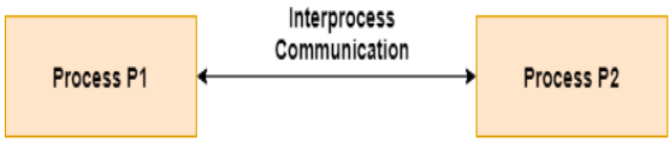
What is Interprocess Communication?

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.

A diagram that illustrates interprocess communication is as follows −



Approaches to Interprocess Communication

The different approaches to implement interprocess communication are given as follows −

Pipe

A pipe is a data channel that is unidirectional. Two pipes can be used to create a two-way data channel between two processes. This uses standard input and output methods. Pipes are used in all POSIX systems as well as Windows operating systems.

Socket

The socket is the endpoint for sending or receiving data in a network. This is true for data sent between processes on the same computer or data sent between different computers on the same network. Most of the operating systems use sockets for interprocess communication.

File

A file is a data record that may be stored on a disk or acquired on demand by a file server. Multiple processes can access a file as required. All operating systems use files for data storage.

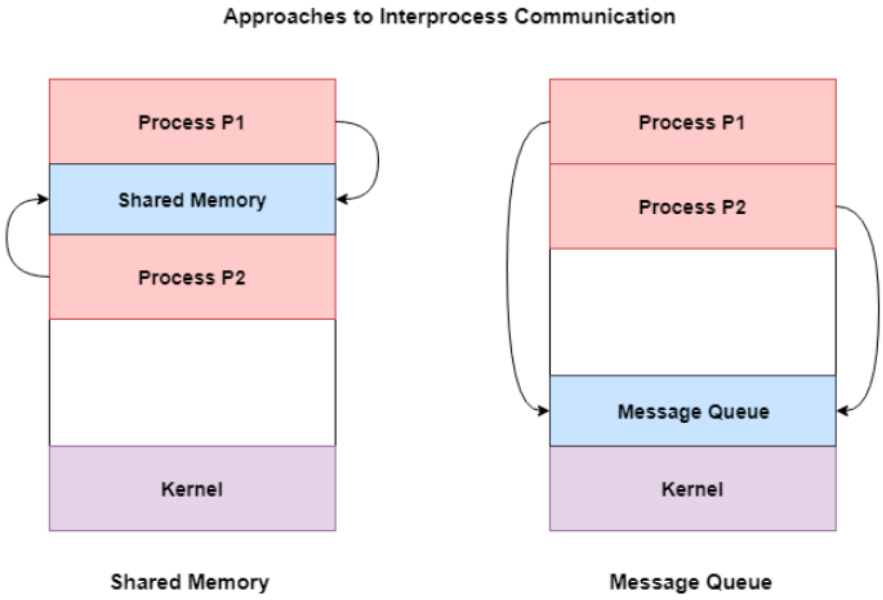
Signal

Signals are useful in interprocess communication in a limited way. They are system messages that are sent from one process to another. Normally, signals are not used to transfer data but are used for remote commands between processes.

Shared Memory
Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.

Message Queue
Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored in the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

A diagram that demonstrates message queue and shared memory methods of interprocess communication is as follows −



MULTIPLE TASKS AND MULTIPLE PROCESSES:

Tasks and Processes

Many (if not most) embedded computing systems do more than one thing that is, the environment can cause mode changes that in turn cause the embedded system to behave quite differently. For example, when designing a telephone answering machine,

We can define recording a phone call and operating the user's control panel as distinct tasks, because they perform logically distinct operations and they must be performed at very different rates. These different tasks are part of the system's functionality, but that application-level organization of functionality is often reflected in the structure of the program as well.

A process is a single execution of a program. If we run the same program two different times, we have created two different processes. Each process has its own state that includes not only its registers but all of its memory. In some OSs, the memory management unit is used to keep each process in a separate address space. In others, particularly lightweight RTOSs, the processes run in the same address space. Processes that share the same address space are often called threads.

As shown in Figure 3.1, this device is connected to serial ports on both ends. The input to the box is an uncompressed stream of bytes. The box emits a compressed string of bits on the output serial line, based on a predefined compression table. Such a box may be used, for example, to compress data being sent to a modem

The program's need to receive and send data at different rates for example, the program may emit 2 bits for the first byte and then 7 bits for the second byte will obviously find itself reflected in the structure of the code. It is easy to create irregular, ungainly code to solve this problem; a more elegant solution is to create a queue of output bits, with those bits being removed from the queue and sent to the serial port in 8-bit sets.

