# 4. RESOURCE MANAGEMENT in REAL-TIME SYSTEMS

## 4.1. Memory Management

Since the majority of control application software is static the software is not dynamically created or eliminated at run-time - the problem of memory management is simpler than for multi-programming, on-line systems. Indeed with the cost of computer hardware, both processors and memory, reducing many control applications use programs which are permanently resident in fast access memory.

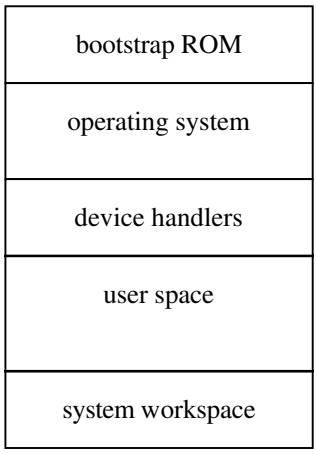


Figure 4.1.1. Non-partitioned memory

With permanently resident software the memory can be divided as shown in Figure 4.1.1. The user space is treated as one unit and the software is linked and loaded as a single program into the user area. The information about the various processes is conveyed to the operating system by means of a create process system call. An alternative arrangement is shown in Figure 4.1.2. The available memory is divided into predetermined segments and the processes are loaded individually into the various segments. The load operation would normally be carried out using the command processor. With this type of system the entries in the process descriptor have to be made from the console using a memory examine and change facility.

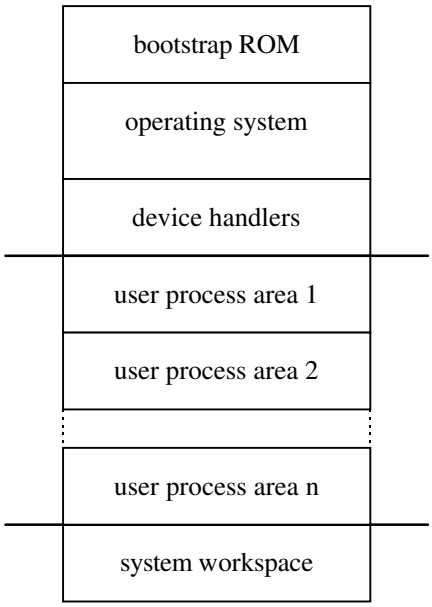


Figure 4.1.2. Partitioned memory

Partitioned memory was widely used in many early real-time operating systems and it was frequently extended to allow several processes to share one partition; the processes were kept on the backing storage and loaded into the appropriate partition when required. There-was of course a need to keep

any processes in which timing was crucial (hard time constraint processes) in fast access memory permanently. Other processes could be swapped between fast memory and backing storage. The difficulty with this method is, of course; in choosing the best mix of partition sizes. The partition size and boundaries have to be determined at system generation.

A number of methods: have been :used to overcome the problem of fixed partitions. One method; 'referred to` as floating memory divides the available memory into small blocks, for example 64 words: The processes are installed on the backing storage and when a process is required to run the operating system examines a map of memory and finds a contiguous area of memory which will hold the process. The process is loaded into. the memory and the memory blocks occupied by the processes are marked as occupied in the memory map, The area occupied by a process is closely related to the actual size of the process and not to some predetermined fixed partition size. A process which for some reason becomes suspended or delayed will have the memory area it occupies marked in the memory map as occupied but available; hence if another process becomes ready then the suspended process can be returned to backing storage and the ready process loaded into its area. In this type of system information must be held in the process descriptor to indicate if the process can be swapped; since, for example, a control process which has to run every 40 ms would have to be held permanently in memory in order to guarantee the sampling rate. A problem which is generated by this system is fragmentation of the available memory. Small areas of free memory become spread about the memory address space; none of the individual areas are large enough to take a process but the combined areas could if they could be brought together: Some form of garbage collection is necessary to bring dispersed areas into contiguous blocks.

