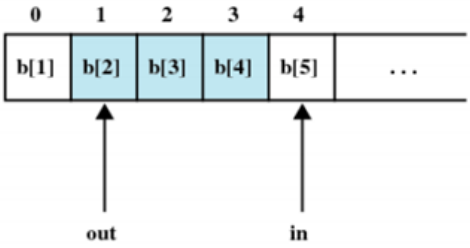
3. Bounded Waiting

There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted. (For example, a process requesting entry into their critical section will get a turn eventually and there is a limit as to how many other processes get to go first.)

2.2 Producer Consumer Problem

- The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e., synchronization between more than one processes
- In the producer-consumer problem, there is one Producer that is producing product and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed size.
- The job of the Producer is to generate the data, put it into the buffer, and again start generating data. While the job of the Consumer is to consume the data from the buffer.
- The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.

- The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
- The producer and consumer should not access the buffer at the same time.
- The **producer–consumer problem** (also known as the **bounded-buffer problem**) is a classic example of a multi-**process synchronization** problem.
- The problem describes two processes, the producer and the consumer, who share a common, fixed-size **buffer** used as a **queue**.
- The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.
- The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
- The solution for the producer is to either go to sleep or discard data if the buffer is full.
- The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer empty.
- The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
- The solution can be reached by means of inter-process communication, typically using semaphores.
- An inadequate solution could result in a deadlock where both processes are waiting to be awakened.
- The problem can also be generalized to have multiple producers and consumers.

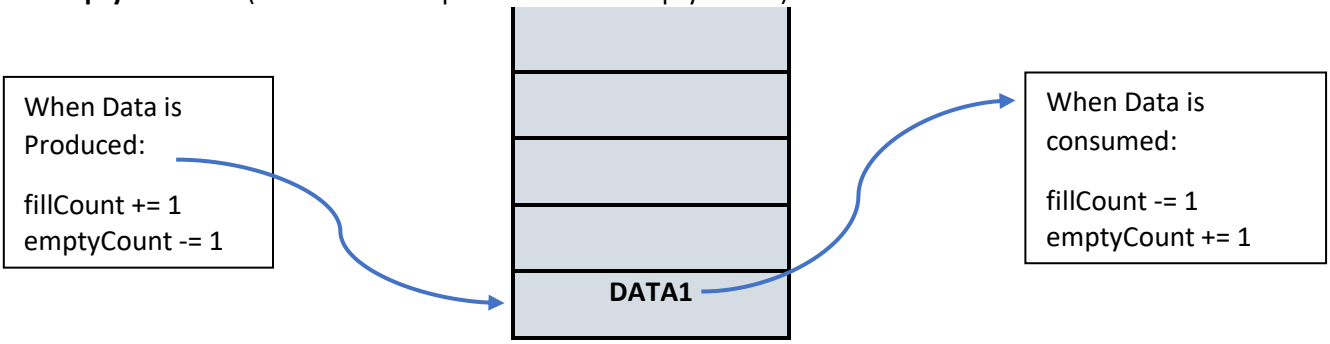


shaded area indicates portion of buffer that is occupied

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

Producer Consumer Example:

// Default Value:
fillCount = 0
emptyCount = 5 (for below example there are 5 empty buffer)



- One or more producers are generating data and placing these in a buffer
- A single consumer is taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at any one time

A solution to the producer-consumer problem satisfies the following condition.

1. A producer must not overwrite a full buffer as it will generate data overflow problem.
2. A consumer must not consume an empty buffer as it will create starvation (underflow) problem.
3. Producers and consumers must access buffers in a mutually exclusive manager.

