PROCESS MANAGEMENT

- real-time systems carry out a set of activities or tasks and most of these activities are interrelated and interact with one another,

- continuous variations in the nature of the environment make it difficult to predict which activities, from a set of possible activities, a real-time system will be required to carry out next,

Processes /or tasks/

- a real-time system can he seen as one that carries out a set of activities,

- in order to specify a process completely, it is necessary to define precisely what activity the process must carry out and also its interaction with its environment,

- most real-time systems must be able to perform more than one activities that may be either unrelated or related but most of activities are related,

- as a result an action of one process must take into account actions of the other processes,

- thus, there is a set of independent processes and the real time system should include interactions between processes,

- interaction of processes will reflect interactions of the components they control,

- as number of processes is increasing the situation becomes extremely complex,

Processes

- these interactions may be either

1. direct or immediate

- action of one device will cause the starting or stopping of another device's activity or

2. indirect or not-immediate

- a device may receive information about the status or actions of another device and be obliged to modify its actions accordingly to it,

- corresponding processes will need to co-ordinate their activities so that the action of one process can cause the activation or suspension of other processes,

- also, they will need to be able to communicate i.e. to send information to one another and to access information that reflects the status of the system as a whole

Implementation of Processes

- a process consists of a processor, program code and sufficient memory to accommodate this program code,

- a computer must provides these resources for as many processes as required in a real time system,

- these resources can be provided by as

1. dedicating unique resources to each process as in a distributed system, or

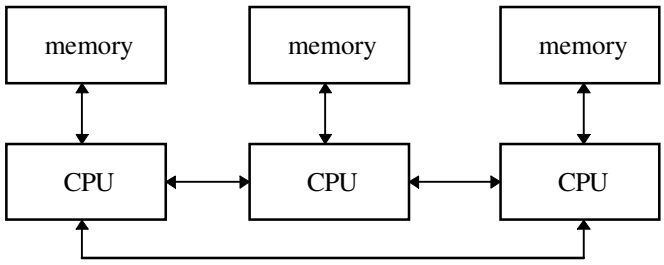
2. sharing the available resources between the processes.

dedicated or distributed resources = separate resources dedicated to each process i.e. each process in the system has a separate microprocessor, together with enough memory for the code and running the process,

- processors use their own local memory to store program code,

- they communicate with one another via communications links that are set up to allow interactions between the CPU and memory modules,

- there is a clear correspondence between the channels and the physical links between the microprocessor modules



dedicated /distributed/ memory architecture

problems of dedicated resources:

1. high costs

- often economic constraints rule out the possibility providing a unique CPU and/or memory module for each process,

- if insufficient resources are available, it becomes necessary to share

resources between processes,

2. lack of flexibility

- rigid allocation of process to CPU and/or memory implies that no new processes can be introduced into the system without altering the hardware configuration,

- some systems may require that the number of processes in the system vary with the workload on the system. A fixed CPU allocation would imply that the number of processes in the system is constant,

- this would severely constrain possible expansion. The system could not he extended to include new activities without adding further hardware.

3. low reliability

- to increase system reliability it requires to include redundant CPU and memory modules in the system,

- when a hardware component fails the system can be reconfigured to use the redundant standby modules,

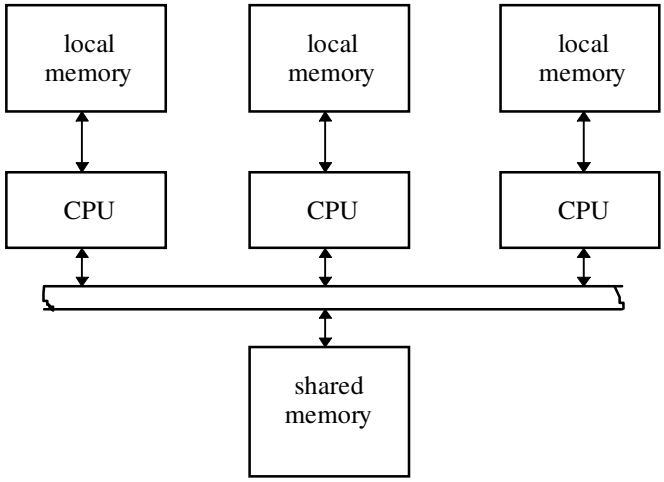
- this technique is difficult to employ if there is a rigid allocation of process to CPU/memory,

