2. REAL-TIME MACHINES

The design of real-time systems is an extremely difficult task. On the one hand, there is a complex and often hardly defined problem, on the other hand, the hardware is only capable to satisfy certain set of constraints. Before designing the hardware and the software of the real-time systems the designer must clearly know what kind of requirements has the system. Formal methods were developed to specify and verify the requirements of real-time systems. All these approaches provide a description that defines the system to be designed and built. They are often in graphical form and include a set of rules that highlight those areas where the specification is incomplete.

There are number of software development support systems. If there were available a machine that could use the requirement specification developed by software development support systems, then the real-time software design would be solved very simple. It would be necessary to describe accurately the system to built and then this description would serve as the software model.

Unfortunately, such a machine does not exist. Currently available computers cannot use the requirement specification, the forms of symbolic model that would best represent the application. However, it is possible to elaborate such a machine. This machine is virtual machine that is created by placing layers of software between the user and the hardware. The layers of software create successive virtual machines or level of abstractions. Each successive layer forms a more sophisticated machine whose attributes are more closely tailored to the designer's requirements. The software comprising each virtual machine layer automatically translates data and actions that define the machine at that layer into the simpler data structures and actions that define the previous layer.

This automatic translation of higher-level abstraction into detailed lower-level operations achieves a significant reduction in overall system complexity. The designer has only to create mapping between the requirement specification and the data and actions required for the virtual machine. This is also not easy, but it is certainly less complex than mapping the problem directly onto the hardware.

If no virtual machine exists, then the designer must be aware of the detailed operation of the hardware.

2.1. Real-Time Virtual Machine

Real-time systems carry out a set of activities or tasks. Controlling a device could be one activity; keeping a record of the overall performance of the system may be another. All the activities in the system are interrelated and interact with one another. Some activities may be more vital than others, dependent upon the environmental conditions affecting the controlled system at any one time. 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. Clearly a real-time system is multifaceted and exhibits enormous complexity. Designer are faced with a set of interdependent activities, all jostling to be carried out within strict time limits. Unless some form of abstraction is applied, the design process becomes almost impossible. A method of attaining this abstraction is to design, create and use a suitable virtual machine. If such a machine can be developed for a real-time environment, then the design problem can be reduced to manageable proportions.

The existence of a virtual machine enables a designer to attain a degree of abstraction from the computer hardware. However, the virtual machine must do more than simply reduce programming complexity. It must be adapted to suit the needs of its environment and area of application. It must be designed in such a way that it facilitates and encourages the creation of clear, logical and easily implemented designs. More, it must promote the design of systems that are as simple as possible.

2.1.1. Processes

A real-time system can he seen as one that carries out a set of activities. The most elementary real-time system would, therefore, be one which carried out one, and only one activity. In order to build such a system it would he necessary to provide a suitable program of instructions, together with the processing hardware on which it could run. The combination of a simple a program and its environment that includes hardware, which together perform one activity, is a process in the virtual machine. 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 the outside world.

But real-time virtual machine must be able to support more than one activities. These activities may be either unrelated or related. It is unlikely that a real-time system would control number of totally unrelated activities. Normally, it would be required to control an environment where the devices and therefore the processes controlling them, are interrelated in some way. In this situation the action of one device must take into account the action of the other devices. In this case there is a set of independent processes, and the virtual machine that supports multiple but independent processes is no longer adequate. It must be extended to include interaction between processes. In a graphical representation this interaction could be illustrated with arrows. Where one process wishes to interact with another, an arrow will be drawn between the two processes. If each process wishes to interact with all the others, then they must be connected by arrows, Clearly, as number of processes is increasing the situation becomes extremely complex. Designers who are to design a worthwhile virtual machine, must control this complexity. Fortunately, if designers have some thought to the way that devices, and therefore processes, wish to interact, they make a number of abstractions. These abstractions, if applied to the virtual machine, will achieve considerable reduction of the potential complexity.

The interaction of processes will reflect the interaction of the device they control. This interaction may be direct or immediate. in that the action of one device will cause the starting or stopping of another device's activity. Alternatively, the interaction may be indirect where a device may receive information about the status or actions of another device and be obliged to modify its actions accordingly. The corresponding processes will need to interact in a similar manner. They will need to be able to synchronise 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 send information to one another and to access information that reflects the status of the system as a whole. In order to include the necessary facilities in the virtual machine model, process interaction must be discussed in more detail.

2.1.2. Communication

Information may be communicated between processes in two ways First, a direct transfer of information may occur from one process to another. Second, each process may access or update pieces of shared information. This shared information is available to some or all of the processes in the system.

Channels In order to provide direct communication between processes in the virtual machine the concept *of a channel* must be introduced. The graphical representation for a channel used in the system is shown in Figure 2.1.1. A channel provides the medium for items of information to be passed between one process and another. A channel has a certain elasticity. 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. The virtual machine is unconcerned as to precisely how the transfer of information is effected. It is sufficient in the virtual machine design to say that, where a channel exists between two processes, items of information may flow through that channel from one process to another.