Example of implementing producer consumer using semaphore

Semaphore fillCount=0;//items stored
Semaphore emptyCount=BUFFR_SIZE;//remaining space	

Procedure producer()
{
While(true)
{
Item=produceitem();
 	
Down(emptyCount);
 	
putItemintoBuffer(item);
 	
Up(fillCount);
}
}
	Procedure consumer()
	{
	While(true)
	{
	Down(fillCount);
	
	Item=removeItemFromBuffer();
	
	Up(emptyCount);
	
	consumeItem(item);
	}
	}

2.3 Semaphores

Dijkstra proposed a significant technique for managing concurrent processes for complex mutual exclusion problems. He introduced a new synchronization tool called Semaphore. Semaphores are of two types 1. Binary semaphore 2. Counting semaphore

1. **Binary semaphore** can take the value 0 & 1 only.
2. **Counting semaphore** can take nonnegative integer values.

Semaphore Types and its Usage

These are the uses for semaphores:

1. **Binary semaphores** can take on one of two values, 0 or 1. They can be used to solve the critical section problem and can be used as mutexes on systems that do not provide a separate mutex mechanism.

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

[Mutual-exclusion implementation with semaphores.]

2. **Counting semaphores** can take on any integer value, and are usually used to count the number remaining of some limited resource. The counter is initialized to the number of such resources available in the system, and whenever the counting semaphore is greater than zero, and then a process can enter a critical section and use one of the resources. When the counter gets to zero (or negative in some implementations), then the process blocks until another process frees up a resource and increments the counting semaphore with a signal call. (The binary semaphore can be seen as just a special case where the number of resources initially available is just one.)

• Semaphores can also be used to synchronize certain operations between processes. For example, suppose it is important that process P1 execute statement S1 before process P2 executes statement S2.

• First, we create a semaphore named **s** that is shared by the two processes and initialize it to zero.

- After initialization, it can only be accessed by 2 atomic operations as follows:
    1. Wait
    2. Signal

• Then in process P1 we insert the code:

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

and in process P2 we insert the code:

```
wait(s)
{
   while(s<=0); // no code block for while
    s=s-1;
}
```

• Because **s** was initialized to 0, process P2 will block on the wait until after P1 executes the call to signal.

Semaphore Implementation

Solution of critical section problem using semaphores

- P1, P2, P3, P4… P_n are the processes that wants to go to the critical section

```
Semaphore s=1; // default value 1
do
{
   wait(s)
//critical section
 signal(s)
//remaining code
}
while(T)
```

- In the above scenario, s is a semaphore variable with the default value 1.
- Process P1 enters the do .. while loop and executes wait(s) function, wait function will decrement the s value with 1, so semaphore value is now 0 (s=0).
    - Note: s=0 means process has entered critical section
- After executing wait(s) function, Process P1 enters the critical section.
- By that time, Process P2 wants to enter critical section so, P2 will enter the do .. while loop and will have to execute wait(s) function with s=0.
- When wait function receives value 0 it will go into infinite loop and will create the wait condition and therefore P2 will not be able to enter critical section.
- By the time, Process P1 has completed its execution of critical section and now will execute signal(s) function with s=0.
- Signal(s) will increment the value of s by 1, therefore s=1

- Note: s=1 means process has exited critical section.

- As soon as the value of s is incremented by 1, process P2 will again execute wait(s) function and again decrement the s value to 0 and will enter the critical section.
- By that time Process P1 will execute the remaining code and will be terminated.
- After finishing the critical section execution Process P2 will execute signal(s) function and increment the s value by 1 so Process P3 can now enter the critical section.
- The above steps will be repeated until the process P_n is terminated.
- Key to the success of semaphores is that the wait and signal operations be atomic, that is no other process can execute a wait or signal on the same semaphore at the same time. (Other processes could be allowed to do other things, including working with other semaphores, they just can't have access to this semaphore.)

Drawback of Semaphore

1. They are essentially shared global variables.
2. Access to semaphores can come from anywhere in a program.
3. There is no control or guarantee of proper usage.
4. There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
5. They serve two purposes, mutual exclusion, and scheduling constraints.

