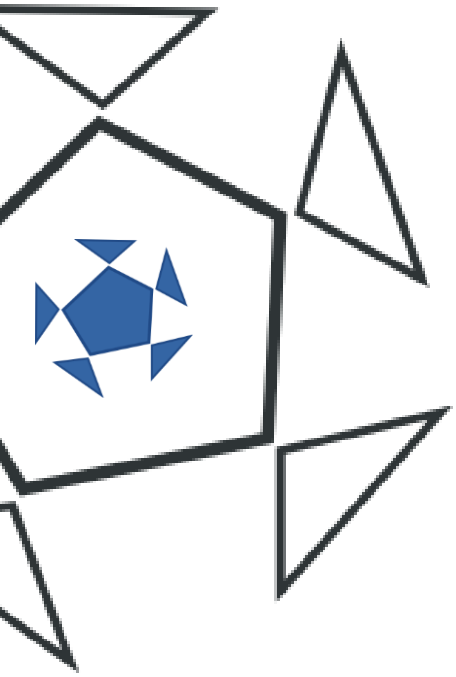
Raza's Simplified

Chapter 3



INTERPOCESS COMMUNICATOION AND SYNCHRONIZATION

In this Chapter

- Introduction
- Interprocess Communication
- Implementation of Message Passing Systems
- Problems in IPC
- Dead Locks
- Deadlock Management

Introduction

A computer can run many processes at a time. It can be a uniprocessor system or a multiprocessor one, running processes simultaneously. An OS gets more complicated if it is intended to run on a multiprocessor system.

Many issues arise in such systems. These include sharing of hardware and other resources. Many programs are written which use concurrent process [i]execution and result in better performance. An OS must be designed to handle concurrency. Such an OS must be able to:

1. Run concurrent processes
2. Handle concurrent resource sharing
3. Handle concurrent interrupts generated by the I/O devices.

These issues are also encountered by the concurrent applications.

To achieve all this, interprocess communication and process synchronization are used. These provide applications and OS with the functionality needed to run processes concurrently.

Cooperating Process

A process which is exchanging some information with another process or controlling the execution of another process is called a cooperating process. These processes are linked together and related to each other.

Methods to Share Information Among Processes

Information can be shared by:-

1. Allowing cooperating processes to share some memory locations.
2. Making a shared file.
3. Using OS special.

Interprocess Communication

Interprocess communication (IPC) is the process of exchanging information between running processes.

IPC Methods

1-It can be done using a **shared file.** This method is the most common one. Even a set of processes not running concurrently can use this method. In this method, a process writes some data in a file and later on the second one reads it. The maximum size of file is set by the file system..

2-A **shared memory** can be created. This method is much important in OS which allows threads. Many OS provide specific supervisor calls (SVCs) which generate these memory locations. Other OS allow the concept of **RAM disk** working with file system. *This is a virtual disk created in memory acting as a disk.* The file system objects, e.g. files, which are created in RAM disk, are actually created in RAM. It provides all the functions on the file in the RAM disk as if it was in a disk.

3- An OS may also support **signaling.** A signal is an interrupt sent from one process to another.

For this to be practical, two basic functions are designed:

a) Send_Signal (pid, signalId)

This function sends a signal with id= signalId, to a process with id=pid.

b) React_Signal (routine, signalId)

On receiving the signal with id signalId, the receiving process will execute the function specified as *routine*.

If signal numbers are defined in a meaningful way, then processes can easily communicate with each other.

On many OS, this system has been improved and developed into a complete messaging system for processes.

While designing such a system, following factors are considered.

1. Object Type Receiving a Message:

Many types of objects receive messages: processes, mailboxes, or communication buffers. They are identified by a name or a unique number associated with them.