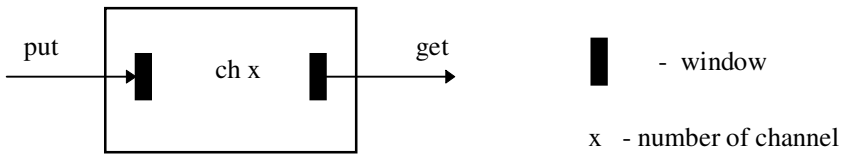

Figure 2.1.1. The channel

Pools To provide system-wide information another concept to the virtual machine must be introduced - a pool. 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. A pool acts as a reporatory of information; any item in it will be available to processes using the pool. The virtual machine provides no indication as to how the pool wi11 be set up; it simple provides the facility for specifying that a pool of information does exist. Figure 2.1.2. illustrates a symbol for a pool.

Pools are as important an element in the virtual machine as processes. Whereas processes are software models of the activities that a real-time system must perform, pools form the software models of items in the system. These items include physical devices and mechanisms in the controlled system and conceptual items such as logical files. In many application fields the pool has information written into it endlessly, but the storage allocated to the pool does not need to be infinite. since eventually old information ceases to he of any practical use and can be disregarded.

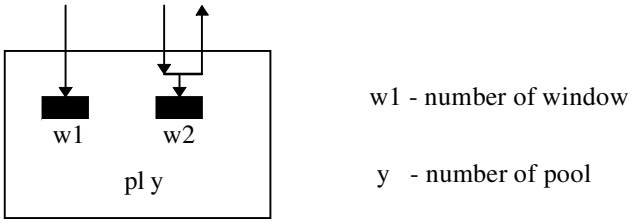

Figure 2.1.2. The pool

Intercommunication Data Areas. Channels are pools must be further developed, particularly for large systems and multiprocessor target systems in order to describe their behaviour. The basic communication model of the virtual machine contains activities, referred to earlier as processes, and intercommunication data areas (IDAs) which combine the channel and pool concepts. This concept contains the following elements :

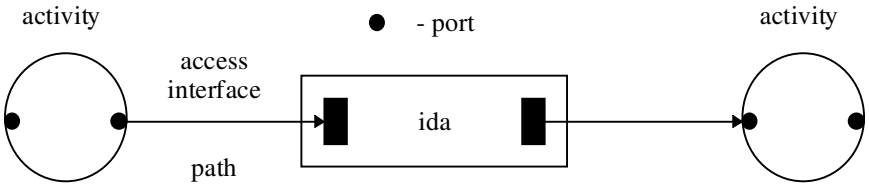

Figure 2.1.3. Intercommunication data area

- *activity:* This is a basic processing element having a single thread of program execution. An activity does not have right of access to either data or code outside of itself; all external communication must be through an IDA.
- *intercommunication data area:* This is a basic element through which activities communicate with each other. An IDA may provide just communication or just a pool, but typically contains both a data or information area and associated access procedures. The latter relate only to the activity which invokes them. The consequence of this is that an IDA may exhibit active concurrency with respect to more than one activity, and consequential upon this is a requirement to provide synchronisation and mutual exclusion, concepts which are discussed later.
- *path:* It provides the means for interaction between activities and IDAs.

- *port:* It may have a number of ports on its boundary, to each of which a path is connected as required by the activity. The type of port is determined by an access interface specification.
- *window:* The window is the corresponding point on the boundary of an IDA to which a port may be connected. An IDA may have any number of windows.
- *access interface:* This specifies a set of interactions that can take place on a path, implying common constraints on both activity and IDA.

The channels and pools discussed earlier are alternative forms of IDA with slightly modified graphical symbolism. Figure 2.1.3. shows a simple system consisting of five activities communicating via an lDA. two channels and two pools.

The simple concepts illustrated here are inadequate for complex systems, particularly with respect to the hierarchical decomposition of the problem into subsystems from both process and communication points of view.

External Devices Many real-time systems employ computers embedded in larger systems. There is a need to communicate with hardware devices external to the computer system. This is achieved by a server, shown graphically in Figure 2.1.4, and representing the only elements which include handlers and low-level code for direct communication with external hardware devices.

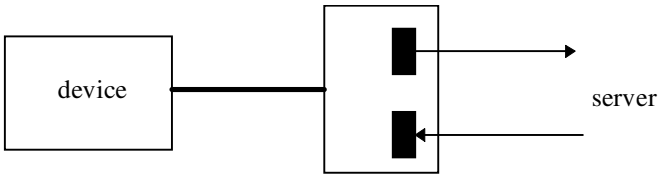

Figure 2.1.4. The server

2.1.3. Synchronisation

As well as communicating information to one another, processes may wish to synchronise their activities. This synchronisation 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.

If each process has the power to stop or start the activity of any other process in the system. It is important that a process must not be ;allowed to have this direct power over other processes. Fortunately, it is possible to design simple procedures which fulfil all the requirements of interprocess synchronisation and. at the same time, limit the the process-activation and suspension powers of the process using them.

Significant events Interprocess synchronisation centres around the occurrence of significant events in the system. Consider an example. One process in a system may need to suspend its activities at a certain point until time an event will not happen. Another process in the system will be the one which controls the event and changes its state. Synchronisation between these two processes centres around the significant event of .changing the state of the event. The first process must wait for the event to occur; the second process will signal that the event has occurred.

Synchronisation is based upon communication of a piece of information between processes. This information is simply: the event has occurred. If two procedures - WAIT(event) and SIGNAL(event) - exist, they can be used to implement all necessary synchronisation. 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. Should the event have already occurred the process will resume immediately. A waiting process can be thought of as being in the act of reading event information from a channel or pool. Once this information appears, it can continue.

