The Process & Thread

Basic of Concurrency

In a single-processor, multitasking system, processes/threads are **interleaved** in time to yield the appearance of simultaneous execution. Even though actual parallel processing is not achieved and even though there is a certain amount of overhead involved in switching back and forth between processes/threads, interleaved execution provides major benefits in processing efficiency and in program structuring. In multiple processor system, it is possible not only to interleave processes/threads but to **overlap** them as well. Both techniques can be viewed as examples of concurrent processing and both present the same problems such as in sharing (global) resources e.g. global variables and in managing the allocation of resources optimally e.g. the request use of a particular I/O channel or device. The following figure try to describe the interleaving the processes. P stands for process and t is time.

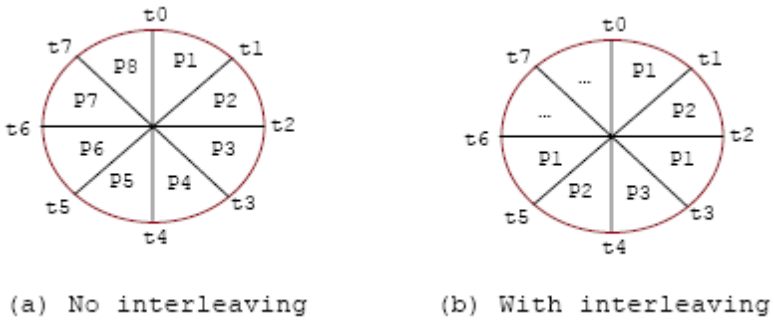


Figure 1: Interleave concept.

The following is another figure describing **interleaving** and **overlapping** as well.

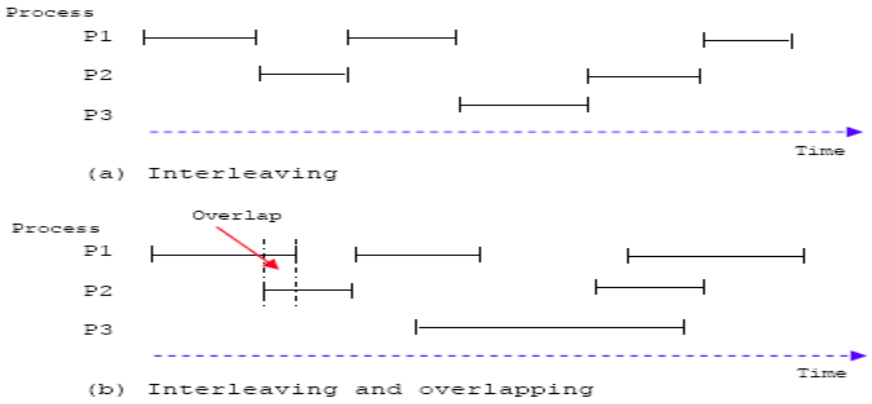


Figure 2: Process/thread Interleaving and overlapping

Process Interaction (Interprocess)

In network and distributed computing more process interaction happens. Now let consider these concurrency condition when the processes interact each other instead of a standalone process. Theoretically, from the operating system point of view, these interactions can be classified on the basis of the degree to which processes are aware of each other's existence. These have been

summarized in Table 1. Keep in mind that in implementation, several processes may exhibit aspects of both **competition** and **cooperation**.

Degree of Awareness	**Relationship**	**Influence that One Process Has on the Other**	**Potential Control Problems**
Processes unaware of each other e.g multitasking of multiple independent processes.	Competition for resources e.g. two independent applications may both want access to the same disk or file. 0perating system must regulate these accesses.	1. Results of one process independent of the action of 2. others. Timing of process may be affected.	1. 2. Mutual exclusion. 3. Deadlock (for renewable resource) Starvation.
Processes indirectly aware of each other e.g. using shared object such as I/O buffer.	Cooperation by sharing the common object.	1. Result of one process may depend on information obtained from others. 2. Timing of process may be affected.	1. 2. Mutual exclusion. Deadlock for renewable 3. resource. 4. Starvation. Data coherence.
Processes directly aware of each other e.g having communication available to them and are designed to work jointly.	Cooperation by communication.	1. Results of one process may depend on information obtained from others. 2. Timing of process may be affected.	1. 2. Deadlock for consumable resource. Starvation.