2.4 Monitors

- Monitor is a programming language concept used to provide process synchronization.
- Hoare and Hansen introduced the concept of monitor in 1974-75.
- A **monitor** is a collection of —
  - **Variables**: representing shared data,
  - **Procedures**: representing 'a set of atomic operations on shared data, and
  - **Condition Variables:** controlling progress of a process.
- Different languages have their own syntax to implement monitors.

Though, to get understanding of the monitor, the general syntax of a monitor can be considered as given in following figure.

```
Monitor Monitor_Name
{
        // Shared variable declarations
        // Condition variable declarations
        procedure P1 (...)
        {
                // body of procedure P1
        }
        procedure P2 (...)
        {
                // body of procedure P2
        }
}
```

A process cannot access monitor's data (variables) directly. To access monitor's data, a process has to call a procedure in a monitor, Only a procedure defined in monitor can access monitor's internal data, i.e. shared data. Procedures defined outside of a monitor cannot access monitor's internal data. Here, procedure is similar to a function or method supported by programming languages.

2.5 Inter Process Communication

Inter-process communication (IPC) means the processes to communicate with each other while they are running. IPC allows processes to synchronize their action without sharing the same address space. Messages provide a basic capability between two processes. Inter-process communication is best provided by a message passing system.
IPC is particularly useful in a distributed environment where the communicating processes may reside on different computers connected with a network.

2.5.1 Message Passing System

Message passing system requires the synchronization and communication between the two processes. Message passing used as a method of communication in microkernels. Message passing systems come in many forms. Messages sent by a process can be either fixed or variable size. The actual function of message passing is normally provided in the form of a
pair of primitives.
a) Send (destination_name, message)
b) Receive (source_name, message)
Send primitive is used for serding a message to destination. Process sends information in the form of a message to another process designated by a destination. A process receives information by executing the receive primitive, which indicates the source of the sending process and the message.

Design characteristics of message system for IPC.
1. Synchronization between the process
2. Addressing.
3. Format of the message.
4. Queueing discipline.

2.5.2 Synchronization

The communication of a message between two processes implies some
of Synchronization between the two processes. Sender and receiver
blocking or nonblocking. Three combinations are possible using blocking and
nonblocking.

- Blocking send, blocking receive
- Nonblocking send, blocking receive
- Nonblocking send, nonblocking receive.

1. Blocking send, blocking receive :

Both the sender and receiver are blocked until the message is delivered. This is called Rendezvous. This combination allows for tight synchronization between processes.

2. Nonblocking send, blocking receive :

Sender may continue on, the receiver is blocked until the requested message arrives. A process that must receive a message before it can do useful work needs to be blocked until such message arrives. An example is a server process that exists to provide a service or resource to other processes.

3. Nonblocking send, nonblocking receive :

Sending process sends the message and resumes the operation. Receiver retrieves either a valid message or a null i.e. neither party is required to wait.
Nonblocking send is most suitable for natural concurrent programming tasks
Disadvantage of nonblocking send is that an error can lead to a situation in which a process repeatedly generates messages. It also consumes system resources such as buffer space, processor time etc. Nonblocking send places the burden on the programmer to determine that a message has been received. Blocking receive is suitable for many concurrent programming tasks.

2.5.3 Addressing (Naming)

Processes that want to communicate must have a way to refer each other.
The various schemes for specifying processes in send and receive primitives are of two types :
1. Direct communication
2. Indirect communication

2.5.3.1 Direct Communication

- Send and receive primitives are defined as :

Send (M, message)

Send primitive is used for sending message. Sender must specify the sender name.
In the above primitive, M is the name of the destination and message is actual data.
Here process P is sending message to process M.

Receive (P, message)

A process receives information by executing the receive primitive, which indicates the source of the sending process and the message. Here P is the name of source, which has sent the message to the process M. Fig. 2.10 shows communication between two processes.