The SIGNAL operation broadcasts the fact that an event has occurred. Its action is to place event information in a channel or pool. This in turn may enable a waiting process to continue.

The designer, when using the virtual machine, need not be concerned with how the WAIT and SIGNAL operations are implemented. He is nevertheless in a position to base the design of process synchronisation on the occurrence of significant events. He can display process interdependency diagrammatically by showing the channels and pools through which knowledge of the event is transmitted.

Interrupts must be mentioned at this The designer has to display sources of events, external to the software, as squares shown in Figure 2.1.5. This feature is, in fact, a *hardware interrupt* mechanism. However, the user of the virtual machine need not concern himself with the physical implementation of the mechanism.

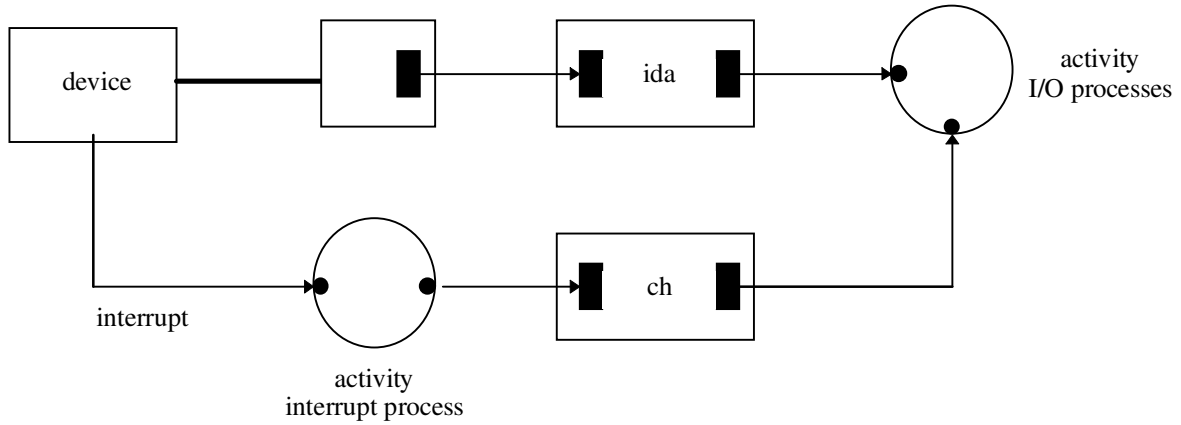

Figure 2.1.5. Handling interrupts

2.1.4. Example

In order to illustrate how to use virtual machine it may be useful to analyse a very simple example. There is a small information system. A number of terminals are attached to the computer system by using terminal. a person may interrogate and possibly update information in a set of files held on the computer's back-storage. The person must first make himself known to the system by typing in a password. He can then using a set of one-line commands, read from and, if he is eligible, write to the system files. This arrangement is, of course, a very simple basis for a management information system, stock-control system, airline booking scheme or similar. To start the design for the software needed to control this system the designer has to identify first the *items* in the system, and then the *activities* that must be carried out.

Items of the information systems

The Users Each user of the system will have a password, and some security code that indicates to which files he may read and/or write.

The Terminals In this system it is assumed that each terminal is allocated a unique location in the computer's main storage, in which it can place a character as it is typed. Similarly, there is a unique location where it accesses the next character to be sent to the terminal.

The Commands Each command to the system will take the form of a line of text and represents one action. Such actions would include "read a file record" or "write a file record" or "move to the next record"·.

The Responses These will be strings of text. displayed at the terminals as responses to commands

The Files These are the information files that the users wish to manipulate.

Activities of the information systems

Handle Terminal Input The system must move characters out of each terminal's input location fast enough to ensure that no character is overwritten by the next incoming character. The characters make up commands. Each command consists of a string of characters terminated by an end-of-line indication. The activity-handling terminal input will send in characters until an end-of-line character is received, at which point it will signal that a command has been read.

Handle Terminal Output The system will build responses to the user in the form of strings of characters. The designer must provide an activity that will place these responses, character by character, into the terminal's output location.

File Handle The designer need to include an activity to control the organisation and access to the information in the system files.

Carry Out Command Finally the designer will interpret the user commands. He must ensure that the user and/or the command is valid and legal, then take action on the command and provide a response to the user.