- with a distributed system it may be possible to transfer a process to another processor in the system, but this is not always possible if direct connection with external devices is involved,

<u>shared resources =</u>

- if the local memory does not contain all of the program code for a process, then it is held in the shared memory,

- in order to create a process one need only load one of the memory modules with the relevant program code element,



shared memory architecture

types of resource sharing:

1. processor sharing

- when discussing the ways in which a CPU (or CPUs fewer in number than all the processes) may be shared among the processes, it is; necessary to consider three factors

1. how the CPU is to be shared - in other words, what mechanisms are required to enable a processor executing the program code of one process to change its activity and execute the program code element of another process;

2. when the CPU is to be shared - that is, at what times, or as a result of what events, should the CPU change from executing one process to executing another;

3. which process should the CPU direct its attention to, when sharing of the CPU is necessary.

serial execution

- the simplest method of sharing a CPU for the processes to be executed without interruption, i.e. one process after another,

- this method causes the system to lose its responsiveness to environmental changes,

- it can only change the current process and thus its effect on the environment at the end of execution of the current process,

- processes must be as short as possible if the controlled system is to remain unaware of the fact that the processes are not running in parallel,

- if only one CPU is in use, the processes must not execute synchronising WAIT operations since, clearly, no other process can run to execute the corresponding SIGNAL operation,

- such processes may also communicate with each other for example one of the preceding processes has left information that is required for the current process,

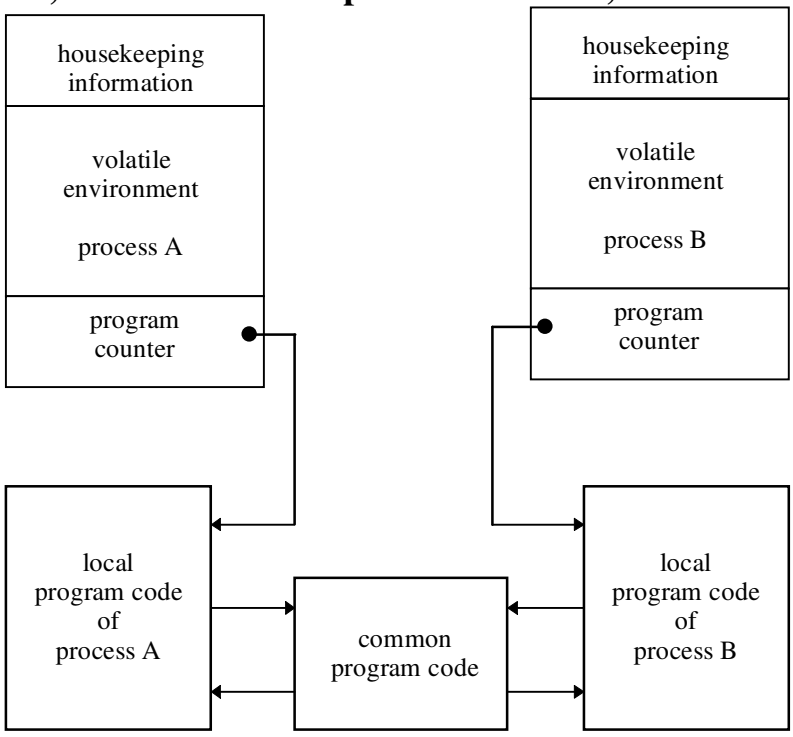
- such simple co-operative scheduling is quite suitable for many small real-time systems, thus avoiding a lot of the critical problem areas now discussed through serial execution with only limited pre-emption at the interrupt level.

pseudo-parallel execution

- the major drawback of the serial method occurs when processes take considerable time to execute, and the system therefore becomes very sluggish in its response to changes in the environment.