Some definitions

1. **Race condition** -In a multithreaded application, a condition that occurs when multiple threads access a data item without coordination, possibly causing inconsistent results, depending on which thread reaches the data item first.
2. **Deadlock** -In multithreaded applications, a threading problem that occurs when each member of a set of threads is waiting for another member of the set. At the end no thread get the resource and all keep waiting.
3. **Concurrency** -The ability of more than one transaction or process to access the **same data at the same time**. For the data changes in the database table's cell as an example, this issue must be handled carefully.
4. **Asynchronous call** -A call to a function that is executed separately so that the caller can continue processing instructions without waiting for the function to return.

5. **Synchronous call** -A function call that does not allow further instructions in the calling process to be executed until the function returns. There are two types of file I/O synchronization: synchronous file I/O and asynchronous file I/O. Asynchronous file I/O is also referred to as overlapped I/O.
6. **Synchronous file I/O** -A thread starts an I/O operation and immediately enters a wait state until the I/O request has completed.
7. **Asynchronous file I/O** -A thread performing asynchronous file I/O sends an I/O request to the kernel. If the request is accepted by the kernel, the thread continues processing another job until the kernel signals to the thread that the I/O operation is complete. It then interrupts its current job and processes the data from the I/O operation as necessary

Competition Among Processes for Resources

Concurrent processes come into conflict with each other when they are competing for the use of the same resource. Two or more processes need to access a resource during the course of their execution. Each process is unaware of the existence of the other processes and each is to be unaffected by the execution of the other processes. It follows from this that each process should leave the state of any resource that it uses unaffected. Examples of resource include I/O devices, memory, processor time and the clock.The execution of one process may affect the behavior of competing processes. If two processes both wish access to a single resource then one process will be allocated that resource by the operating system and the other one will have to wait. In an extreme case the blocked process may never get access to the resource and hence will never successfully terminate. In the case of competing processes, three control problems must be solved.

1. The need for **mutual exclusion**. Suppose two or more processes require access to a single non-sharable resource, such as a printer. During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data and / or receiving it. We will refer to such a resource as a**critical resource** and the portion of the program that uses it as a **critical section** of the program. It is important that only one program at a time be allowed in its critical section. We cannot simply rely on the operating system to understand and enforce this restriction because the detailed requirement may not be obvious. Well, in the case of printer, for example, we wish any individual process to have control of the printer while it prints an entire file else lines from competing processes will be interleaved :o).
2. Another control problem is a **deadlock** (permanent blocking of a set of processes that either compete for system resources or communicate with each other). Consider two processes P1 andP2 and two **critical resources**,R1 andR2. Suppose that each process needs access to both resources to perform part of its function. Then it is possible to have the following situation:R1 is assigned by the operating system to P2, andR2 is assigned to P1. Each process is waiting for one of the two resources. Well, neither will release the resource that it already owns until it has acquired the other resource and performed its critical section. Both processes are deadlocked.
3. Final control problem is **starvation**. Suppose that three processes, P1, P2 andP3, each requires periodic access to resource R. Consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource. WhenP1 exits

its critical section, either P2 orP3 should be allowed access to R. Assume that P3 is granted access and that before it completes its critical section, P1 again requires access. If P1 is granted access afterP3 has finished, and if P1 and P3 repeatedly grant access to each other, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.

Control of competition inevitably involves the operating system because it is the operating system that allocates resources.

Cooperation Among Processes by Sharing

In this case processes that interact with other processes without being explicitly aware of them. For example, multiple processes may have access to shared variables or to shared files or data bases. Processes may use and update the shared data without reference to other processes but know that other processes may have access to the same data. Thus, the processes must **cooperate** to ensure the integrity of the shared data.Because data are held on resources (device, memory) the control problems of mutual exclusion, deadlock and starvation are again present but here the data items may be accesses in two different modes, reading and writing and only writing operation must be mutually exclusive here. Consider two processesP1 andP2 are sharing data/value A. At time t0,P1 are updating data A toB, and then at t1,P2 are updating data A to C. When P1 reread its previously updated data, well, the data is not accurate anymore (C instead of B). This is also called a **race condition** and there is no data integrity for each process.