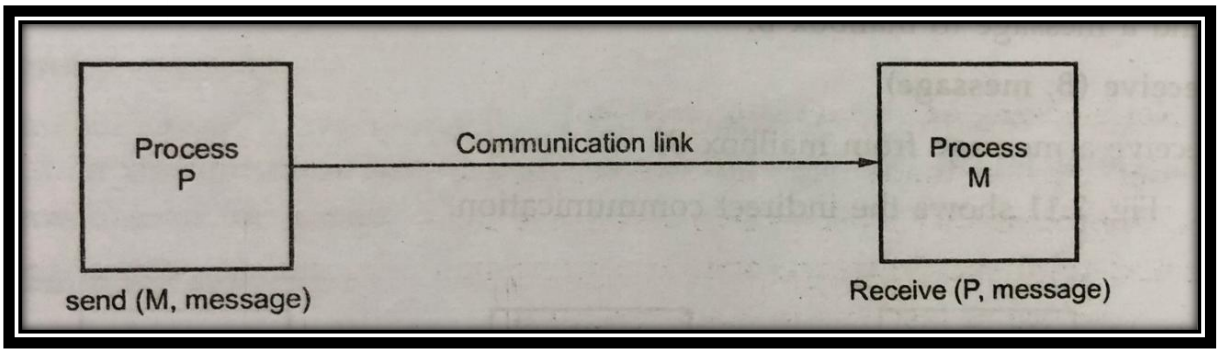


Figure: Direct Process Communication

• Both the processes are in the one network or in the different places.

Properties of communication link :

1. A link is associated with exactly two processes.
2. Exactly one link exists between each pair of processes.
3. A communication link is established automatically between every pair of the process that want to communicate.

The above explained scheme is symmetry addressing. Both the sender and receiver processes use the name for communication. Asymmetry addressing is also possible in the direct communication. The send and receive primitives are :

Send (M, message)

Send a message to process M from process P.

Receive (id, message)

Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

2.5.3.2 Indirect Communication

In indirect communication, messages are not sent directly from sender to receiver but to a shared data structure consisting of queues that can temporarily hold messages. Such queues are generally referred to as mailboxes. When the two processes are communicating, one process sends a message to the appropriate mailbox and the other process picks up the message from the mailbox. Each mailbox has a unique identification Two processes can communicate only if they share a mailbox. Indirect communication allows greater flexibility in the use of messages.

The send and receive primitives are as follows :

Send (B, message)

Send a message to mailbox B.

Receive (B, message)

Receive a message from mailbox B.

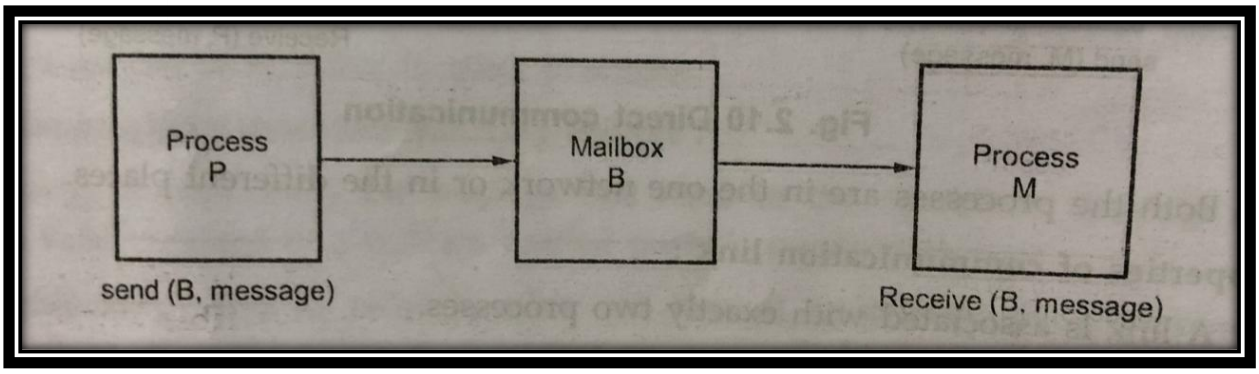


Figure: Indirect Communication

Properties of communication link:

1. A link may be associated with more than two processes.
2. A number of different links may exist between each pair of communicating process, With each link corresponding to one mailbox.
3. A link is established between a pair of processes only if both members of the pair have a shared mailbox.