2. How the Processes Link with the Messaging object:

An OS may impose a condition that all communicating processes must establish a connection with the destination before exchange of information. A communicating object can be accessed by any process or some limits might be enforced. In case a mailbox or a communication buffer is the destination, one out of three situations would result:

1. The object would be created automatically
2. Predefined objects would always exist in fixed number
3. A routine would be defined to dynamically create an object

3. Number of Processes Sharing a Messaging Object:

These limits define how many processes would use an object. Maximum number of processes reading the object, and maximum number of processes writing the object can be defined separately.

4. The Direction of Flow of Information through the Messaging Object:

A process would use a message in unidirectional or bidirectional mode. In unidirectional mode, the object can be written **or** read only. It cannot be used for both purposes. In bidirectional mode, it can be written **and** read as needed.

5. Number of Messaging objects Associated with a process:

OS can impose limits on the number of messaging objects that a process can access. There are two types of limits. First, a limit defined as the maximum number of objects allowed for a process at the system level. Second, it can be a limit defined for the number of objects that two processes would use when communicating with each other.

6. Size of messages

Messages can be fixed size or of variable size. The OS defines maximum size of the message.

7. Capacity of Messaging Object

The OS can define the message storage. It is called message buffering. It can be:

- **Nonexistent**: it means that the sending process will stop sending new messages if it has already sent one and it will wait until the recipient has received the first message. It is also called *rendezvous* scheme.
- **Bounded:** A maximum number of messages to store is set. A process sending messages continues to send new messages until the capacity is full. Now, new messages will be sent when older are received and removed from the storage.
- **Unbounded:** it means that a process can send any number of messages. Message sending is never blocked.
- **Acknowledgement based:** The sender sends a message and waits for acknowledgement signal. Till then, it stops sending new messages. After receiving this acknowledgment, the sender and receivers continue to execute.

8. Method to Send the Message

A message can be sent either by value or by reference.

4. In by value, a message is actually copied into the memory of the receiving process.
5. In by reference, the receiving process is given the address of the memory where the sent message is stored, and it is read from there.

Advantage*:* This method requires less overhead and a message can be changed even after it has been sent.

Disadvantage**:** This method violates the memory protection mechanism and processes do not have independent memory

9. Actions to take when a process has issued a signal to another process. While sender is waiting for the response, the recipient terminates before responding.

In such a case, the waiting process can either terminate or an error status is returned.

Implementation of Message Passing Systems

First Implementation

A) Int Send_Process(pid,message)

This sends a *message* to the process whose id is *pid* and this message is placed in the recipient's message queue. This allows a process to send as many messages as needed and it never blocks. Using this, any process can send a message to any other process. Unbounded number of messages can be queued for a receiving process. Messages can be of any length.

Failure: This will fail only when the recipient does not exist.

B) Int Receive_Process(pid,message)

This receives a message from a sender *pid* and copies it into *message*. A message is received in the order in which it is sent. If the message queue is empty, the recipient blocks until a message is received. pid=0 has special meanings. it is used when sender is not specified. So the recipient can receive messages from any process. This function returns the pid of the sender.

Failure: If the sending process with the specified pid does not exist then this function fails.

Second Implementation

In this scheme a mailbox is used. So messages are first put in mailbox and then they are received.

A) Int Create Mailbox(mbx)

It is used to create a new mailbox with name *mbx*.

Failure: It will fail if the mailbox with name mbx already exists.

B) Int Delete_Mailbox(mbx)

It is used to delete a mailbox named *mbx*.

Failure: **i)** It will fail if mbx does not exist, or,

ii) The process executing this is not the owner of mbx.

C) Int Send_Mailbox(mbx, message)

It sends a *message* to the mailbox *mbx*. This message is copied into the mailbox. Messages are strings and they can have any length. Sender blocks until the message is received.

Failure: This primitive fails if the mbx does not exist.

D) Int Recieve_Mailbox(mbx, message)

The message is received from the mailbox named mbx and it is put in *message*. A message is received in the order it is sent. If the queue is empty, the receiving process blocks until a message arrives.