Cooperation Among Processes by Communication

Typically, communication can be characterized as consisting some sort of messages. Primitives for sending and receiving messages may be provided as part of the programming language or by the system's kernel of the operating system. Because nothing is shared between processes in the act of passing messages, mutual exclusion is not a control requirement for this sort of cooperation. However the problems of deadlock and starvation are present. As an example of deadlock, two processes may be blocked, each waiting for a communication from the other. For a starvation, consider three processes P1,P2 and P3, which exhibit the following behavior: P1 is repeatedly attempting to communicate with either P2 orP3, and P2 andP3 are both attempting to communicate with either P1. A sequence could arise in which P1 andP2 exchange information repeatedly, while P3 is blocked waiting for a communication from P1. There is no deadlock because P1 remains active, butP3 is starved.

Requirement for Mutual Exclusion

The successful implementation of concurrency among processes requires the ability to define critical sections and enforce mutual exclusion. This is fundamental for any concurrent processing scheme. Generally, any facility or capability that is to provide support for mutual exclusion should meet the following requirement:

1. Mutual exclusion must be entered: Only one process at a time is allowed into its critical section among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its non-critical section must do so without interfering with other processes.

3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely; no deadlock or starvation can be allowed.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processes.
6. A process remains inside its critical section for a finite time only.

In Windows, the mutual exclusion has been satisfied through using its APIs. We will examine the details later on.

Back To Windows OS: Synchronization

To avoid **race conditions**,**deadlocks** and other related conditions as discussed before, it is necessary to synchronize access by multiple threads to shared resources. Synchronization is also necessary to ensure that interdependent code is executed in the proper sequence.In Windows, to synchronize access to a resource, you can use one of the**synchronization object**s in one of the **wait functions**. Each synchronization object instance can be in either a **signaled** or **non-signaled state**. A thread can be **suspended** on an object in a non-signaled state; the thread is **released** when the object enters the signaled state. The mechanism is: a thread issues a **wait** request to the NT executive by using the handle of the synchronization object. When an object enters the signaled state, the NT executive releases all thread objects that are waiting on the synchronization object. The**wait functions** allow a thread to block its own execution until a specified non-signaled object is set to the signaled state. The functions described in this section provide mechanisms that threads can use to synchronize access to a resource.

The following Table lists examples that cause each object type to enter the signaled state and the effect it has on waiting threads.

Object Type	Basic definition	Set to signaled State When	Effect on Waiting Threads
Process	A program invocation, including the address space and resources required to run the program.	Last thread terminates.	All released.
Thread	An executable entity within a process.	Thread terminates.	All released.
File	An instance of an opened file or I/O device.	I/O operation completes.	All released.
Event	An announcement that a system event has occurred.	Thread sets to the event.	All released.
Semaphore	A counter (variable) that regulates the number of threads that can use a resource.	Semaphore count drops to zero.	All released.
Waitable timer	A counter that records the passage of time.	Set time arrives or time interval expires.	All released.

Table 2

[Synchronization Objects]

A synchronization object is an object whose **handle** can be specified in one of the **wait functions to coordinate** the execution of multiple threads. More than one process can have a handle to the same synchronization object, making interprocess synchronization possible. In Windows Win32 programming, the following object types are provided exclusively for synchronization.

Type	Description
Event	Notifies one or more waiting threads that an event has occurred.
Mutex	Can be owned by only one thread at a time, enabling threads to coordinate mutually exclusive access to a shared resource.
Semaphore	Maintains a count between zero and some maximum value, limiting the number of threads that are simultaneously accessing a shared resource.
Waitable timer	Notifies one or more waiting threads that a specified time has arrived.

Though available for other uses, the following objects can also be used for synchronization.