- ideally, in order to react to a change in the controlled system, it should be possible for a CPU to swap from one process to another at any time, giving the impression to the environment that the processes are being conducted in parallel,

- to make this possible the concepts of volatile environment process descriptor, scheduler and dispatcher are used,



process code sharing

process's volatile environment = includes such information as the

contents of hardware registers, memory management registers and the program counter,

- while the process is running on a CPU, the volatile environment is continually changing,

- if the process is denied the use of CPU for a period, while the CPU is required by another process, then the first process's volatile environment must be preserved,

- if this is not done the process will be unable to continue from where it left off,

- if this is done and the information is restored to the registers, then the process will be unaware that it ever had anything but total use of the CPU,

- there is a significant overhead in changing the environment of a process every time a change is implemented and this must be considered when designing process swapping procedures,

- a place to store a process volatile environment must be provided while it is not using the CPU and it takes the form of a data structure called process descriptor (PD),

- scheduler and dispatcher is the mechanism that effects how processes are shared,

- when dispatcher activated it stores, in the relevant process descriptor, the contents of the hardware registers that make up the volatile environment of the currently active process,

- it then replaces the contents of the registers with volatile environment information from another process's process descriptor.

<u>example of process switching</u>

- controlled system influences the attention of the software system by means of the hardware interrupt mechanism,

- if an interrupt occur, the hardware saves a portion of the current volatile environment (including at least the program counter) in a fixed location and replaces it with the volatile environment of the process designed to handle the interrupt,

- when executing the interrupt handler process is careful to ensure that it does not affect any of the unsaved portions of the interrupted process's volatile environment,

- when it has completed its action, the interrupt handler can execute a return-from-interrupt instruction, which restores to the registers the saved portions of the interrupted process's volatile environment,

- the interrupt specifies exactly which process will subsequently be activated (the interrupt handler). The *when* and *which* aspects of process swapping are indivisible concepts when the swapping occurs as a result·of an interrupt,

- on its completion, an interrupt handler process can swap the processor back to the interrupted process by executing a return-from-interrupt instruction,

- however, the occurrence of an interrupt normally implies a change in the status of the system's environment, this change may require the activation of a process other than the one that was interrupted,

- thus it is usual for the interrupt handling process to terminate by signalling the dispatcher, rather than executing a return-from-interrupt instruction,

- dispatcher then finds which process should run next and effects the necessary swapping of complete volatile environments.

2. memory sharing

- if one CPU can be shared between more than one process, it remains necessary to provide main memory to hold the process code.

- the simplest way is to provide in the system enough memory for each process to have its own unique area,

- for many systems the high cost of main memory will force a reduction of main memory capacity,

- this implies that the available memory must be shared between the processes in the system,

- there are two ways of memory sharing:

a. to share the program code that resides in a memory; or

b. to share the use of the section of memory itself.

program code sharing

- real-time systems frequently contain a number of actions that form part of more than one process in the system

- these activities are implemented as subroutines and a copy is built into the program code of any process that needs it,

- if one of these subroutines was of any considerable size, great saving in memory space could be achieved if the processes could share the code of the common routine, rather than each having a copy.

- a subroutine consists of program code and data, thus the shared routine also needs local memory areas for its temporary working variables,

I. serially re-usable code

- no assumptions about the values in its local variables when it is entered, in other words, the initial values of the local data have no effect on the routine's action,

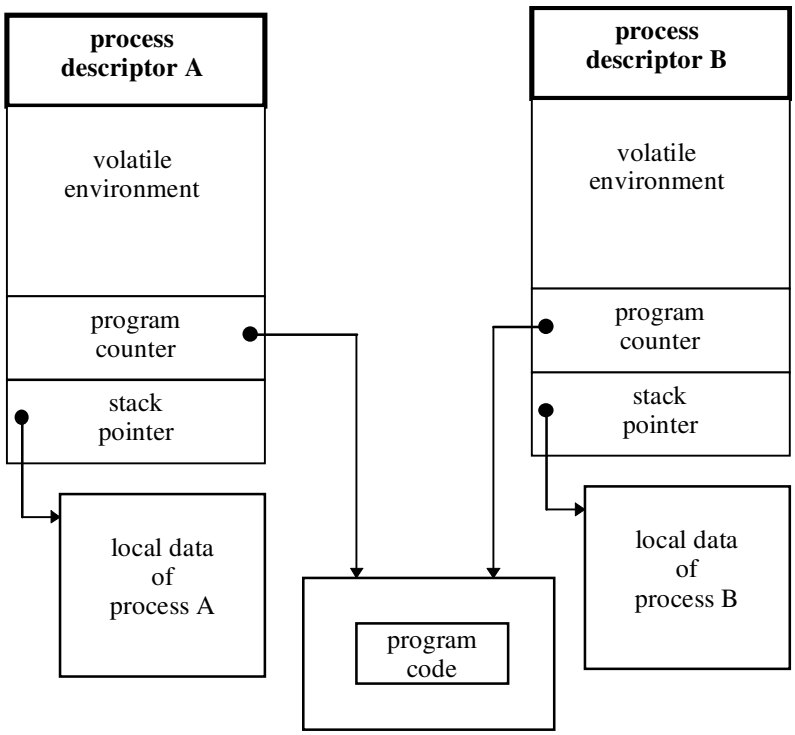
- usually, the first action of the subroutine would be to set its local variables to a fixed, initial set of values,