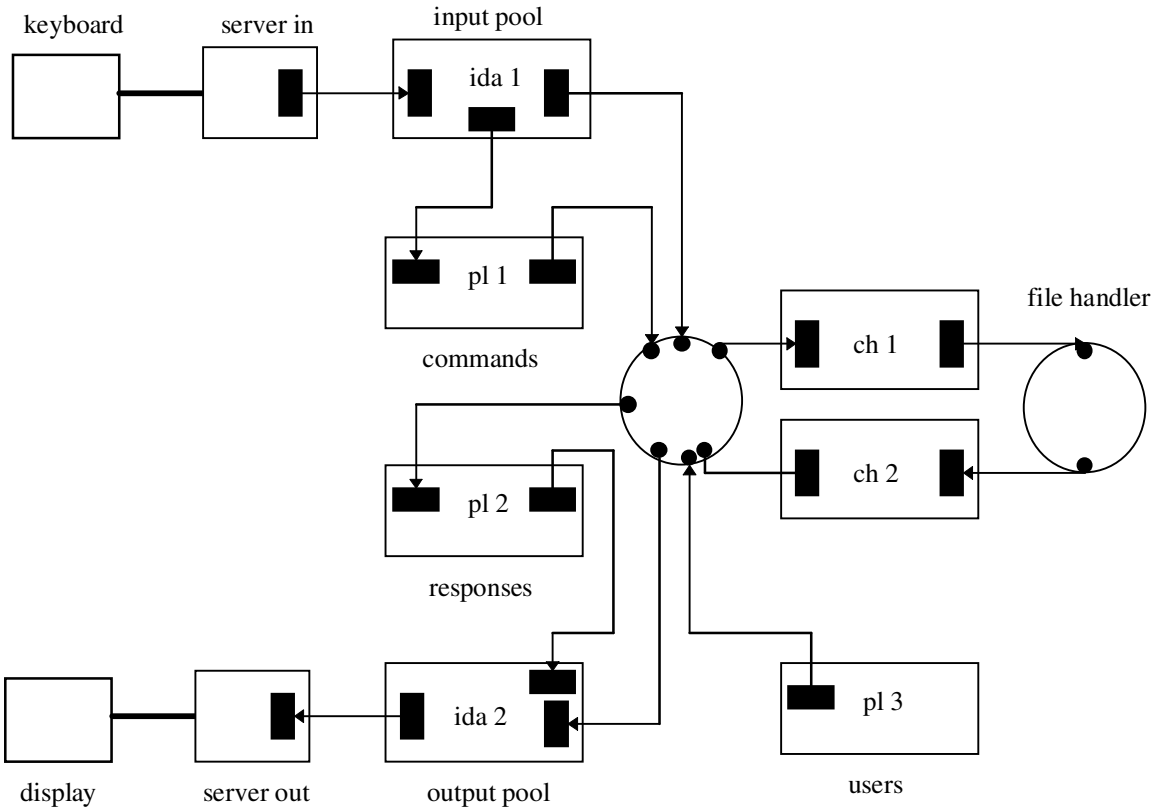

Figure 2.1.6. Example of information system

Design of the information systems

Every *item* in the system will be modelled *as a pool.* Every *activity* will be modelled as a process. One possible design is shown in Figure 2.1.6. . A · terminal is in fact contains two devices. the keyboard for input and the screen or printer for output.

The IN process builds up commands, character by character. in the COMMAND pool. When it receives an end-of-Iine indicator, it sends a message down cl. indicating that a command has been received. When the OUT process receives a message down c2 it transfers the; response specified by the message to the terminal output location. When signalled by IN, the control process checks the validity of the command by referring to the users pool to check the passwords and security codes. It then sends a request for action by the file handler down c3. It receives an acknowledgement to this request on c4. It then places the response to the command in RESPONSES and signals OUT

2.2. Implementation of the Real-time Machine

This chapter will describe the software and hardware necessary to support the existence of such a virtual machine. It will be necessary to implement processes, and to provide communication and synchronisation facilities between the processes. The process consists of a processor, program code and sufficient memory to accommodate this program code. A machine must be produced that provides these resources for as many processes as required. Each process must. as far as the process itself is concerned, have its own processor (CPU), program and memory. These resources can be provided by either dedicating unique resources to each process as in a distributed system, or by sharing the available resources between the processes.

2.2.1. Implementation of processes

2.2.1.1.Dedicated resources

The recent availability of cheap microprocessors and memory elements means that it is a viable proposition to provide separate resources dedicated to each process. It is possible to provide for each process in the' system a separate microprocessor, together with enough memory for the code element. If the designer specifies a new process, a new CPU/memory is slotted into the system.

Figures 2.2.1 and 2.2.2 illustrate schematically, two commonly used configurations of multiprocessor hardware. In Figure 2.2.1 processors use their own local memory to store program code. They communicate with one another via communications links that are ·set up to allow relatively long distances between the CPU/memory modules. There is a clear correspondence between the channels of the virtual machine and the physical links between the microprocessor modules.

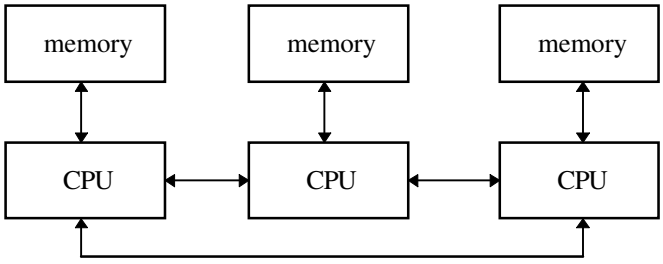

Figure 2.2.1. Distributed memory architecture

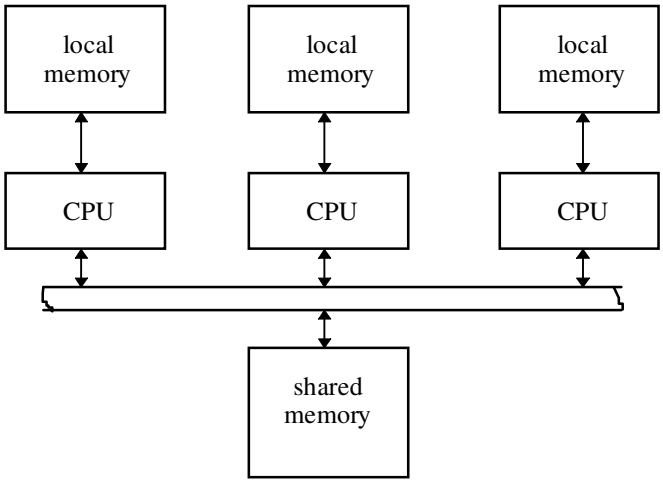

Figure 2.2.2 Shared memory architecture

Figure 2.2.2 shows processors connected via a common bus arrangement. The local storage for each CPU may contain all or some of the process code. If the local memory does not contain all of the program code for a process, then some will be held in shared memory. The shared memory is the logical place to hold the information in the system pools.