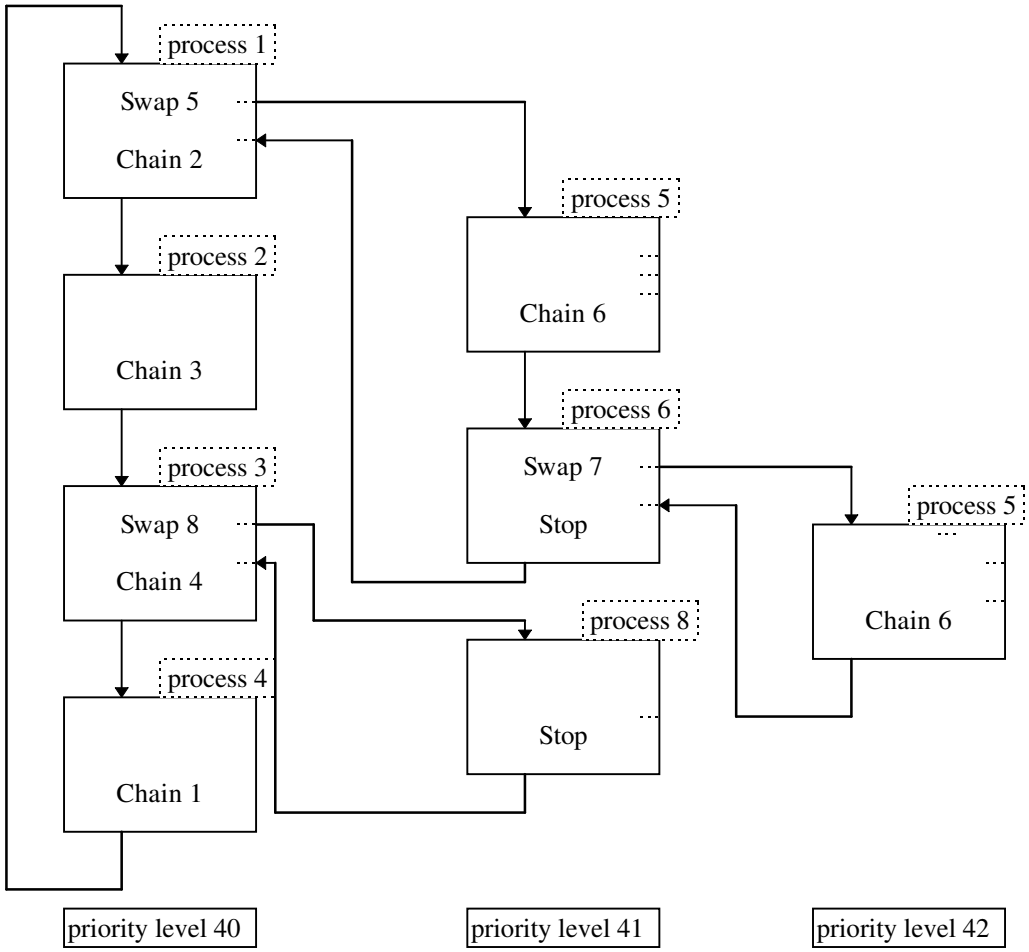


Figure 4.1.3. Process chaining and swapping

Other systems which permit dynamic allocation of memory allow the processes themselves to initiate program segment transfers, either by chaining or by overlaying. In chaining the -process is divided into several segments which run sequentially. On completion of one segment the next segment is

loaded from memory into the area occupied by the previous segment; any data required to be passed is held either on the disk or in a common area of memory.

Process swapping involves one process invoking another process: the first process is transferred to backing storage and the second process brought into memory and made available to run. The procedure. is shown in Figure 4.1.3. Process 1 invokes process 5 by swapping it into priority level 41 and in turn process 5 chains process 6 into level 41. Process 6 swaps process 7 into level 42. When process 7 terminates the operating system returns control to process 6 and when it terminates control is returned to process l. It should be noted that process 1 remains suspended until process 6 terminates and similarly process 6 is suspended until process 7 terminates.

The difference between chaining and overlaying is that in overlaying a part of the process, the root process, remains in memory and the various segments are brought into an overlay area of memory. In a multi-tasking system there may be several different overlay areas each of which may be shared by several processes. A typical arrangement is shown in Figure 4.1.4. in which it is assumed that two processes (1 and 15) have overlay segments; each process maintains a root segment and overlay area and the various overlays are loaded into and out of the overlay areas.