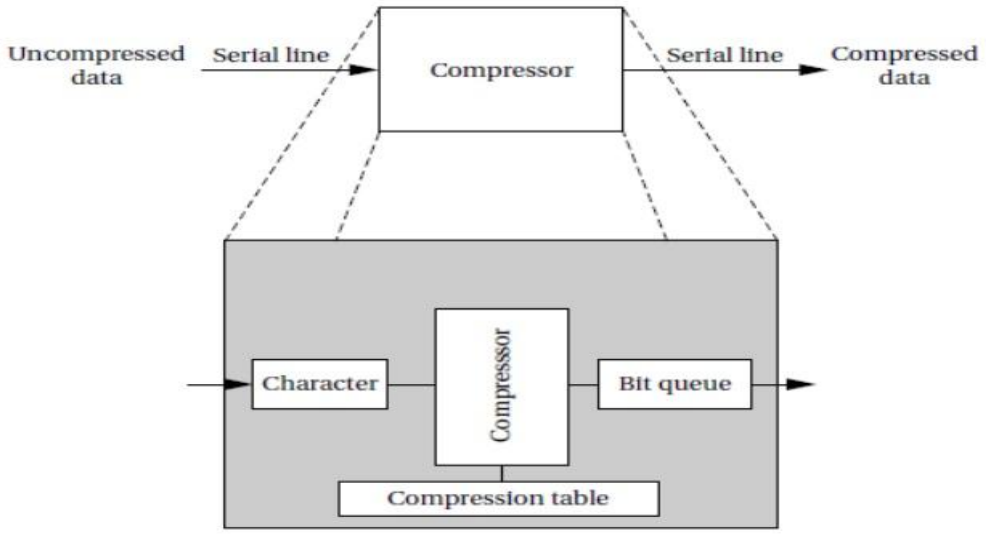


Fig 3.1 An on-the-fly compression box.

Difference between ISR and function call :

Interrupt service routine(ISR)	Function call
The interrupt is usually initiated by an internal (i.e. divided by zero, register overflow etc.) or a external signal (i.e. external pins of) microprocessor rather than the execution of instructions(i.e. software interrupt). After storing the current status of the program (i.e. value of PC ,PSW) in the stack, the ISR is executed. ISR performs different types of tasks depending on the device which interrupted or instructions written by a programmer(in the case of software interrupts).	The function call is invoked by execution of instructions, which perform the specific tasks, and also reduces the size of the program.
• The ISR address is written inside the interrupt vector table. • **For example** – In the case of 8086, the first 1KB of memory, address 00000 H ... 003FF H, is reserved for the IVT. • The ISR address for each interrupt is fixed.	The address of the subroutine is written inside the instructions which is written in the main program code.
The address of the ISR is determined by the hardware.	The address of the subroutine is written inside the main program.
ISR is used for all general purpose tasks. **For example** – If the paper in the printer is not present, then the interrupt is generated by the printer which executes an ISR(i.e. error message on the display).	Function calls are made for program specific tasks(i.e. for application specific tasks).
When an interrupt occurs during the execution of a current program, therefore, after execution of the current instruction, the processor executes ISR. After the execution of ISR, the processor must resume to program exactly as before the interrupt occurred. For this, the content of the PC, content of µP registers and the content of some status conditions is stored. The collection of all status bit conditions in a microprocessor is called PSW(program status word). During the interrupt cycle, the contents of PC and PSW are pushed onto the stack. The branch address for the particular interrupt is then passed to PC and a new PSW is loaded into the status register. The last instruction in the ISR is the return from interrupted instruction. When this instruction is executed, the old PSW and the return address are popped from the stack.	Here, only a PC is stored on the stack to get the address of the next instruction in the main program. It is necessary for the subroutine to have access to data from the calling subroutine and to return results to that subroutine. Therefore, subroutine parameters and data linkage is done. This can be done through • The AC register can be used for a single input parameter and a single output parameter. In computers with multiple processor registers, more parameters can be passed this way. • Another way to pass data to a subroutine is through the memory.

Synchronization in Interprocessor Communication :

Synchronization is an essential part of interprocess communication. It refers to a case where the data used to communicate between processors is control information. It is either given by the interprocess control mechanism or handled by the communicating processes.

It is required to maintain the correct sequence of processes and to make sure equal access to shared writable data.

Multiprocessor systems have various mechanisms for the synchronization of resources. Below are some methods to provide synchronization are as follows −

Mutual Exclusion
Semaphore
Barrier
Spinlock
Mutual Exclusion
Mutual Exclusion requires that only a single process thread can enter the critical section one at a time. This also helps synchronize and prevents the race condition by creating a stable state.