- a lock mechanism - WAIT operation - is applied at the beginning of the subroutine and an unlock mechanism - SIGNAL operation - at the end, such that only one process may be executing its code at any one time in order to provide that processes can safely use the code one after the other,

II. r*e-entrant or pure code*

- if all the temporary data areas needed by a subroutine S were to be part of the process currently using the subroutine S, rather than part of subroutine S itself then S would consist of executable code only,

- therefore, it could be executed by more than one process at a time, provided that the subroutine S did not modify its own instructions in any way,



processes with common program code

- if a piece of re-entrant code is to act on different data areas, which are dependent on the process using it, it must access this data via a relocation pointer which is associated with each process

- this relocation pointer containing the address of the beginning of the process data could he passed as a parameter when the subroutine is called,

- each process will call the re-entrant subroutine by transferring control to the start of the routine,

- the return address of the subroutine call is managed by a stack mechanism and the stack area must be part of the individual processes

- the amount of shared code that a process uses could be extended to the whole of the process's code

example of the re-entrant code:

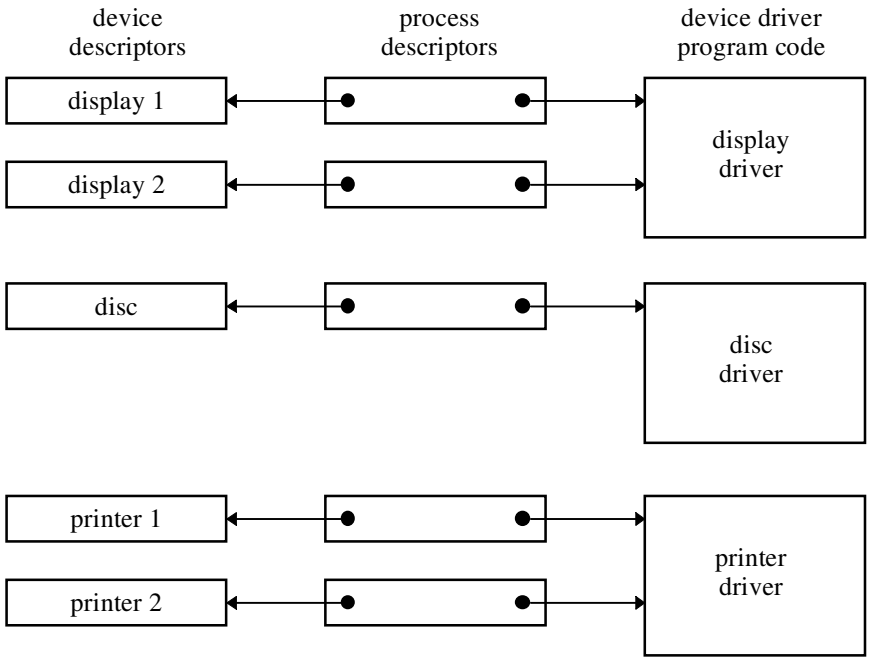
- device drivers in a conventional operating system have re-entrant code,

- device descriptors contain device-specific information such as character set and speed for a printer,

- the device driver code, process descriptor. plus a device descriptor

constitutes a process,

- to introduce a new process to handle another terminal the system programmer need only to create another device and process descriptor,



input/output-driver code and data structures

<u>memory segment sharing</u>

- even with extensive use of code sharing it will not always be possible simultaneously to accommodate all the process code in main memory,

- in this case it will be necessary to hold code and data element that are not currently being executed in a back-storage medium and to copy to the main storage when it becomes necessary to execute them,

- if a code is resident, it may be executed immediately; if not, it is non-resident and must be copied to main memory before it can be executed by the CPU.