Failure: i) This will fail if the process executing it is not the owner of the mbx, or

ii) mbx does not exist.

Third Implementation

This implementation is suited for communication between parents and their children. It uses a buffered channel called pipe. A communication channel descriptor is used in this scheme which exists between the process who created the pipe and its children. This does not use the name or id of the destination.

A) Void Create_Pipe(pdr, pdw)

Creates a pipe with read descriptor as pdr and write descriptor as pdw.

Failure: This function never fails.

B) Int Close(pd)

It closes read or write descriptor specified as pd for the process executing this routine.

Failure: It will fail if the descriptor is not open.

C) Int Send_Pipe(pdw, byte)

It writes a *byte* in the pipe using write descriptor *pdw*. Maximum size of pipe capacity is 4096 bytes. If the pipe is full, then writing is blocked until some read primitive removes some bytes from the pipe.

Failure: i) This will fail if the pdw is not open, or

ii) If no process has any read descriptor open in this pipe.

D) Int Receive_Pipe(pdr, byte)

It reads a byte from a pipe using *pdr* and stores it in location *byte*. The bytes are read in the order they are written. A special value is returned when the pipe is empty or no process has any open write descriptor in the pipe. The process will block if the pipe is empty.

Failure: If the descriptor *pdr* is not open then it will fail.

Problems in IPC

The cooperating processes face three problems:

1. Starvation
2. Deadlock
3. Data inconsistency

1- Starvation

Suppose that there is a mailbox and multiple processes are using it. The algorithm uses priority to read messages from mailbox. Suppose there are two processes, P1 having high priority and P2 having low priority. If P1is continuously generating read requests, then requests of P2 will not be satisfied for a long period of time. This is called starvation.

2- Deadlocks

They occur when two or more processes block and each of them wait for an event under control of the other. Suppose that P1 and P2 are running. P1 sends a message to P2 using mbx1 and P2 sends a message to P1 using mbx2. What will happen if P1 is waiting for a message from P2 and P2 is waiting for a message from P1 simultaneously? This situation can never be satisfied and both processes will wait forever and hence a deadlock will occur.

3- Data Inconsistency:

Suppose there is a variable named *a* and it contains 10. Two processes access it and each increments it by 1. If they access it one after the other, then variable will become 12 which is incorrect. But if both access it concurrently, then they will get *a*=10, and they will make 11 separately and store it back. The program gives correct output if a process is not allowed to access shared data when another process is updating it.

Dead Locks

Dead locks are the situations in which a conflict over sharing of resources among processes occurs. A deadlock occurs when a process is holding a resource for its needs that some other process in the set has also requested. Since every process is waiting for a resource to become available, all processes in this set stop execution and system performance declines.

Four conditions are necessary to be true simultaneously for a deadlock to occur.

1. Mutual Exclusion:

It means that the resources are not sharable. It can be allocated to only one process at a time.

2. Hold and Wait:

Every process is holding a resource and before releasing it, the process has requested more resources.

3. No Preemption:

The allocated resources to a process cannot be taken or preempted for the sake of another process.

4. Circular Wait:

Processes in the set are holding at least one resource that the next process has requested.

For example: consider a set of four processes,P1,P2,P3,P4 and four resources, R1,R2,R3 and R4. P1 is holding R1 and requesting R2, P2 is holding R2 and requesting R3, P3 is holding R3 and requesting R4, P4 is holding R4 and requesting R1.

Deadlock Management

Deadlocks are handled by performing four different activities:

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection

4. Deadlock Recovery

Deadlock Prevention

The theory behind deadlock prevention is simple. Deadlocks occur when four conditions occur together. These conditions are m**utual exclusion, hold and wait, no preemption** and c**ircular wait.** If anyone of these is not occurring, then deadlock can be prevented. But applying such a policy to eliminate at least any one of these also presents some problems. Let us analyze them.