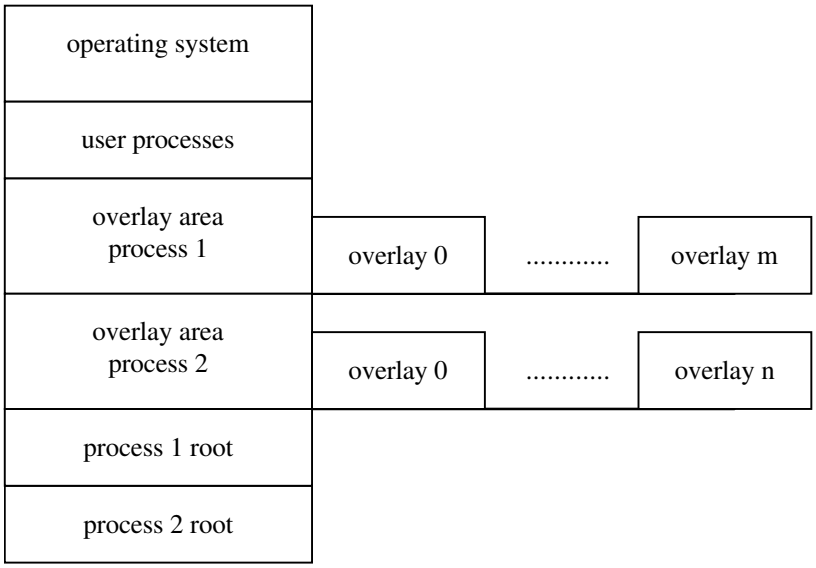


Figure 4.1.4. Process overlaying

Many of the real-time operating systems which provide facilities to swap processes between fast access memory and backing storage were designed for computer systems with small amounts of fast access memory, typically 32K words, and limited memory address space (this is limited by the address lines available on the bus and the CPU architecture). Some of these computer systems have been extended to operate with larger memories by increasing the number of address lines on the bus system and providing memory management units. The way in which a memory management unit operates is to map areas of physical memory onto the actual memory address space. As a consequence the operating systems have been modified to support a larger memory and many do this by using the extended memory to hold the process overlays with a requirement that the root process still be located in the first segment of the memory:

Dynamic memory allocation is complex to handle and should be avoided wherever possible in embedded real-time systems. RAM is now so cheap that the cost of adding extra memory is usually much less than the cost of programming to provide dynamic memory allocation. It should never be used in safety-critical applications.

### 4.1.1. Code sharing

In many applications the same actions have to be carried out in several different processes. In a conventional program the actions would be coded as a subroutine and one copy of the subroutine would be included in the program. In a multi-tasking system each process must have its own copy of the subroutine or some mechanism must be provided to prevent one process interfering with the use of the code by another process. The problems which can arise are illustrated in Figure 4.1.1.1. Two processes share the subroutine S. If process A is using the subroutine but before it finishes some event occurs which causes a rescheduling of the processes and process B runs and uses the subroutine, then when a return is made to process A, although it will begin to use subroutine S again at the correct place, the values of locally held data will have been changed and will reflect the information processed within the subroutine by process B.