Object	Description
Change notification	Created by the FindFirstChangeNotification() function, its state is set to signaled when a specified type of change occurs within a specified directory or directory tree.
Console input	Created when a console is created. The handle to console input is returned by the CreateFile() function when CONIN$ is specified, or by the GetStdHandle() function. Its state is set to signaled when there is unread input in the console's input buffer, and set to non-signaled when the input buffer is empty.
Job	Created by calling the CreateJobObject() function. The state of a job object is set to signaled when all its processes are terminated because the specified end-of-job time limit has been exceeded.
Memory resource notification	Created by the CreateMemoryResourceNotification() function. Its state is set to signaled when a specified type of change occurs within physical memory.
Process	Created by calling the CreateProcess() function. Its state is set to non-signaled while the process is running, and set to signaled when the process terminates.
Thread	Created when a new thread is created by calling the CreateProcess(),CreateThread(), or CreateRemoteThread() function. Its state is set to non-signaled while the thread is running, and set to signaled when the thread terminates.

Table 4

In some circumstances, you can also use a **file**, **named pipe**, or **communications device** as a synchronization object; however, their use for this purpose is discouraged. Instead, use asynchronous I/O and wait on the **event object** set in the OVERLAPPED structure. It is safer to use the event object because of the confusion that can occur when multiple simultaneous

overlapped operations are performed on the same file, named pipe, or communications device. In this situation, there is no way to know which operation caused the object's state to be signaled.

Wait Functions

The wait functions allow a thread to block its own execution. The wait functions do not return until the specified criteria have been met. The type of wait function determines the set of criteria used. When a wait function is called, it checks whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the criteria to be met.There are four types of wait functions:

1. single-object.
2. multiple-object.
3. alertable.
4. registered.

Single-object Wait Functions

The SignalObjectAndWait(),WaitForSingleObject(), and WaitForSingleObjectEx() functions require a handle to one synchronization object. These functions return when one of the following occurs:

1. The specified object is in the signaled state.
2. The time-out interval elapses. The time-out interval can be set to **INFINITE** to specify that the wait will not time out.

The SignalObjectAndWait() function enables the calling thread to atomically set the state of an object to signaled and wait for the state of another object to be set to signaled.

Multiple-object Wait Function

The WaitForMultipleObjects(),WaitForMultipleObjectsEx(),MsgWaitForMultipleObjects(), and MsgWaitForMultipleObjectsEx() functions enable the calling thread to specify an array containing one or more synchronization object handles. These functions return when one of the following occurs:

1. The state of any one of the specified objects is set to signaled or the states of all objects have been set to signaled. You control whether one or all of the states will be used in the function call.
2. The time-out interval elapses. The time-out interval can be set to **INFINITE** to specify that the wait will not time out.

The MsgWaitForMultipleObjects() andMsgWaitForMultipleObjectsEx() function allow you to specify input event objects in the object handle array. This is done when you specify the type of input to wait for in the thread's input queue.For example, a thread could use MsgWaitForMultipleObjects() to block its execution until the state of a specified object has been set to signaled and there is mouse input available in the thread's input queue. The thread can use the GetMessage() or PeekMessage() function to retrieve the input.

When waiting for the states of all objects to be set to signaled, these multiple-object functions do not modify the states of the specified objects until the states of all objects have been set signaled. For example, the state of a mutex object can be signaled, but the calling thread does not get ownership until the states of the other objects specified in the array have also been set to

signaled. In the meantime, some other thread may get ownership of the mutex object, thereby setting its state to non-signaled.

Alertable Wait Functions

The MsgWaitForMultipleObjectsEx(),SignalObjectAndWait(),WaitForMultipleObjectsEx(), and WaitForSingleObjectEx() functions differ from the other wait functions in that they can optionally perform an alert-able wait operation. In an alertable wait operation, the function can return when the specified conditions are met, but it can also return if the system queues an I/O completion routine or an APC for execution by the waiting thread.

Registered Wait Functions

The RegisterWaitForSingleObject() function differs from the other wait functions in that the wait operation is performed by a thread from the **thread pool**. When the specified conditions are met, the callback function is executed by a worker thread from the thread pool.By default, a registered wait operation is a multiple-wait operation. The system resets the timer every time the event is signaled (or the time-out interval elapses) until you call the UnregisterWaitEx() function to cancel the operation. To specify that a wait operation should be executed only once, set the dwFlags parameter of RegisterWaitForSingleObject() to WT_EXECUTEONLYONCE.