- most real-time systems divide their main memory area into two segments:

1. the first holds frequently executed program code and the program code of highly time-critical processes, these code elements reside permanently in memory,

2. the second area is set aside as a free space to be shared between

the remaining code in the system,

- these remaining elements consist of processes that are less time-critical in nature and code and that are less frequently executed,

- relative size of the two areas will depend upon the requirements and constraints of the particular system.

- the problem of sharing memory occurs

1. when an active process wishes to execute a program code segment that is not currently in memory,

2. when a process wishes to access a data area that is not currently resident in the memory

- this event is detected by mechanisms in either the software or the hardware and the memory-sharing process is then alerted.

- memory share is solved by a set of processes, built to control the allocation of memory,

- there is a mechanism to maintain a list of all available pieces of memory, when a process requires space for its program code, a search is made through this list,

- if a memory with suitable size is found, it is allocated to that process, the required program code element is then copied into this area from back-storage

- if a higher-priority process requests a large amount of memory and finds it unavailable, a process may be called on to enlist all the available free memory space,

- if this proves inadequate then lower-priority processes may be called on to relinquish their current memory allocation, allowing the high-priority process space to run,

- if large amounts of information are to be shared between processes then files on back-storage would be most appropriate,

- if, however, real-time constraints make slow information access unacceptable, more expensive main storage will have to be used,

- a fixed block size approach overcomes the problem of memory fragmentation at the expense of some overhead.

Communication

- in order to facilitate communication between processes, information available to one process must be made available to another process,

- an item of information stored in one area of the system's memory, accessible by one process, must be made accessible to other processes,

- this means that either the data is copied to another area of memory, or the same area of memory is available to all processes wishing to communicate.

- information may be communicated between processes in two ways

1. direct transfer of information

- it may occur from one process to another

2. indirect transfer of information

- each process may access or update pieces of shared information, where this shared information is available to some or all of the processes in the system.

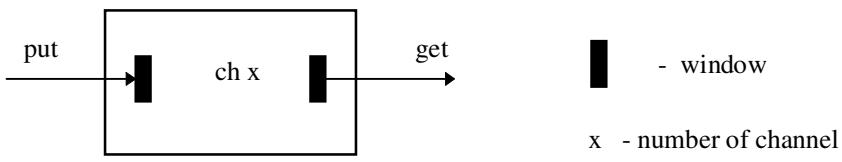
channels = provides the medium for items of information to be passed between one process and another,

- it provides a pipe of information passing from one process to another and thus it also provides a link between processes within the system,

- for the processes to run truly asynchronously there must be some buffering of information; the larger the buffers, the greater the system flexibility,

- more than one item of information can pass through a channel at any one time,

- items passing through the channel wil1 usually be ordered so that the first item passed into the channel will he the first item removed at the other end,



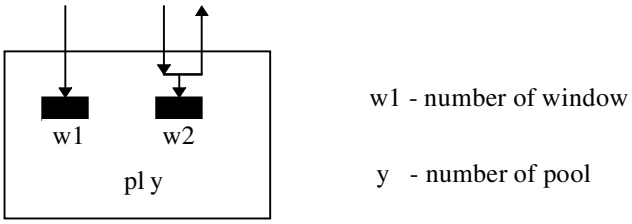
channel

pools = items of information in a pool are :available for reading and/or

writing by a number of processes in the system,

- **information does not flow within a pool i.e. any item in it will be available to processes using the pool,**

- **processes are software models of the activities that a real-time system must perform, pools form the software models of items in the system.**