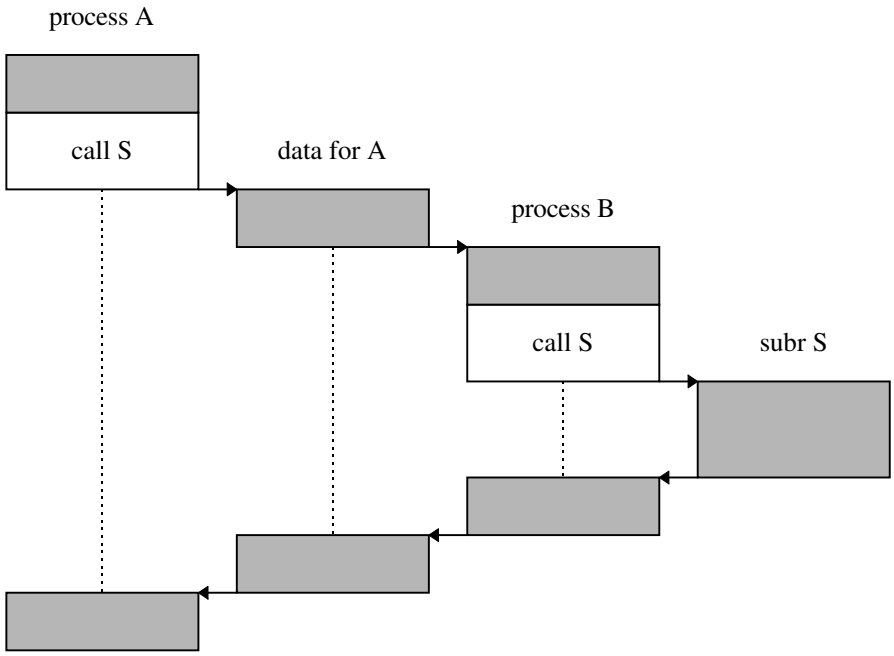


Figure 4.1.1.1. Sharing a subroutine in multi-processing system

Two methods can be used to overcome this problem:

- serially reusable code;
- re-entrant code.

### 4.1.2. Serially Reusable Code ,

As-shown in Figure 4.1.2.1., some form-of lock mechanism is placed at the beginning of the routine such that if any process is already, using the routine the calling process will not be allowed entry until the process which is using the routine unlocks it. The use of a lock mechanism to protect a subroutine is an example of the need for mechanisms to support mutual exclusion when constructing an operating system.

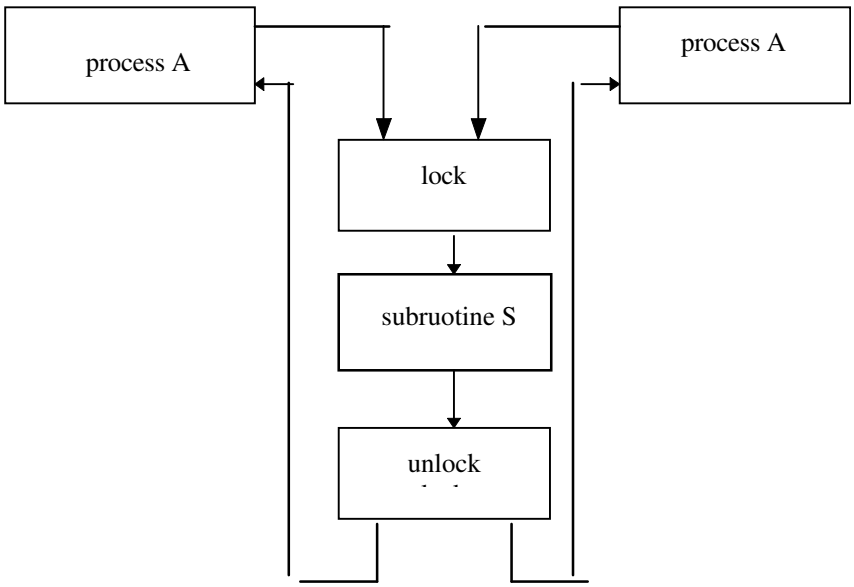


Figure 4.1.2.1. Serially reusable code

### 4.1.3. Reentrant Code

If the subroutine can be coded such that it does not hold within it any data, that is it is purely code - any intermediate results are stored in the calling process or in a stack associated with the process - then the subroutine is said to be re-entrant. Figure 4.1.2.2. shows an arrangement which can be used: the process descriptor for each process contains a pointer to a data area - usually a stack area - which is used for "the storage of all information relevant to that process when using the subroutine: Swapping between processes while they are using the subroutine will not now cause any problems since the contents of the stack pointer will be saved with the volatile environment of the process and will be restored when the process resumes. All accesses to data by the subroutine will be through the stack and hence it will automatically manipulate the correct data.