Wait Functions and Synchronization Objects

The wait functions can modify the states of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the function to return. Wait functions can modify the states of synchronization objects as follows:

1. The count of a semaphore object decreases by one, and the state of the semaphore is set to non-signaled if its count is zero.
2. The states of mutex, auto-reset event, and change-notification objects are set to non-signaled.
3. The state of a synchronization timer is set to non-signaled.
4. The states of manual-reset event, manual-reset timer, process, thread, and console input objects are not affected by a wait function.

Wait Functions and Creating Windows

You have to be careful when using the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses a wait function with no time-out interval, the system will deadlock. Two examples of code that indirectly creates windows are DDE and theCoInitialize()function. Therefore, if you have a thread that creates windows, useMsgWaitForMultipleObjects() or MsgWaitForMultipleObjectsEx(), rather than the other wait functions.

Inter Process Communication (IPC)

A process can be of two type:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilised for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. 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:

1. Shared Memory
2. Message passing

The Figure 1 below shows a basic structure of communication between processes via shared memory method and via message passing.

An operating system can implement both method of communication. First, we will discuss the shared memory method of communication and then message passing. Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.
Let's discuss an example of communication between processes using shared memory method.

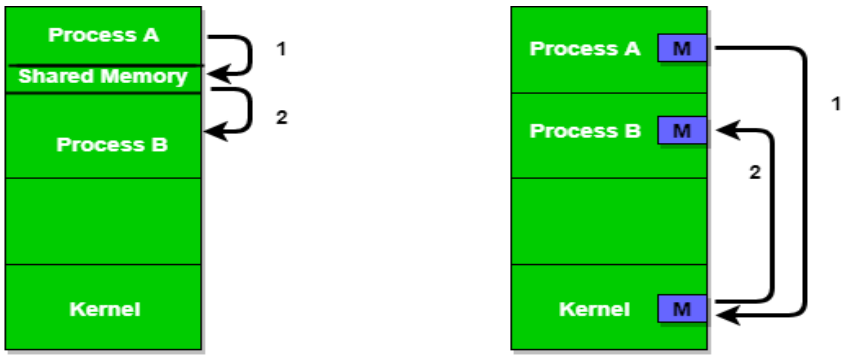


Figure 1 - Shared Memory and Message Passing

i) Shared Memory Method

Ex: Producer-Consumer problem

There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed. There are two version of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it. We will

discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer. Similarly, the consumer first check for the availability of the item and if no item is available, Consumer will wait for producer to produce it. If there are items available, consumer will consume it. The pseudo code are given below:

Shared Data between the two Processes

```
#define buff_max 25
#define mod %

   struct item{
      // different member of the produced data
      // or consumed data
      ---------
   }
   // An array is needed for holding the items.  // This is the shared place which will be
   // access by both process    // item shared_buff [ buff_max ];
   // Two variables which will keep track of
   // the indexes of the items produced by producer
   // and consumer The free index points to
   // the next free index. The full index points to
   // the first full index.
   int free_index = 0;
   int full_index = 0;
```

Producer Process Code

```
item nextProduced;
   while(1){
      // check if there is no space
      // for production.// if so keep waiting.
      while((free_index+1) mod buff_max == full_index);
      shared_buff[free_index] = nextProduced;
      free_index = (free_index + 1) mod buff_max;  }
```

Consumer Process Code

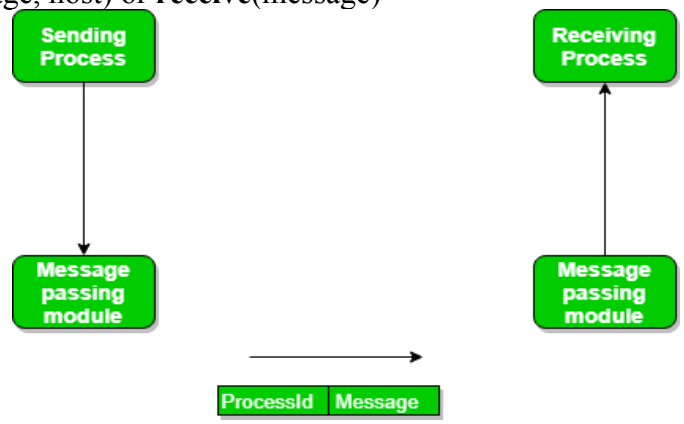
```
item nextConsumed;
   while(1){
      // check if there is an available   // item  for consumption.
      // if not keep on wait// get them produced.
      while((free_index == full_index);
      nextConsumed = shared_buff[full_index];
      full_index = (full_index + 1) mod buff_max;  }
```