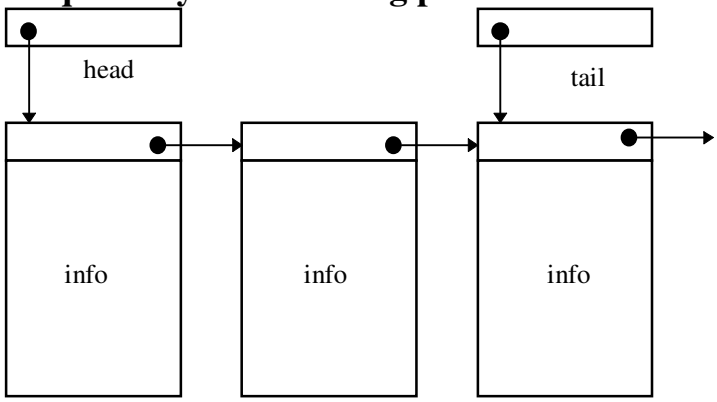
pool

Implementation of Communication

implementation of the channel:

I. queue

- **it is a first-in-first-out (FIFO) queue where messages are placed on the tail of the queue by the sending process and removed from the head of the queue by the receiving process**



queue

- **the queue consists of three separate parts:**

  1. sender's memory awaiting transmission,

  2. communication medium,

  3. receiver's memory awaiting a message

- **the queue need not necessarily have a FIFO organisation, messages could be given priorities and placed in priority order within the**

queue i.e. certain sending processes could be given the highest priority, and their messages always placed at the front of the queue.

- when queues are operating there is always the possibility that they will grow to the extent that the system free-space will become exhausted,

- there are two ways to handle this situation

1. the system could refuse further work - that is, refuse requests from its controlled system for action

for example: the system controlling an airline booking system could refuse to process any new transactions until an overload was reduced.

2. to allow the system free-space to expand into the back-storage (if present). Once the free-space in main memory is exhausted, the queue would start using a free-space area on fast back-storage.

2. circular buffer or hopper*:*

- a fixed-sized buffer is set between the communicating processes where the buffer /or the storage component/ is divided into a number of `message'-sized elements

- the transmitting process uses and moves the `loading' pointer, while the receiving process uses and moves the `unloading' pointer,

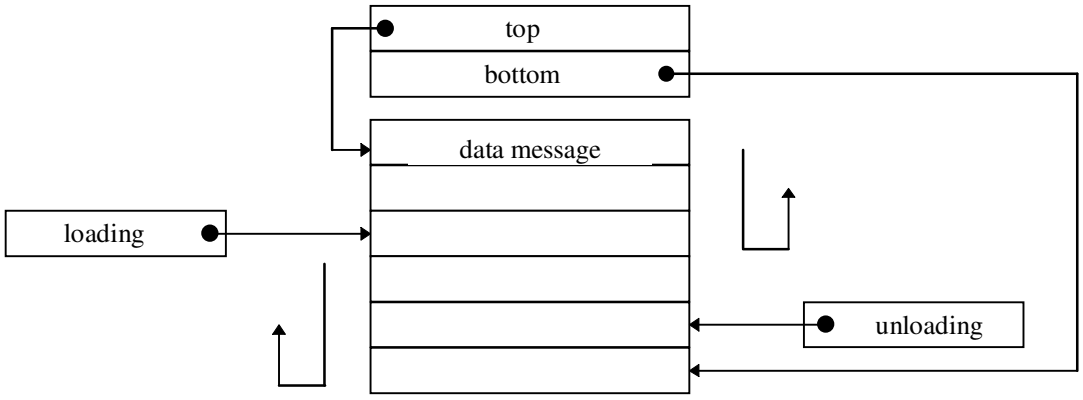


Figure 2.2.7. The hopper

- advantages and disadvantages of the queues and the circular buffer

queue

- can be extended for as long as there are package-sized elements of storage available in the system's free-space area.

circular buffer

- **has fixed size and that reduces its flexibility**
- **However, it can be manipulated faster than the queue mechanism, as transfers to and from the system free-space area are not required.**

3. pool

- **pools usually take the form of system tables, shared data areas and ; shared files.**
- **they consist of a piece of main memory or back-storage which can be read from or written to, by more than one process.**
- **they are used to store information that is relevant to more than one process in the system,**
- **the data structures will take the form of tables, lists or individual storage elements held in main memory.**
- **often, for system security and recovery purposes, a regularly updated copy of this information will be held on back-store.**
- **it is essential to control strictly the access to information in pools.**
- **if a large number of processes have direct access to a particular pool, a malfunction in one of the processes will cause corruption of pool information, and this will propagate throughout the system.**
- **similarly, if any change is made to the format of pool information during the maintenance phase of the product's lifetime, the code for all the processes using the pool will have to be changed.**
- **we can introduce the concept of a pool monitor. If a process wishes to access a piece of information in a pool, it does not go to the pool directly, but asks the pool's monitor to carry out the access,**

Co-ordination

- **as well as communicating information to one another, processes may wish to co-ordinate their activities,**
- **co-ordination involves the ability of one process to stimulate or inhibit its own action or that of other processes,**
- **in other words, in order to carry out the activities required of it, a process**

may need to have the ability to say STOP or GO or WAIT A MOMENT to itself, or other processes.