A process can communicate with some other process via a number of different mailboxes. The relationship between senders and receivers can be one-to-one, one-to-many or many-to-many. Owner of the mailbox is either by process or by the operating system. Each mailbox has a unique owner so that no confusion about receiving message sent to this mail.

2.5.4 Buffering

Buffering is used in direct and indirect communication. Messages exchanged by communicating processes reside in a temporary queue. Buffering is implemented in three ways :

1. Zero capacity
2. Bounded capacity
3. Unbounded capacity

1. Zero capacity:

Maximum length of the queue is 0 (zero). Communication link can not have any messages waiting in it. In this case, the sender must block until the recipient receives message.

2. Bounded capacity:

In bounded capacity, the queue has a finite length i.e. 'n' At most 'n' messages can reside in it. If the queue is full, it discards the message and sender is blocked until space is available in the queue.

3. Unbounded capacity:

The queue has potentially infinite length. Any number of messages can wait in it.
The sender never blocks.

2.5.5 Message Format

Below figure shows a typical message format for operating systems that support variable length messages. Operating system supports variable length message and fix length messages. Fixed length messages minimize processing and storage overhead.

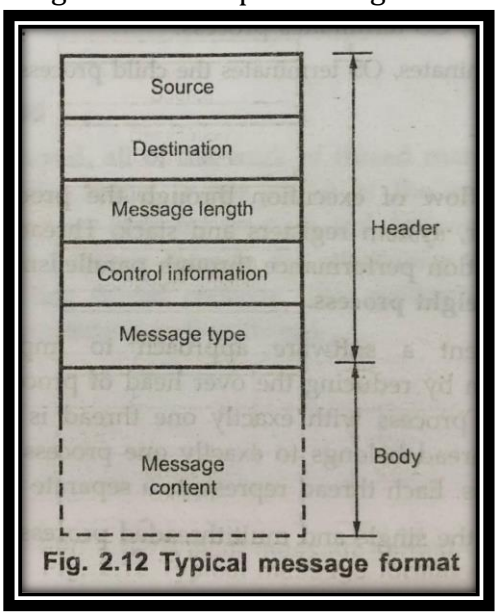


Fig. 2.12 Typical message format

• Message format is divided into two parts: Header and body. Header contains source address, destination address, message length, message type and control information. Message content is the body part.

2.6 Classical IPC Problems

The Dining Philosopher and The Sleeping Barber Problem are the two classical Inter Process Problems

2.6.1 The Dining Philosopher

Five philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. In the centre of the table is a large plate of food. A philosopher needs two forks to eat a helping of food.

Unfortunately, as philosophy is not as well paid as computing, the philosophers can only afford five forks. One fork is placed between each pair of philosophers, and they agree that each will only use the fork to his immediate right and left."
The problem is to design a set of processes for philosophers such that each philosopher can eat periodically, and none dies of hunger. A philosopher to the left or right of a dining philosopher cannot/ eat while the dining philosopher is eating, since forks are a shared resource. Figure below shows the situation of the dining philosophers.