With either of the above configurations it is clear that CPU/memory/code combination for each process must be provided. In order to create a process one need only load one of the memory modules with the relevant program code element. Clearly this simple approach should be utilised wherever feasible. It has, however, a number of drawbacks in certain circumstances.

Cost Often economic constraints rule out the possibility providing a unique CPU/memory module for each process. If insufficient resources are available, it becomes necessary to share resources between processes. None the less, in applications requiring a small fixed number of processes, with modest reliability requirements, the simple multi-microprocessor realisation of the virtual machine would be the most desirable configuration.

Lack of Flexibility There are more compelling reasons for introducing a more flexible relationship between processes and their hardware resources. A rigid allocation of process to CPU/memory implies that no new processes can be introduced into the system without altering the hardware configuration. In some applications this can prove a severe restriction. 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 impinge on possible expansion. The system could not he extended to include new activities without adding further hardware.

Reliability Considerations of system reliability make it convenient to include redundant CPU; and memory modules in the system. When a piece of hardware 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.

2.2.1.2. Shared resources

A. 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.

A/1. Serial Execution. The simplest method of sharing a CPU is for the processes to be executed once through their code without interruption, 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. The 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.

Provided the application lends itself to short processes that carry out highly independent activities then the simplicity of the serial approach has much to commend it. *How* is simple in this case: commence the next process at its starting point. Each process, when terminating, executes a piece of code that performs this function. And when '? At the completion of the current process.

This does not mean however that such processes may not communicate with each other. Processes such as input and output may assume that the preceding process has left information that is required for the current process in known places. 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.

A/2 Pseudoparallel 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 and dispatcher must be introduced. .

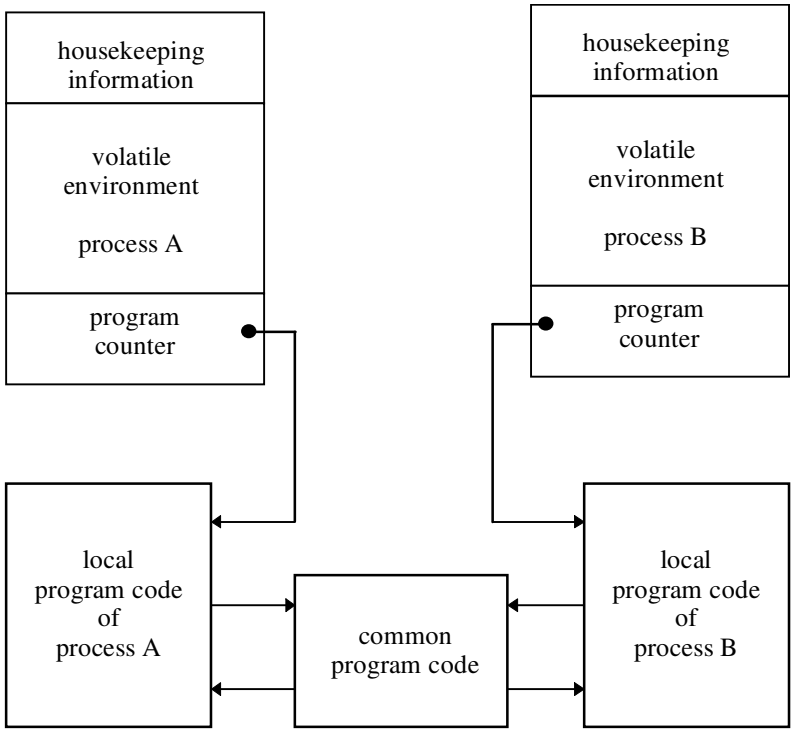

Figure 2.2.3 Process code sharing

A process's ***volatile environment*** is that information that, if lost, would mean that the process could not continue from the point at which it last executed an instruction. It is the information that would be lost if another process used the CPU. The 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 the·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. Note, however, that 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.

Therefore, a place to store a process volatile environment must be provided while it is not using the CPU. At the·same time, space for summary information about the process for system housekeeping purposes must be also provided. This storage element will take the form of a data structure that, in effort, models the process. This data structure is called ***process descriptor*** (PD). So, it is a further extension of the process concept thus the process consists of a process descriptor and the program code.

The ***dispatcher*** is the mechanism that effects the how processes are shared. It can be implemented in hardware or software. When 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.

The when of process sharing is related to the conditions necessary for the current process to give up use of the processor. A process may relinquish the use of the CPU voluntarily, or be forced to do so. A voluntary release of the CPU would occur when a process either terminates or executes an unsuccessful WAIT operation. A forced release of the CPU occurs when an interrupt occurs and an interrupt handling process is activated.