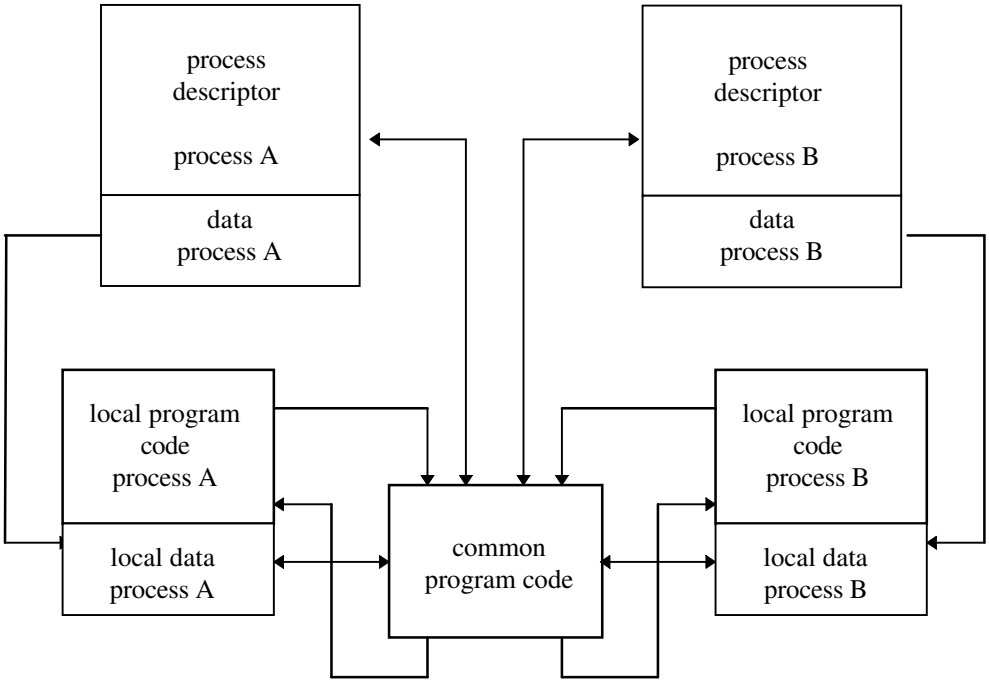


Figure 4.1.2.2. Re-entrant code for code sharing

Re-entrant routines can be shared between several processes since they contain no data relevant to a particular process and hence can be stopped and restarted at a different point in the routine without

any loss of information. The data held in the working registers of the CPU is stored in the relevant process descriptor when process swapping takes place. Device drivers in conventional operating systems are frequently implemented using re-entrant code.

## 4.2. Resource Control

One of the most difficult areas of programming is the transfer of information to and from external devices. The availability of a well-designed and implemented input/output subsystem (IOSS) in an operating system is essential for efficient programming. The presence of such a system enables the application programmer to perform input or output by. means of system calls either from a high-level language or from the assembler. The IOSS handles all the details of the devices: In a multi-tasking system the IOSS should also deal with all the problems of several processes attempting to access the same device.

A typical IOSS will be divided into two levels as shown in Figure 4.2.1. The I/O manager accepts the system calls from the user processes and transfers the information contained in the calls to the device control block (DCB) for the particular device. The information supplied in the call by the user process will be for example, the location of a buffer area in which the data to be transferred is stored (output) or is to be stored (input); the amount of data to be transferred; type of data, for , example binary or ASCII; the direction of transfer; and the device to be used.