In the above code, The producer will start producing again when the (free_index+1) mod buff max will be free because if it it not free, this implies that there are still items that can be consumed by the Consumer so there is no need to produce more. Similarly, if free index and full index points to the same index, this implies that there are no item to consume.

ii) Messaging Passing Method

Now, We will start our discussion for the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives.
  We need at least two primitives:
  – **send**(message, destinaion) or **send**(message)
  – **receive**(message, host) or **receive**(message)



The message size can be of fixed size or of variable size. if it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for programmer but complicated for the OS designer. A standard message can have two parts: **header and body.**

The **header part** is used for storing Message type, destination id, source id, message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Message Passing through Communication Link.

Direct and Indirect Communication link

Now, We will start our discussion about the methods of implementing communication link. While implementing the link, there are some questions which need to be kept in mind like :

1. How are links established?
2. Can a link be associated with more than two processes?
3. How many links can there be between every pair of communicating processes?
4. What is the capacity of a link? Is the size of a message that the link can accommodate fixed or variable?
5. Is a link unidirectional or bi-directional?

A link has some capacity that determines the number of messages that can reside in it temporarily for which Every link has a queue associated with it which can be either of zero capacity or of bounded capacity or of unbounded capacity. In zero capacity, sender wait until receiver inform sender that it has received the message. In non-zero capacity cases, a process does not know whether a message has been received or not after the send operation. For this, the sender must communicate to receiver explicitly. Implementation of the link depends on the situation, it can be either a Direct communication link or an In-directed communication link.

Direct Communication links are implemented when the processes use specific process identifier for the communication but it is hard to identify the sender ahead of time.
For example: the print server.

In-directed Communication is done via a shred mailbox (port), which consists of queue of messages. Sender keeps the message in mailbox and receiver picks them up.
Message Passing through Exchanging the Messages.
Synchronous and Asynchronous Message Passing:
A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system. In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so Message passing may be blocking or non-blocking. Blocking is considered **synchronous** and **blocking send** means the sender will be blocked until the message is received by receiver. Similarly, **blocking receive** has the receiver block until a message is available. Non-blocking is considered **asynchronous** and Non-blocking send has the sender sends the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null. After a careful analysis, we can come to a conclusion that, for a sender it is more natural to be non-blocking after message passing as there may be a need to send the message to different processes But the sender expect acknowledgement from receiver in case the send fails. Similarly, it is more natural for a receiver to be blocking after issuing the receive as the information from the received message may be used for further execution but at the same time, if the message send keep on failing, receiver will have to wait for indefinitely. That is why we also consider the other possibility of message passing. There are basically three most preferred combinations:

- Blocking send and blocking receive
- Non-blocking send and Non-blocking receive
- Non-blocking send and Blocking receive (Mostly used)

In Direct message passing, The process which want to communicate must explicitly name the recipient or sender of communication.
e.g. **send(p1, message)** means send the message to p1.
similarly, **receive(p2, message)** means receive the message from p2.
In this method of communication, the communication link get established automatically, which can be either unidirectional or bidirectional, but one link can be used between one pair of the sender and receiver and one pair of sender and receiver should not possess more than one pair of link. Symmetry and asymmetry between the sending and receiving can also be implemented i.e. either both process will name each other for sending and receiving the messages or only sender will name receiver for sending the message and there is no need for receiver for naming the sender for receiving the message.The problem with this method of communication is that if the name of one process changes, this method will not work.
In Indirect message passing, processes uses mailboxes (also referred to as ports) for sending and receiving messages. Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes. Each pair of processes can share several communication links and these link may be unidirectional or bi-directional. Suppose two process want to communicate though Indirect message passing, the required operations are: create a mail