Events

- co-ordination between processes centres around the occurrence of events in the system,

- co-ordination is based upon communication of a piece of information between processes where this information is that the event has occurred

- using two procedures - WAIT(event) and SIGNAL(event) co-ordination could be performed

- WAIT (event) causes the process using it to suspend activity as soon as the WAIT operation is executed, and it will remain suspended until such time as notification of the occurrence of an event is received

- if the event have already occurred the process will resume immediately,

- SIGNAL operation broadcasts the fact that an event has occurred,

- its action is to place event information in a channel or pool to enable a waiting process to continue.

interrupts = to display sources of events, external to the software as a *hardware interrupt* mechanism,

Implementation of Co-ordination

- there are several basic co-ordination functions

- ACTIVATE(process id),

- SUSPEND(process id)

- DELAY(process id, time).

- ACTIVATE, SUSPEND and DELAY functions are dangerous because the affected processes do not have total control of their co-ordination activities,

- all necessary co-ordination activities can be implemented using WAIT and SIGNAL operation,

- implementation of WAIT and SIGNAL depends on the system involved for example at the

lowest level they rely on a hardware lock-and-unlock mechanism to ensure that they are impervious to process swapping while they take

action,

high level the may be embedded in high-level language constructs.

- process co-ordination based on WAIT and SIGNAL operation may use semaphores,

semaphore = a simple data item that can only take non-negative integer values and can only be manipulated by three procedures:

initialise(semaphore, value):

- initialise sets the value of the semaphore to value.

wait(semaphore):

- wait will decrease the value of the semaphore by one, but only if the result is non-negative.

signal(semaphore):

- signal simply increments the value of the semaphore by one.

- processes wishing to co-ordinate their activities execute WAIT and SIGNAL operations on shared semaphores,

- if a process executes a WAIT operation and the value of the semaphore is one or greater, then the process can decrease the semaphore and continue,

- if, however, the semaphore has the value zero at the time the process executes the WAIT operation, then decreasing the semaphore would result in a negative value, the process must therefore wait until such time as another process executes a SIGNAL operation in the semaphore, thus allowing the first process to decrease the semaphore and continue,

- the value of a semaphore can never be negative, this implies that the number of completed WAIT operations on a semaphore must always be less than or equal to the initial value of the semaphore plus the number of SIGNAL operations that have occurred,

- both WAIT and SIGNAL are indivisible operations; once begun, they must be completed,

- the processor cannot be swapped among processes while they are being executed, this stipulation ensures that the vital increasing and decreasing of the semaphore occurs without interruption.

- WAIT and SIGNAL operations are implemented by software trap to provide atomic WAIT and SIGNAL operations

Monitors

- processes should be unaware of the mechanisms driving the channel or pool.

- they should only need to present the channel or pool with information, or to request information from the channel or pool, and then expect that correct action will be taken.

- furthermore, the processes should have no knowledge of the internal workings of the channel or pool. Unless this is the case, the processes will have to know too much about their environment for them to be regarded as free-standing programs.

- In the simplest case, the necessary facilities must be provided by creating a number of shared, re-entrant subroutines to be called by the processes. These access procedures would effect the necessary information transfers and, by suitable use of WAIT and SIGNAL operations, ensure that processes did not corrupt the channel or pool data structures. This type of data corruption is all too possible if asynchronous processes are allowed to manipulate common data.

This collection of controlling routines is called as a monitor. When a process wishes to read or write to a channel or pool, it asks the relevant monitor to do the reading or writing for it. Only the monitor is aware of the internal structure of the channel or pool.

