Interprocess communication (IPC) is used for synchronisation, mutual exclusion and data exchange of co-operating processes in various applications. It is important that the IPC mechanism is efficient, reliable and easy to use, or else it is circumvented, resulting in ad-hoc solutions that increase the complexity and complicate maintenance.

Many computer systems run applications consisting of co-operating processes. Such system requires mechanisms for interprocess Communication (IPC). IPC involves synchronisation, mutual exclusion and/or data exchange. There is a great variety of applications where IPC is used, e.g. telecom-, robotic- and control systems. Depending on the application type, different IPC mechanisms are needed. In some applications it is sufficient to use a shared storage, e.g. RAM, for communication. Others make use of more sophisticated communication through e.g. message queues, signals etc. Common issues that a developer of applications using IPC must be aware of include (see section 6).

- Race conditions
- Priority inversion
- Deadlock
- Starvation

Not considering the above issues when designing applications may result in not (well) working software. To assist the application engineers in their design work, Operating System (OS) vendors have built in IPC functions in their Operating Systems. Many of the communication functions provided by an OS take care of race conditions, mutual exclusion and synchronisation. An example of a mechanism that is deadlock-, starvation free and supports bounded priority inversion is the priority ceiling semaphore .To prevent deadlocks, starvation etc. the application engineer must know how to use the communication functions that are provided.

An IPC operation involves at least mechanisms for IPC processing and process management .Other mechanisms involved are scheduling and process dispatching. IPC processing is the transferring and buffering of data etc. involved in IPC communication. The process manager is coping with movement of processes between different state lists e.g. moving a process from the waiting list to the ready list. A scheduler makes sure that the right process is executing and is invoked by the process manager when the process ready list has been changed. If the scheduler finds a process to switch to, the process dispatching mechanism is invoked, which includes saving and restoring process context. The mechanisms that an IPC designer must consider are IPC processing and process management. But to improve performance of an IPC operation, also scheduling and process switching are considered.

Principally, there are two ways of implementing IPC, namely in software and in hardware . Software implementations are based on the utilisation of processor instructions. Hardware implementations can be categorised as follows

IPC mechanisms

There are many different IPC mechanisms. In this section we present a subset of them. Our selection criteria have been the more common mechanisms and those that have been implemented in hardware. Before presenting the selected mechanisms in more detail, a general discussion concerning IPC mechanisms is given. IPC is used in many applications and is aimed to solve mutual exclusion, synchronisation and data exchange among co-operating processes. Due to different application needs, different IPC mechanisms have evolved. For instance, applications that often perform synchronisation, need a mechanism that is optimised for that. If the mechanism also supports data exchange, it is more likely that it doesn't provide an optimal solution to synchronisation. For applications that both need synchronisation and data exchange yet another mechanism could be the best choice. Accordingly, there are IPC mechanisms that are optimal for solving one or two, or all of the tasks an IPC mechanism is suppose to solve i.e. mutual exclusion, synchronisation and data exchange. But that is not all. There are some implementation dependent attributes, which can be associated with the primitives. For example, consider an application that needs to exchange data between a group of processes in a synchronous way. The attributes here are collective and block, which are listed below together with some other attributes.

Data exchange attributes:

Blocked. Synchronous communication.

Non-blocked. Asynchronous communication.

Partly blocked. Synchronous communication that timeouts after a specified time.

Buffered. Holds data until completion.

Non-buffered. No buffering of data i.e. receives must precede sends of data.

Reliable. Reliable communication over network i.e. data will not be lost.

Unreliable. Unreliable communication over network i.e. data may be lost.

Collective. Communication between a group of processes i.e. broadcast or multicast communication.

Point-to-point. Communication between two processes. Synchronisation and mutual exclusion attributes:

Blocked.

Non-blocked.

Partly blocked.

Collective. Synchronisation between a group of processes.

Point-to-point. Synchronisation between two processes.

Deadlock free.

2.1 Semaphores

Semaphores were invented by Dijkstra [Dijkstra67] and various IPC mechanisms are based on semaphores. There are two types of semaphores, namely binary and counting ones. The binary semaphore is either 0 (taken) or 1 (free). Mutex is another common name for a binary semaphore, which stands for mutual exclusion. A counting semaphore is 0 (taken) or any number greater than 0 (free). The two primitives to work on a semaphore are P and V (of Dutch origin) other names for the same primitives are sem_take respectively sem_give. Sem_take decrease the semaphore value with 1, if the value is greater than 0. If the value is 0, the process is put to wait. Sem_give increases the semaphore value with 1, if no processes are waiting for it. If processes are waiting for the semaphore, one is picked (by the OS) to complete the sem_take.

Semaphores are used for synchronisation and mutual exclusion. For instance a semaphore can be used to protect a printer from simultaneous requested printouts, which could result in corrupted printed text. Figure 2 shows an example of C code that uses the semaphore primitives sem_take and sem_give to protect printouts to a printer. Before any of the two processes may access the printer they must acquire semaphore P, which must be created and initialised to 1 before it is used. When respectively process has successfully acquired P they may use the printer. After finished the printout processes must also release P, making it possible for other processes to use the printer. Hence, process 1 gets preemted (for some reason by process 2) after successfully acquiring P. Now process 2 tries to acquire P, but fail since P is busy, and gets blocked. Then process 1 starts to execute the printout and finish by releasing P. Process 2 can now safely continue its execution calling the printer, since it has been granted P.

```
Process1(void){
  sem_take(P);                /*Acquire P*/
  printOut("P1 is printing");/*Call the printer*/
  sem_give(P);                /*Release P*/
}
Process2(void){
  sem_take(P);                /*Acquire P*/
  printOut("P2 is printing");/*Call the printer*/
  sem_give(P);                /*Release P*/
}
```

Figure 2: Semaphore example code.

If the printer in the example were not protected, the printouts would have been corrupted if process 1 were preemted by process 2 in the middle of a printout.