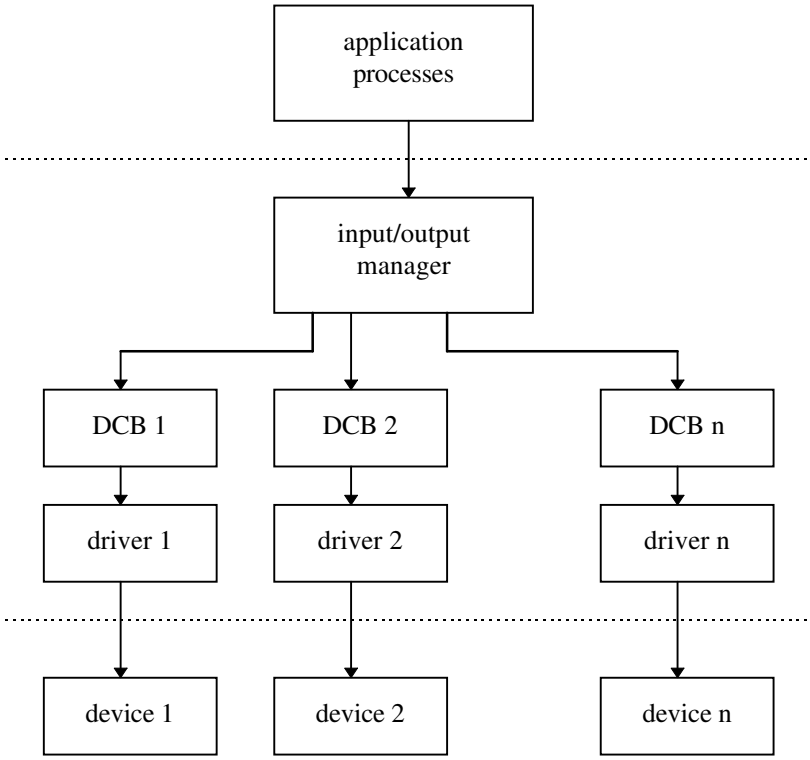


Figure 4.2.1. General structure of the input/output subsystem

The actual transfer of the data between the user process and the device will be carried out by the device driver and this segment of code will make use of other information stored in-the DCB. A separate device driver may be provided for each device or, as is shown in Figure 4.2.1. a single driver may be shared between several devices; however, each device will require its own DCB. The actual data transfer will usually be carried out under interrupt control:

Typically a DCB will contain the information shown in Table 4.2.1. The physical device name is the name by which the operating system recognises the device and the type of device is usually given in the form of a code recognised by the operating system. The operating system will normally be supplied with DCBs for the more common devices. The DCBs may require modifying to reflect the addresses used in a particular system, although many suppliers, adopt the policy of using standard

addresses both for the physical address of the device on the bus and for the interrupt locations and interrupt handling routines. The addition of non-standard devices will require the user to provide appropriate DCBs. This process is usually made reasonably simple by providing source code for sample DCBs which can be 'modified to meet particular needs.

| device-related information | data-related information |
|---|---|
| physical device name | data address areas |
| type of device | bytes to be transferred |
| device address | current byte count |
| device status | binary/ASCII |
| interrupt address | |
| interrupt service routine address | |

Table 4.2.1. Device control block

In a multi-tasking system provision has to be made to deal with overlapping requests for a particular device, for example several processes may wish to send information to the log device - typically a printer. The normal way of handling output to a printer in a single-user environment is to send a record, that is one line, at a time, a return being made to the user program between each line. If this is done in a multi-programming system and the printer is not allocated to the specific program then there is a danger of the output from the different programs becoming intermingled. The solution usually adopted in a multi-user environment is to spool the output, that is it is intercepted by the operating system and stored in a file on the disk. When the program terminates or the user signs off, the contents of the spool file associated with that program or that user are printed out.

A similar solution can be used in a multi-tasking environment providing the use process can force the printing out of the spool file for that process. This addition is needed because in a real-time multi-tasking system processes may not terminate. Although spooling provides user processes with the ability to control the interleaving of output, there is still the problem of what action to take if the device is in use when a use process makes a request to use it. There are several possible solutions:

- Suspend the process until the device becomes available.
- Return immediately to the process with information that the device is busy and leave it to the process to decide what action to take - normally to call a delay and try again later.
- Add the request to a device request queue and return to the calling process; the calling process must check at some later time to see if the request has been completed.

There are advantages and disadvantages to each method and a good operating system will provide the programmer with a choice of actions, although not all options will be available for every device.

*Option* 1 is referred to as a non-buffered request, in that the user process and the device have to rendezvous. In some ways it can be thought of as the equivalent of hardware handshaking - the user process asks the device `are you ready?' and waits for a reply from the device before proceeding.

*Option* 2 is the equivalent of polling and is rarely used.