box, use this mail box for sending and receiving messages, destroy the mail box. The standard primitives used are : **send(A, message)** which means send the message to mailbox A. The primitive for the receiving the message also works in the same way e.g. **received (A, message)**. There is a problem in this mailbox implementation. Suppose there are more than two processes sharing the same mailbox and suppose the process p1 sends a message to the mailbox, which process will be the receiver? This can be solved by either forcing that only two processes can share a single mailbox or enforcing that only one process is allowed to execute the receive at a given time or select any process randomly and notify the sender about the receiver. A mailbox can be made private to a single sender/receiver pair and can also be shared between multiple sender/receiver pairs. Port is an implementation of such mailbox which can have multiple sender and single receiver. It is used in client/server application (Here server is the receiver). The port is owned by the receiving process and created by OS on the request of the receiver process and can be destroyed either on request of the same receiver process or when the receiver terminates itself. Enforcing that only one process is allowed to execute the receive can be done using the concept of mutual exclusion. **Mutex mailbox** is create which is shared by n process. Sender is non-blocking and sends the message. The first process which executes the receive will enter in the critical section and all other processes will be blocking and will wait.
Now, lets discuss the Producer-Consumer problem using message passing concept. The producer place items (inside messages) in the mailbox and the consumer can consume item when at least one message present in the mailbox. The code are given below:

Producer Code

```
void Producer(void){
      int item;
      Message m;
      while(1){
         receive(Consumer, &m);
         item = produce();
         build_message(&m , item ) ;
         send(Consumer, &m);
      }}
```

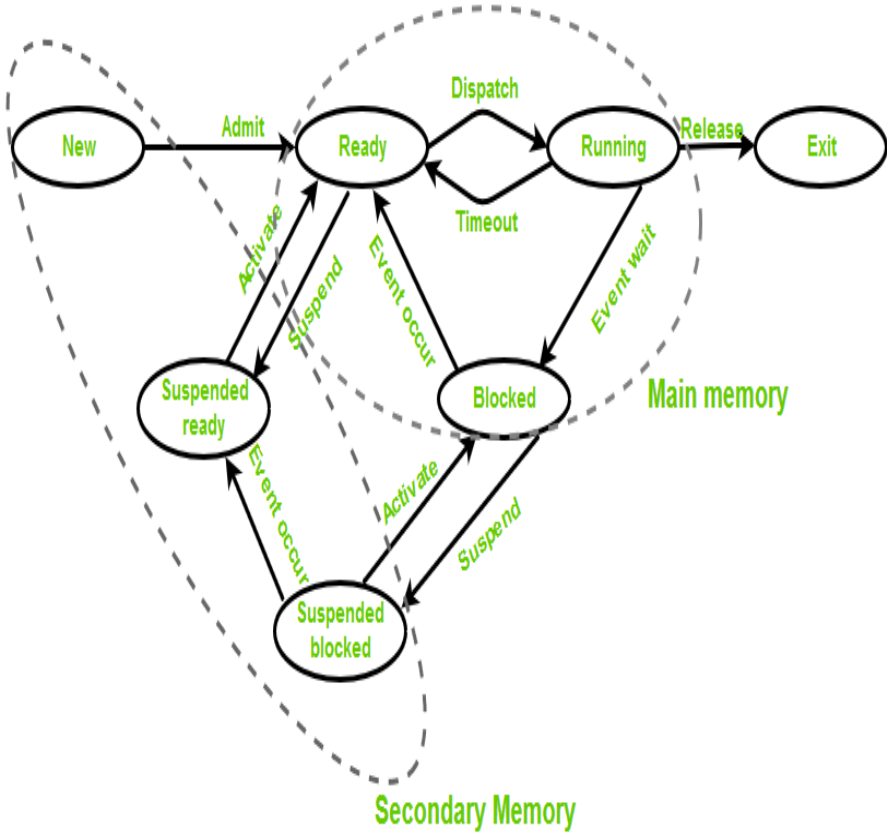
Consumer Code

```
filter_none
edit
play_arrow
brightness_4

void Consumer(void){
      int item;
      Message m;
      while(1){

         receive(Producer, &m);
         item = extracted_item();
         send(Producer, &m);
         consume_item(item);
      }
   }
```