Semaphore
Semaphore is a type of variable that generally controls the access to the shared resources by several processes. Further, Semaphore is divided into two types as follows:

Binary Semaphore
A binary semaphore is limited to zero or one. It could be used to control access to one resource. In particular, it can be used to force the same release of an important category in the user code.

Counting Semaphore
Counting semaphore may take any integer value. It could be used to control access to resources having many instances.

Barrier
A barrier (as evident by its name) does not allow an individual process to proceed unless all the processes do not reach it. Many parallel languages use it, and collective routines impose barriers.

Spinlock
As evident by its name, a spinlock is a type of lock that prevents processes from operating any function unless it is available. The processes which are trying to acquire the spinlock wait in a loop while checking if the lock is available or not. This is also known as busy waiting because the process is not doing any helpful operation even though it is active.

Multiple threading :

When the first multi-tasking systems were established, they did not have a central controller. Multi-tasking was established by having programs voluntarily give up control to the system, and the system would then give control to another process. This system worked reasonably well, except that any program that was misbehaving would slow down the entire system. For instance, if a program got stuck in an infinite loop, it would never give up control, and the system would freeze.

The solution to this problem is preemptive multithreading. In a preemptive environment, control could be moved from one process to another process at any given time. The process that was "preempted" would not even know that anything had happened, except maybe there would be a larger than average delay between 2 instructions. Preemptive multithreading allows for programs that do not voluntarily give up control, and it also allows a computer to continue functioning when a single process hangs.

There are a number of problems associated with preemptive multithreading that all stem from the fact that control is taken away from one process when it is not necessarily prepared to give up control. For instance, if one process were writing to a memory location, and was preempted, the next process would see half-written data, or even corrupted data in that memory location. Or, if a task was reading in data from an input port, and it was preempted, the timing would be wrong, and data would be missed from the line. Clearly, this is unacceptable.

The solution to this new problem then is the idea of synchronization. Synchronization is a series of tools provided by the preemptive multithreaded operating system to ensure that these problems are avoided. Synchronization tools can include timers, "critical sections," and locks. Timers can ensure that a given process may be preempted, but only for a certain time. Critical sections are commands in the code that prevent the system from switching control for a certain time. Locks are commands that prevent interference in atomic operations.

ISR vs IST :

Interrupt service routines

An interrupt service routine (ISR) is a software routine that hardware invokes in response to an interrupt. ISR examines an interrupt and determines how to handle it executes the handling, and then returns a logical interrupt value. If no further handling is required the ISR notifies the kernel with a return value. An ISR must perform very quickly to avoid slowing down the operation of the device and the operation of all lower-priority ISRs.

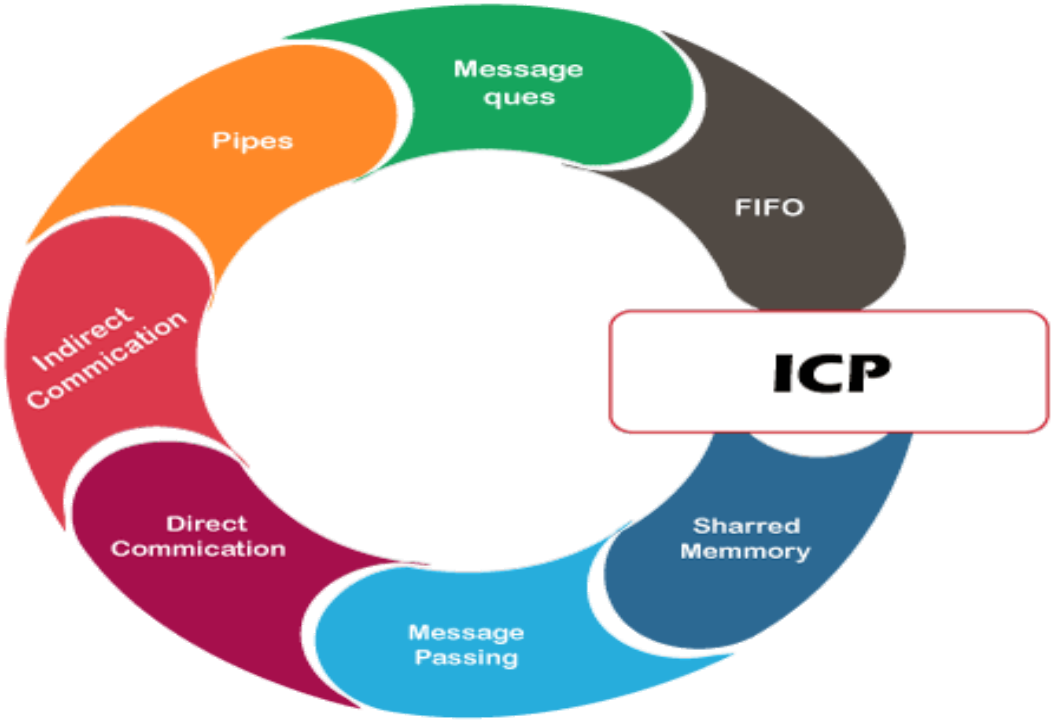
Although an ISR might move data from a CPU register or a hardware port into a memory buffer, in general it relies on a dedicated interrupt thread (or task), called the interrupt service thread (IST), to do most of the required processing. If additional processing is required, the ISR returns a logical interrupt value to the kernel. It then maps a physical interrupt number to a logical interrupt value. For example, the keyboard might be associated with hardware interrupt 4 on one device and hardware interrupt 15 on another device. The ISR translates the hardware-specific value to the standard value corresponding to the specific device.