Eliminating Mutual Exclusion

Most of the resources are not sharable. But some devices like printers can be shared intelligently if a print spooling mechanism is implemented. Spooling is derived from **SPOOL** which means **Simultaneous peripheral operations online.** When more than one user is attempting to print, the printer cannot be shared. But if spooling is implemented, then the prints from every user are not sent directly to the printer. They are placed in the spooler memory and users continue to work. The spooler sends these pints one by one to printer. The spooling can eliminate mutual exclusion for at least printers if direct access is not desired.

Eliminating Hold and Wait

To avoid Hold and wait, new requests for resources must be made by only those processes which are not holding any resource currently.

It can be achieved in two ways:

1. The processes must request all needed resources before starting execution. But it is not feasible due to two reasons:
    i. A process cannot foretell all of the resources it would need. Many resource needs are determined as the process starts and continues to execute. Its computational results decide what resources would be needed. Otherwise it would request those resources which it will never use. Also this would result in resource allocation much before actual use. It will result in low system performance and low resource utilization. **Starvation** can also occur, if a process is requesting most commonly used resources, they will not be free at the

same time, so the process will never be executed due to the lack of resources.

ii. It requires a process requesting a new resource to release all previously allocated resources. Suppose a process has produced large data and has saved it to the disk. Now it requests a print out. Since it will have to release the disk where the data is stored, how would the process print?

Eliminating No Preemption

Preemption means to suspend use of a resource by a process and allocate the resource to another process. It is possible only for those resources which allow their state to be saved. Now they can be allocated to another process. Later on, this saved state can be used to reallocate them to the first process, as it is done with CPU in context switching[ii]. But it is not possible for every device to save its state and then restore it later. Only CPU can be shared in this way. So it remains an impractical solution.

Eliminating Circular wait

To do this, every resource is given a unique priority level. If a process is requesting a resource, then it will be allocated to it if the requested resource's priority is higher than all other resources held by the process. If the priority of the requested resource is low, then a held resource having the highest priority must be released. In such a situation it resembles one of the techniques used to eliminate hold and wait. But it offers more flexibility. And if the priorities are assigned to the processes in a good and workable way, many requests of many processes are satisfied. But it is very difficult to make such priorities. On general grounds, there is no such priority which would fulfill the needs of all processes. This method remains the most practical and useful method to avoid deadlocks.

Deadlock Avoidance

Another way to deal with deadlocks is deadlock avoidance. In this scheme, a resource that would lead to a deadlock is not allocated. We do it by allocating resources only to a single process at a time.

This is not an efficient way but it does not allow the deadlock to occur. And a more practical policy would be implemented in the coming future.

Execution Sequence:

A sequence of processes in which each process runs till it terminates or blocks.

Safe Execution Sequence:

In a safe execution sequence each process runs till it is completed.

Safe State

It means a state of system in which processes run and there is no chance of deadlocks.

Unsafe State

In this state there is at least one running process and requests for resources are in a sequence that can cause deadlock. It doesn't mean that the deadlock will necessarily occur. If the OS ignores requests of resources from the unsafe execution sequence, then deadlock will be avoided.

Deadlock Detection

When a deadlock occurs it must be detected so the OS can take steps to eliminate it. Resource allocation graph and its reduction techniques are used to detect a deadlock.

Construction

A rectangle representing the system is drawn. Then all processes are represented as different circles drawn at the upper side of the rectangle. At the lower part of system rectangle, resources are drawn as small rectangle with the name of resource below it. Every resource contains thick dots equal in to the number of instances of the resource.

Now arrows are drawn.