States of a Process



- **New (Create) –** In this step, the process is about to be created but not yet created, it is the program which is present in secondary memory that will be picked up by OS to create the process.
- **Ready –** New -> Ready to run. After the creation of a process, the process enters the ready state i.e. the process is loaded into the main memory. The process here is ready to run and is waiting to get the CPU time for its execution. Processes that are ready for execution by the CPU are maintained in a queue for ready processes.
- **Run –** The process is chosen by CPU for execution and the instructions within the process are executed by any one of the available CPU cores.
- **Blocked or wait –** Whenever the process requests access to I/O or needs input from the user or needs access to a critical region(the lock for which is already acquired) it enters the blocked or wait state. The process continues to wait in the main memory and does not require CPU. Once the I/O operation is completed the process goes to the ready state.
- **Terminated or completed –** Process is killed as well as PCB is deleted.
- **Suspend ready –** Process that was initially in the ready state but were swapped out of main memory(refer Virtual Memory topic) and placed onto external storage by scheduler are

said to be in suspend ready state. The process will transition back to ready state whenever the process is again brought onto the main memory.

- **Suspend wait or suspend blocked –** Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory.
  When work is finished it may go to suspend ready.

CPU and IO Bound Processes:
If the process is intensive in terms of CPU operations then it is called CPU bound process. Similarly, If the process is intensive in terms of I/O operations then it is called IO bound process.

Types of schedulers:

1. **Long term – performance –** Makes a decision about how many processes should be made to stay in the ready state, this decides the degree of multiprogramming. Once a decision is taken it lasts for a long time hence called long term scheduler.
2. **Short term – Context switching time –** Short term scheduler will decide which process to be executed next and then it will call dispatcher. A dispatcher is a software that moves process from ready to run and vice versa. In other words, it is context switching.
3. **Medium term – Swapping time –** Suspension decision is taken by medium term scheduler. Medium term scheduler is used for swapping that is moving the process from main memory to secondary and vice versa.

Multiprogramming – We have many processes ready to run. There are two types of multiprogramming:

1. **Pre-emption –** Process is forcefully removed from CPU. Pre-emption is also called as time sharing or multitasking.
2. **Non pre-emption –** Processes are not removed until they complete the execution.

Degree of multiprogramming –
The number of processes that can reside in the ready state at maximum decides the degree of multiprogramming, e.g., if the degree of programming = 100, this means 100 processes can reside in the ready state at maximum.

Preemptive and Non-Preemptive Scheduling

1. Preemptive Scheduling:

Preemptive scheduling is used when a process switches from running state to ready state or from waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for the limited amount of time and then is taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in ready queue till it gets next chance to execute.

Algorithms based on preemptive scheduling are

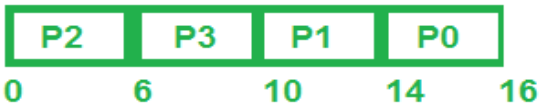
Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4

Preemptive Scheduling

2. Non-Preemptive Scheduling:

Non-preemptive Scheduling is used when a process terminates, or a process switches from running to waiting state. In this scheduling, once the resources (CPU cycles) is allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state. In case of non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits till the process complete its CPU burst time and then it can allocate the CPU to another process.
Algorithms based on non-preemptive scheduling are:

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



Non-Preemtive Scheduling

Key Differences Between Preemptive and Non-Preemptive Scheduling:

1. In preemptive scheduling the CPU is allocated to the processes for the limited time whereas in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to waiting state.
2. The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and wait till its execution.
3. In Preemptive Scheduling, there is the overhead of switching the process from ready state to running state, vise-verse, and maintaining the ready queue. Whereas in case of non-preemptive scheduling has no overhead of switching the process from running state to ready state.
4. In preemptive scheduling, if a high priority process frequently arrives in the ready queue then the process with low priority has to wait for a long, and it may have to starve. On the other hands, in the non-preemptive scheduling, if CPU is allocated to the process having larger burst time then the processes with small burst time may have to starve.

5. Preemptive scheduling attain flexible by allowing the critical processes to access CPU as they arrive into the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.
6. The Preemptive Scheduling has to maintain the integrity of shared data that's why it is cost associative as it which is not the case with Non-preemptive Scheduling.