The controlled system influences the attention of the software system by means of the hardware interrupt mechanism. Should 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 handling*** 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. Clearly, the interrupt hardware acts as a simple dispatcher causing a forced, restricted process swapping.

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

B. 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, and the one to be used if possible, is to provide in the system enough memory for each process to have its own unique area. Often this simple condition cannot be employed. In most modern systems the cost of main memory far outweighs the cost of the CPU. 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. However, the falling cost of memory means that distributed memory becomes a possibility and the problem might be solved. More likely, the requirements of the system will increase to match this advantage, and the problems of sharing memory will remain. There are two ways of memory sharing - to share the code that resides in a memory; or to share the use of the section of memory itself.

B/1/ Code Sharing. Real-time systems frequently contain a number of actions that form part of more than one process in the system. Conventionally. these activities are implemented as subroutines and a copy is built into the program code of any process that needs it. Now, if one of these subroutines were 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 program or subroutine consists of instructions plus data. Therefore, the shared routine will have local storage areas for its temporary working variables. In a shared CPU environment a process may be forced to give up the processor at any time during its execution. So. in Figure 2.2.3 process A may be halfway through executing routing S when it gives up the processor. Process B then takes over the processor and enters routine S. While S is executed. the temporary data in S will be manipulated according to B's requirements. If A then regains use of the processor, the temporary

data used in S will have been changed by B,s action. This fact is unknown to A and may cause its malfunction. There are two ways to avoid this situation :

- *serially re-usable code* One method is to write the code in subroutine S in such a way that it makes 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. If a lock mechanism is applied at the beginning of the subroutine. and an unlock at the end, such that only one process may be executing its code at any one time. then processes can safely use the code. one after the other. The code is serially re-usable. The lock and unlock mechanisms can be implemented using WAIT and SIGNAL operations.
- r*e-entrant or pure code* If all the temporary data areas needed by S were to be part of the process currently using S, rather than part of 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 S did not modify its own instructions in any way. Code elements having this property are referred to as re-entrant or pure procedures.

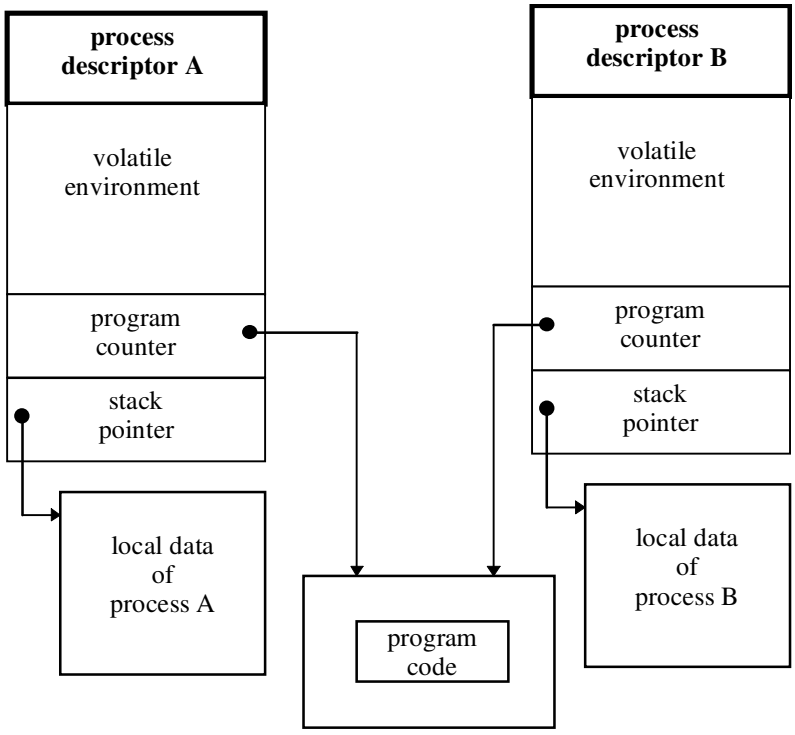

Figure 2.2.4. Processes with common program code

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 in an indirect manner. Typically, the re-entrant code module will access the 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. Some mechanisms will be needed to store the return address of the subroutine call. This is supplied conveniently by a stack mechanism. A stack can be regarded as a last-in-first-out queue of memory elements. The stack area must be part of the individual processes. In some implementations, the local data area is also included as part of the stack.

The amount of shared code that a process uses could be extended to the whole of the process's code. In Figure 2.2.4 there are a number of processes that execute the same code. They are, however, completely different processes because they use different data areas. Each process consists of its data

together with its own program code area. It is unaware and unconcerned that another process is actually using part of the same program code for its action.

The device drivers in a conventional operating system are good example of this situation. When building an operating system' facilities it is usual to include input/output processes to handle different devices attached to the system. To implement device-handler processes the systems programmer constructs the data structures and code modules shown in Figure 2.2.5. The device descriptors contain device-specific information such as character set and speed. The device driver code uses the information in the device descriptors to tailor its actions. The device driver code, process descriptor. plus a device descriptor constitutes a process. To introduce a new process to handle another terminal, for example, the system programmer need only to create another device and process descriptor.