When an ISR notifies the kernel of a specific logical interrupt value, the kernel examines an internal table to map the logical interrupt value to an event handle. The kernel then wakes the IST by signaling the event. An event is a standard synchronization object that serves as an alarm clock to wake up a thread when something interesting happens.

The interrupt service thread is a thread that does most of the interrupt processing. The operating system wakes the IST when it has an interrupt to process, otherwise, it remains idle. For the operating system to wake the IST, the IST must associate an event object with an interrupt identifier. After an interrupt is processed, the IST should wait for the next interrupt signal. This call is usually inside a loop.

When a hardware interrupt occurs, the kernel signals the event on behalf of the ISR, and then the IST performs necessary I/O operations in the device to collect the data and process them. When the interrupt processing is completed, the IST informs the kernel to re-enable the hardware interrupt.

An interrupt notification is a signal from an IST that notifies the operating system that an event must be processed. For devices that connect to a platform through intermediate hardware, the device driver for that hardware should pass the interrupt notification to the top-level device driver. Generally, the intermediate hardware’s device driver has some facility that allows another device driver to register a call-back function, which the intermediate device driver calls when an interrupt occurs.



These are a few different approaches for Inter- Process Communication:

Pipes
Shared Memory
Message Queue
Direct Communication
Indirect communication
Message Passing
FIFO
To understand them in more detail, we will discuss each of them individually

Pipe:-

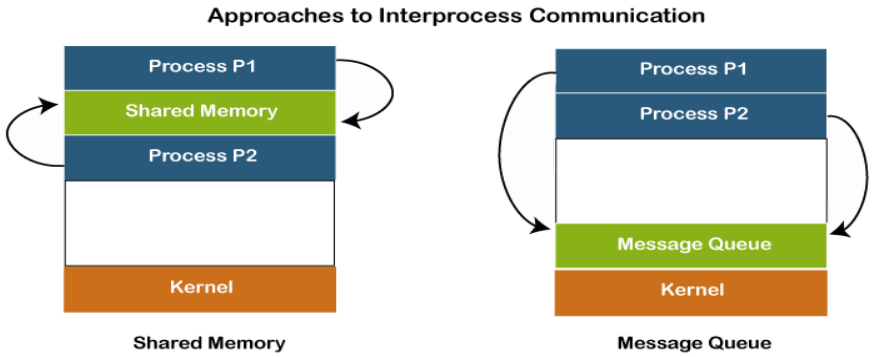
The pipe is a type of data channel that is unidirectional in nature. It means that the data in this type of data channel can be moved in only a single direction at a time. Still, one can use two-channel of this type, so that he can able to send and receive data in two processes. Typically, it uses the standard methods for input and output. These pipes are used in all types of POSIX systems and in different versions of window operating systems as well.

Shared Memory:-

It can be referred to as a type of memory that can be used or accessed by multiple processes simultaneously. It is primarily used so that the processes can communicate with each other. Therefore the shared memory is used by almost all POSIX and Windows operating systems as well.

Message Queue:-

In general, several different messages are allowed to read and write the data to the message queue. In the message queue, the messages are stored or stay in the queue unless their recipients retrieve them. In short, we can also say that the message queue is very helpful in inter-process communication and used by all operating systems.



Message Passing:-

It is a type of mechanism that allows processes to synchronize and communicate with each other. However, by using the message passing, the processes can communicate with each other without restoring the hared variables.

Usually, the inter-process communication mechanism provides two operations that are as follows:

send (message)
received (message)