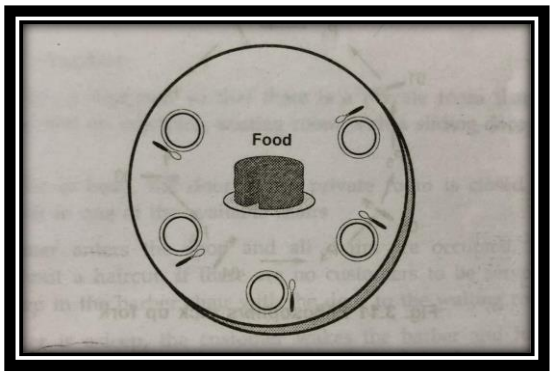


Figure Dining philosopher's arrangement

There are five philosopher processes numbered 1 through 5. Between each pair of philosophers is a fork. Fork is also numbered 1 through 5. So that fork number 3 is between philosophers 2 and 3. This is shown in the below figure.

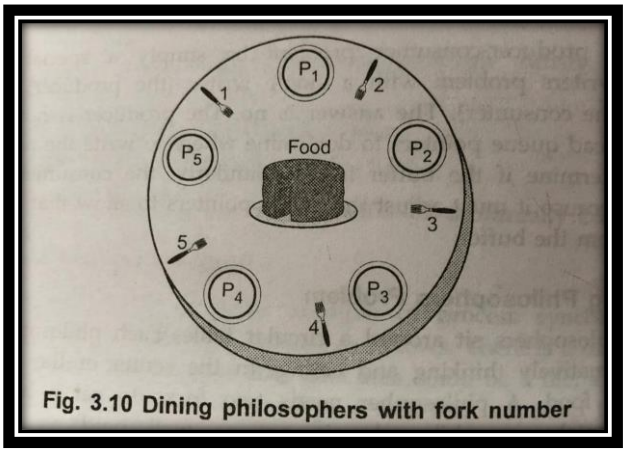


Fig. 3.10 Dining philosophers with fork number

Each philosopher alternates between thinking and eating. Solution to this problem using semaphore is shown below. Each philosopher picks up his right fork before he tried to pick up his left fork. What happens If the timing works out such that all the philosophers get hungry at the same time, and they all pick up their right forks before any of them gets a chance to try for his left fork ? Then each philosopher i will be holding fork i and waiting for fork i + 1 and they will all wait forever which is shown in Figure

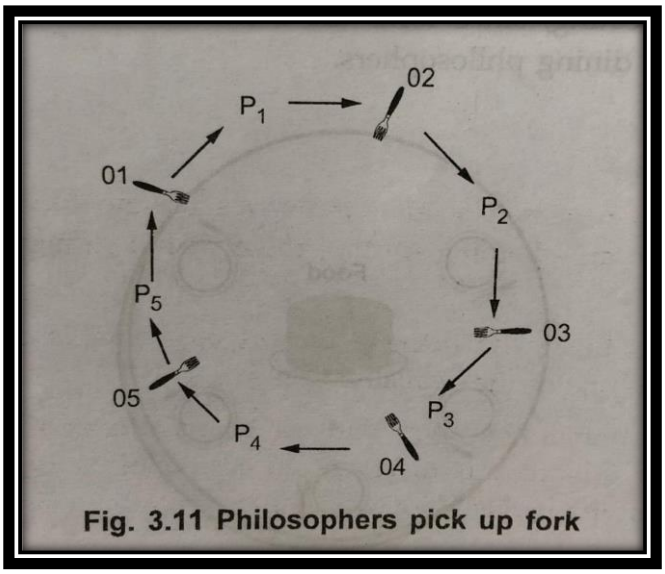


Fig. 3.11 Philosophers pick up fork

• **Following code give the solution for dining philosophers using semaphores.**

```
Semaphore eat [5] = {eating,eating,eating,eating};
Semaphore mutex = 1;
int thinking, hungry, eating;
thinking = 0;
hungry = 1;
eating = 2;
int state [5] = {thinking,thinking,tninking,thinking,thinking};

void takeforks (int j)
{
        mutex.down ( );
        state [jl = hungry;
        test [j];
        mutex.up (
        eat [jl.down ( );
}

void putforks (int j)
{
        mutex.down ( );
        state j = thinking;
test (j  == 0 ? 5: j-1);
test (j == 4 ? O : j+l);

        mutex.up ( );
}

void test (int i)
```

```
{
if (state [jl = = hunger and amp; and amp;
        state [j—ll! = eating and amp; and amp;
        state Ij+ll! = eating)
{
        state [j] = eating;
        eat [j].up ();
}
}
```