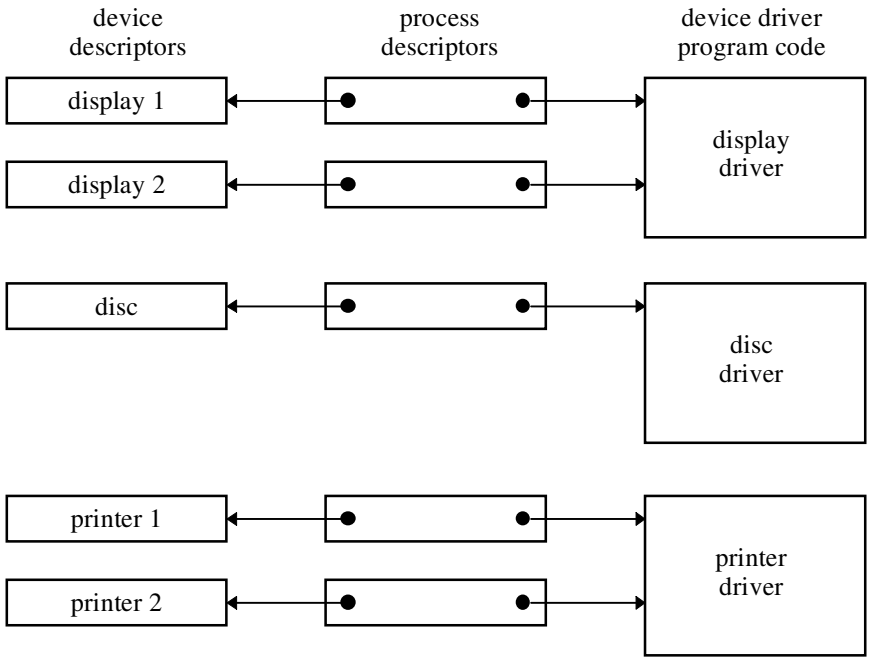

Figure 2.2.5. Input/output-driver data structures

B/2 Memory sharing. 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. Program codes will be held on back-storage and copied to main storage when it becomes necessary to execute them. If a copy of a program code is currently in main storage, it is resident and may be executed immediately; if not, it is non-resident and must be copied to main storage before it can be executed by the CPU.

Most real-time systems divide main memory into two areas. The first holds frequently executed program code and the program code of highly time-critical processes. These code elements reside permanently in memory. 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 that is less· frequently executed. The relative size of the two areas will depend upon the requirements and constraints of the particular system.
The problem of when to share memory occurs when an active process wishes to execute a program code segment that is not currently in memory. It also occurs 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. The memory-sharing process is then alerted.

The problem of how to share is solved by a special process, or set of processes, built to control the allocation of memory. Most mechanisms maintain a list of all available pieces of memory. This list may he sorted in increasing or decreasing order of memory size or memory address. 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. Once this code section is no longer required it may he copied to the back-storage and a piece of memory made available for other processes.

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

2.2.2. Implementation of synchronisation

Many different synchronisation facilities or functions can be thought to be useful. They include DELAYME(time), ACTIVATE(other process), SUSPEND(other process) and DELAY(other process, time). DELAYME can be implemented as a WAIT on an event that will occur later. Facilities like ACTIVATE, SUSPEND and DELAY are dangerous. Their existence implies that the affected processes do not have total control of their synchronisation activities.

In fact, all necessary synchronisation activities can be implemented using WAIT and SIGNAL. The detailed implementation of WAIT and SIGNAL depends on the system involved. At the lowest Ievel they rely on a hardware lock-and-unlock mechanism to ensure that they are impervious to process swapping while they take action. At a high level the may be embedded in high-level language constructs.

2.2.2.1. Semaphores

Dijkstra outlined a synchronisation concept against which other methods are measured. He introduced the idea of semaphore a simple data item that can only take non-negative integer values and can only be manipulated by three procedures: initialise(semaphore, value), wait(semaphore) and signal(semaphore). Initialise sets the value of the semaphore to value. Signal simply increments the value of the semaphore by one. Wait will decrease the value of the semaphore by one, but only if the result is non-negative.

Processes wishing to synchronise 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:

At any time, the value of the semaphore is equal to its initial value plus the number of signal operations that have been applied to it minus the total number of completed (that is passed) wait operations. Now, because 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 while they are being executed. This stipulation ensures that the vital increasing and decreasing of the semaphore occurs without interruption.

A software trap is a machine instruction that, when executed, causes an interrupt to be signalled to the interrupt mechanism. It is an interrupt generated by software. It is worth noting that if wait and signal operations are implemented in such a way that they are entered as a result of a software trap, then they are in fact interrupt handling processes. The dispatcher will be activated on their completion.

2.2.3. Implementation of communication

In order to facilitate communication between processes, information available to one process must be made available to another process. This implies that 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.

In the discussion that follows, memory will refer to both main and back-storage. For the software designer the framework and principles remain the same; economics and performance requirements will govern his decision as to the location of data. Main memory can be accessed quickly, but more information can be placed on back-storage. 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. The data itself must be unaffected by the location of memory storage. A data pool may be located on back-storage or in main memory. Only the access procedures vary. The rapid advance in storage technology and the continual decrease in hardware costs make it vital to keep the data independent of the storage location. Unless the data structure is independent of the storage technology, it will not be possible for the designer to adapt the design to changing hardware economic,.

2.2.3.1. The Channel

The purpose of the channel is to provide a pipe of information passing from one process to another. As well as providing a vehicle for communication, it 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.