*Option 3 is* referred to as a buffered request. It is a form of message passing: the user process passes to the IOSS the equivalent of a letter - this consists of both the message and instructions about the destination of the message - and then the user process continues on the assumption that eventually the message will be delivered, that is sent to the output device. Usually some mechanism is provided which enables the user process to check if the message has been received; that is a form of recorded delivery in which the IOSS records that the

message has been delivered and allows the user process to check. Buffered input is slightly different in that the user process invites an external device to send it a message - this can be considered as the equivalent of providing your address to a person or to a group of people. The IOSS will collect the message and deliver it but it is up to the user process to check its `mail box' to see if a message has been delivered.

### 4.2.1. The input/output subsystem

Given a real-time system whose devices are handled by an IOSS of an RTOS which supports both computer peripherals - VDUs, printers, disk drives, etc. - and process-related peripherals - analogue and digital input and output devices. The system commands used to access the IOSS functions are listed in Table 4.2.1.1. In addition to the commands listed in the table there are commands for analogue output, for pulse output devices and for incremental output devices.

| |
|---|
| - disk transfer request - buffered |
| - disk transfer request - non-buffered |
| - call to check for completion of buffered request |
| - input request from keyboard device - buffered |
| - call to check for completion of input request |
| - call to (a) request system data area, i.e. spool area |
| (b) request user data area |
| (c) check status of device |
| (d) check if user data area is free |
| - request output of message to printer or terminal - buffered |
| - find and reserve area of memory external to the calling process |
| - release area of memory found using the previous call |
| - check if the previously requested buffered scans have been completed |
| - request non-buffered, non-priority analogue scan |
| - request non-buffered, priority scan |
| - request buffered; non-priority scan |
| - request buffered, priority scan |
| - request for a normal, digital output, non-priority |
| - request for a normal digital output, priority |
| - request for a timed digital output, non-priority |
| - request for a timed digital output, priority |

Table 4.2.1.1 IOSS system commands for RTOS

The IOSS system manager maintains a device request queue for each device and is responsible for interpreting the user process request and placing the appropriate information in the device request queue. If the request is a buffered request, then a return is made immediately to the calling process. If the request is non-buffered, then the IOSS manager changes the status of the calling process to LOCKED OUT and jumps to the dispatcher to begin the search for other work. The IOSS manager, in addition to dealing with requests; has to take action on the completion of a transfer. The driver associated with a given device signals the IOSS manager on completion of a transfer. The IOSS manager, for non-buffered requests, sets the status of the user process which made the request to ON. For buffered requests there are two possible actions: if the calling process has checked to see if the action has been completed before it was completed it will have been placed in the LOCKED OUT state and hence the IOSS treats it as a non-buffered request. If completion occurs prior to a check for completion by the user process, then the IOSS records that the transfer has been completed. When a check. is made a return to the calling process will be made with an indication that the transfer is complete. The actual detail of the actions on completion varies for the different types of device.

In addition to dealing with the above, the IOSS manager following completion of a transfer by a device has to check if further requests are waiting in the device request queue; if they are it transfers information to the DCB and initiates the start of transfer before returning to the dispatcher.

### 4.2.2. Output to Printing Device

The RTOS provides the programmer with a choice o£ spooling mechanisms:

*System data* areas: a number of fixed sized areas on the disk are provided and are identified by a tag number;

*User data* areas: the user may define data areas of any size which can be. in a memory or on the disk and are again identified by a tag number.

The system data areas are made available to any user process which requests a data area; the request can be only for a system data area, not one with a specified tag number. A user data area can be assigned to a particular user process. (Note that in this system the assignment is by implication only; all user processes have to agree that a given tag number applies to a given data area and have to agree that use will be restricted to a given process.) The advantages of a user data area are:

1. the area can be in memory or on disk;'
2. the area can be of any size;
3. if use of the area is restricted to one process then it can contain a mixture of permanent and variable data and the user process only needs to transfer the information which has been changed since the last output.