2.6.2 The Sleeping Barber Problem

• A barbershop is designed so that there is a private room that contains the barber chair and an adjoining waiting room with a sliding door that contains N chairs.
• If the barber is busy, the door to the private room is closed and arriving customers sit in one of the available chairs.
• If a customer enters the shop and all chairs are occupied, the customer leaves without a haircut. If there are no customers to be served, the barber goes to sleep in the barber chair with the door to the waiting room open.
• If the barber is asleep, the customer wakes the barber and has a hair cut. Figure below shows an implementation that uses semaphores

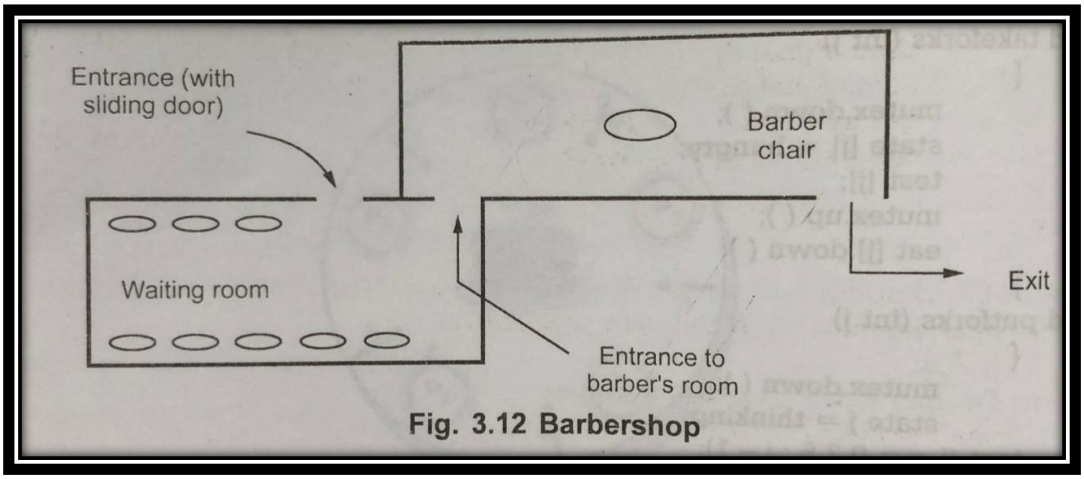


Fig. 3.12 Barbershop

• Solution to the barbershop problem using semaphore.

Global variables :

```
Semaphore door (0);  // Mutex to shared variables.
Semaphore chair (0);  // Haircut synch. , barber sleep
Semapnore q (0); // Waiting customers.
bool slepping = FALSE;  // Barber sleeping.
int waiting = 0; // # of waiting customers.
```

Barber :

```
door. signal ();
while (1)
```

```
{
        door.wait C);
        if (waiting = = 0)
        {
        sleeping = TRUE;
        door.signal ();
        chair.wait ();
        }
else
{
        q.signal ();
        -- waiting;
        door.signal ();
        cut hair; // Rendezvous.
        chair.wait (); // Synch.
}
}
```

Customer:

```
door.wait ();
if (waiting = = N)
{
        door.signal ();
        leave;
}
else if (sleeping)
{
        sleeping = FALSE;
        chair.signal ();
        door.signal ();
        get hair cut; // Rendezvous.
        chair.signal (); // Synch.
}
else
{
        ++waiting;
        door.signal ();  // Fairness problem ???
        g.wait
        get hair cut;  // Rendezvous.
        chair.signal ();   // Synch.
}
```