The information to be passed from one process to another is best designed as a fixed format message. It will usually contain identification information such as the name of the sending process. The message may contain the data itself, or it may be a pointer to the data. Commonly, either of two mechanisms is used to carry the message - the queue or the hopper.

The Queue A widely used form of channel implementation is the first-in-first-out (FIFO) queue. Here. 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. Figure 2.2.6. illustrates this arrangement. Note that when a package of information is to be passed between processes, a message must be built by writing information to a given section of storage. This storage is obtained from an available system free-space area. When a receiving process finishes with the transmitted information, it will return the storage element to the system free-space.

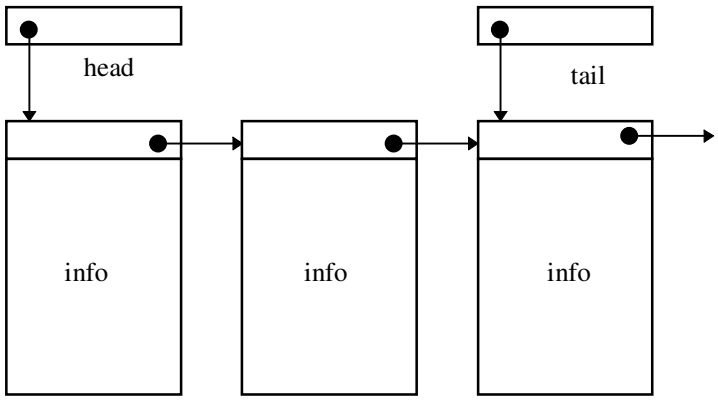

Figure 2.2.6 The queue

A distributed processor arrangement complicates the situation only slightly. In this case, the implementation of a channel requires the provision of a communication circuit with its associated

driving software. The queue is in three pieces. One piece is in the sender's memory awaiting transmission, another is in transit and the other is in the receiver's memory.

The queue need not necessarily have a FIFO organisation. Messages could be given priorities and placed in priority order within the queue. Alternatively, certain sending processes could be given 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. Steps can be taken to avoid this situation. For example, processes with excessively long input queues could be given a high priority. If however, in an overload situation, the available free-space may still become dangerously low. There are two ways to handle this situation. 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.

Many systems, however, are not in a position to ignore their environment. A second solution to the overload problem is 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.

The Hopper An alternative channel mechanism is a hopper or circular buffer. A fixed-sized buffer is set between the communicating processes. The storage is divided into a number of `message'-sized elements. The buffer acts as a hopper; see Figure 2.2.7. The transmitting process uses and moves the `loading' pointer, while the receiving process uses and moves the `unloading' pointer.

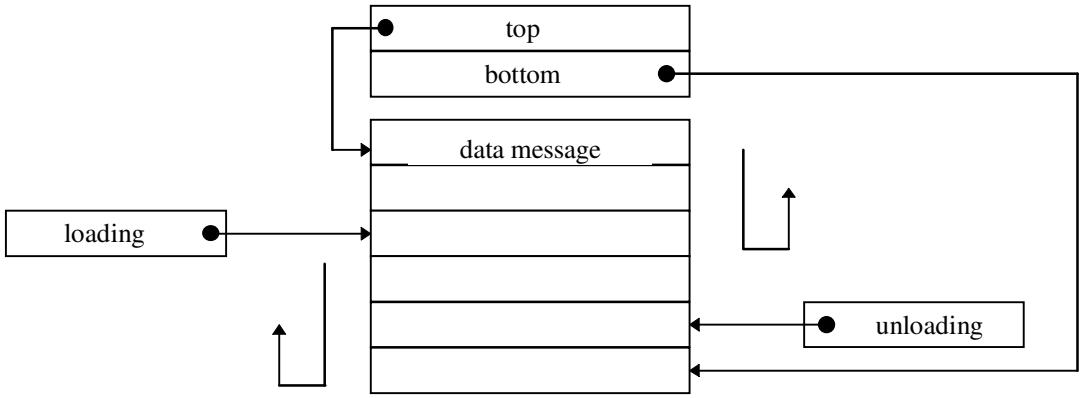

Figure 2.2.7. The hopper

The queue mechanism has the advantage that it can be extended for as long as there are package-sized elements of storage available in the system's free-space area. The hopper, on the other hand, is of fixed size and this 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. The decision as to which mechanism is used depends on system performance requirements.

The Pool Pools usually take the form of system tables, shared data areas and ; shared files. In general, 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 format of the data within the pools will depend on the particular application. They normally consist of data structures which model items in the controlled system or in the software system itself.

The data structures will take the form of tables, lists or individual storage elements held in main memory. In on-line commercial systems, for example, the pools will consist of on-line data files or a centralised database. These files will be read and written to by the processes controlling the on-line terminal activities and transaction processing. 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 have ramifications 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. These problems come about if processes using the pool are aware of layout details for the data structures in the pool. Usually, the processes do not need to know this detailed format information, as they are only interested in the data itself. If this is the case, 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. It will become evident that this concept of a monitor can be extended to channels and, in fact, to any item in the system.

2.2.4. Monitors

While discussing channels and pools it was assumed that implementation of a data structure (queue, file, etc.) is sufficient to fulfil the communication requirements of our virtual machine. In fact, more is required.

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.

2.2.4.1. 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.