Any changes - for example, from a queue mechanism to a hopper mechanism - need only be made known to the monitor. The user processes are unaware, and unconcerned. In this way, processes can be regarded as free-standing programs that utilise communication `pipes' to interface with the outside world. In this case the definition of a monitor is restricted to a collection of controlling routines. Hoare has proposed a more formalised definition that includes the implementation in a high-level language of modules specifically declared as monitors.

Monitor Facilities

Monitors can be designed to provide increasing levels of service to a real-time system.

Basic Facilities **At its simplest, a monitor consists of a group of routines designed to implement the functions of its channel or pool. To do this, it must**

- provide the necessary data-transfer facilities, and

- ensure adequate synchronisation of the user processes to avoid data corruption.

Watchdog Facilities **A monitor can be designed to act as a watchdog or policeman over the activities of its user process. The monitor can check the validity of the information passing through it. In a system test situation, the monitor could cause irregularities to be flagged. In an in-service environment, the monitor could instigate appropriate recovery action if invalid data were being generated by one of its user processes.**

Testing/Debugging Facilities **Monitors provide ideal services in a system-testing environment or 'test harness'. As they control the flow of information and act as the links between processes, they provide ideal points for inserting software tests, and introducing test data. Monitors can be built to include all of the above-mentioned facilities. When they are to be used in a time-stringent environment, all bar the basic facilities can be removed from the monitors by conditional compilation.**

2.2.5. Executive Processes

Various special processes have been mentioned: these include the interrupt-handling process, the process for swapping volatile environments and monitors. All these activities are needed to support the existence of the real-time virtual machine, and fall under the heading of executive processes. The executive may be regarded as that part of the real-time system which supports the existence, and controls the activity, of the processes in the system.

Processes that make use of the executive are called as user processes. The form and power of the services provided by the executive vary according to the size of the system being constructed. Typically they include: process swapping, interrupt handling, memory allocation, scheduling, interface monitoring, input/output device drivers and file-system control.

A user process, when it wants an executive service, calls on the executive to carry out the required actions. This call could be a simple subroutine call. However it is desirable to isolate and protect the executive from the possible malfunctions of other processes in the system. The executive will normally be small and consist of highly reliable pieces of code. If the executive is corrupted in any way, it is difficult for the system to continue operation. However, if an individual user process becomes faulty, it may be possible for the executive to take some corrective action.

2.2.5.l. Protection

In order that no process can corrupt the executive, it is desirable to build a `firewall' between the executive processes and other processes in the system. This protection is not always possible even in a very simple microprocessor system. However, most modern computer systems and the microprocessor systems include some form of memory-management

hardware that does allow certain processes greater privileges than others, and does guarantee that processes cannot interfere with the memory allocated to other processes.

Each process is allocated a protection domain. A protection domain is the combination of the area of memory in which the process runs and the privileges - such as the ability to execute certain subsets of processor instructions - that the process possesses. Only the most privileged processes are given the power to transfer control of execution to another process's memory area. A simple protection set-up is a two-level `kernel' and `user' hierarchy. Processes in the kernel domain, or `kernel space' would have access to all available machine instructions, while processes in user space would be restricted in the set of instructions they would execute.

Now, if memory-management and protection hardware is available to the system, the executive processes will be placed in a privileged area. This arrangement implies that a user process cannot obtain executive service by means of a simple subroutine call. The executive will be situated in a different protection domain, separate from the requesting process. A conventional solution to this problem is for the user process to execute a software trap instruction, after having placed suitable parameters in a location known and available to the executive processes.

The interrupt handling process will then recognise the trap as a call to the executive (system call) and signal to the relevant executive service process. Figure 2.2.8. shows this arrangement. Note that the executive is always entered via the interrupt service routine (or other closely guarded gateway). The executive is exited via the dispatcher. It is important that the software designer make full use of the protection mechanisms made available by the hardware. Some hardware systems provide more than two protection domains. If this is the case, then the designer would arrange to place the different processes and executive subroutines in suitable protection domains.

If no protection hardware is available, then a small amount of protection could be afforded by having the process call an executive monitor which checks the legality of the request, rather than calling the executive routines directly.

The flexibility and low cost of ROM and EPROMS allow us to fully protect the monitor code, since these memory devices cannot be accidentally overwritten. Such devices can be addressed within the normal address space (memory mapped) and constitute a guarantied protection.