1. a dot in the resource stretched to the process and touches the process circle with its head.(resource allocated)
2. To represent a request To represent an allocated resource to a process, an arrow whose tail starts from to use a resource, an arrow with the tail at the process circumference is drawn and it reaches the rectangle of the requested resource but its head does not enter the rectangle.(resource not allocated)

Now to reduce the graph, following 3 rules are followed:

3. If a process has arrows only in one direction (i.e. all arrows either enter the process or leave the process, means all resources are either allocated or allocation request is pending,) then erase all arrows of this process.
4. If a dot has no arrow with its tail on this dot, (means unallocated), and there is a pending request arrow for that resource, then erase this arrow.
5. Repeat this procedure until all arrows are removed or no more arrows can be removed

The system will be in safe state if there are no arrows remaining in the reduced resource allocation graph.

Deadlock Recovery

There are three different methods to recover from deadlocks

1. Automatic preemption
2. Automatic termination
3. Manual intervention

Automatic Preemption

In this method, the OS tries to free a subset of resources allocated to deadlocked processes by preempting them. But this technique must be judged against three major issues.

1. **Selection:** A process and number of its resources to be preempted is decided. Many factors are considered while making such a decision, including:
    i. The priority of process
    ii. The total time a process has already executed
    iii. Remaining execution time
    iv. Number of resources already contained by the process
    v. The number of instances of each resources that if released will break the deadlock
    vi. The number of processes being affected

2. **Consequence:** When a resource is preempted, the upcoming results of this preemption are considered.
    a. What will be the effects on the process whose resources have been preempted?
    b. Is it possible to roll back a process victim of preemption to a previous stage? This is very difficult and costly aspect. Sometimes it is not possible to rollback a process to any stage even to its starting state. The process has proceeded and some input or output has been occurred. Some inputs, especially from user are not recoverable. Input must be saved otherwise it will be lost.
3. **Starvation:** Sometimes a process takes part in repeatedly occurring deadlocks. When the first deadlock occurs, this process is selected and preempted, and then re-executed after deadlock. In the next deadlock, this process is somehow again selected and preempted. This keeps on happening and causes starvation, as such a process is rolled back repeatedly and never completes. To solve this problem, a preemption counter is introduced. Whenever a process is preempted and rolled back, the value in preemption counter is incremented. Next time, when a selection is to be made to preempt a process, the process with highest preemption counter is not preempted.

Automatic Termination

The system decides to terminate one or more or all deadlocked processes. It is very severe in its nature but it solves deadlock quickly. OS can also decide not to terminate all processes, instead a subset can be terminated. To select such a subset, the processes are judged against the automatic preemption factors.

The automatic termination can have catastrophic effects. A process's termination can affect other processes too. Due to cascading termination of parents and children, the complete ancestral tree under the terminating process will be closed. The processes seeking data from the terminating process get badly affected. If the terminating process was writing to a file, then the file will be left inconsistent.

Manual Intervention

In this method, the responsibility of resolving the deadlock is put on the operator's shoulder. It seems good because of the limitations of automatic

termination and automatic preemption but in reality, many computer systems do not have an operator at all. Also, this method is time consuming. It does not suit the needs of today's systems which need to resolve the deadlock within fraction of a second after its identification. So this method is also not much practical.

Ostrich Algorithm

One thing is unwanted in deadlock management; the methods to resolve deadlocks have one or the other drawbacks. Hence there is no practical and perfect method to resolve deadlocks. Besides, deadlocks do not occur often. So why spend so much energy on designing a deadlock free system. Rather the problem is left unsolved, just as an ostrich behaves by hiding its head in the sand.

Computer systems fail and there are many reasons for this. The cost of solution is compared to the cost of system failure before applying the solution. For example, if RAID is implemented, then the chances of data loss due to disk failures lessen. Similarly the use of a UPS allows the power failure management. Although they solve the major problems, for which they are built, but their cost is high. So we implement them only when they are really needed and their use justifies their cost. Similar approach is taken while handling deadlocks. If the strategy to implement a deadlock dealing mechanism justifies its cost, then it is used. In many cases this does not happen. So deadlocks are not managed at all. Happy deadlocking!

---

[i] Concurrent processes are those which reside together in memory.
[ii] See process attributes in chapter 2, Process Management.