The sequence of operations to be carried out in order to output a message via a user data area. Note that the operating system treats the user process differently when it checks for completion of input compared with the check for completion of output. The reason is that it assumes that a check for completion of input will be made only when there is no other work for the process to do; hence if a return to the process on non-completion was made then all the process can do is delay for a short period and then check again - an inefficient procedure. However, on output a process which finds that the previous output is not complete may be able to' take some other action, for example set up another data area with a further request `and continue.

### 4.2.3. Device queues and priorities

In a real-time system a simple device queue based on a first-in-first-out organisation can cause problems in that the process requesting a device effectively loses its priority. Figures 4.2.1.1 and 4.2.1.2. illustrate this. In Figure 4.2.1.1. a number of processes are queued waiting for the printer and one process (76) is already using the device. The higher-priority processes including the very high-priority process 5 will not gain access to the printer until process 76 releases it. If the processes have made a non-buffered request they will be locked out until they reach the head of the printer queue. However, if process 5 has made a buffered request it will be able to continue and, if it runs frequently, then after a short period of time the printer queue will contain several requests from process 5 as is shown in Figure 4.2.1.1. If the system is not overloaded the printer will eventually catch up with the output from process 5. The delay between the requests from process 5 and the eventual output on the printer can be reduced if the printer queue is .organised on: priority basis.

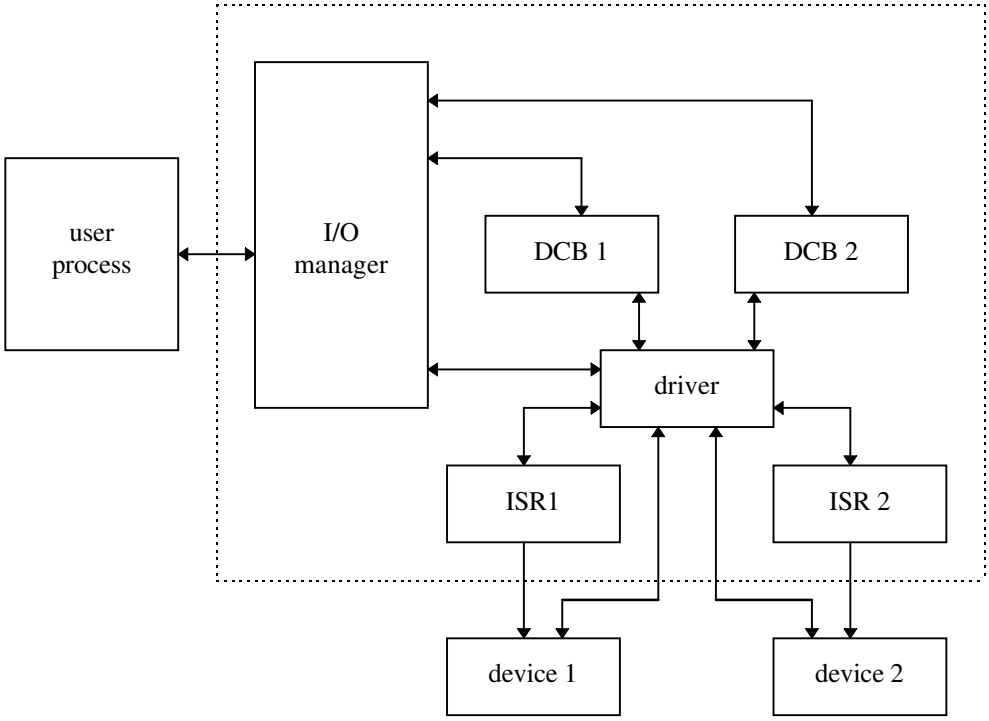


Figure 4.2.1.1. Detailed structure of the input/output subsystem

The position regarding priorities becomes even more complicated when the ISO deals with many devices and hence has several device queues. Decisions on the order in which device queues are serviced are complicated and difficult.

## 4.3. Process Co-Operation and Communication

In real-time systems processes are designed to fulfil a common purpose and hence they need to communicate with each other: However, they may also be in competition for the resources of the computer system and this competition must be regulated. Some of the problems which arise have already been met in considering the input/ output- subsystem and they involve:

- mutual exclusion,
- synchronisation;,
- data transfer.

### 4.3.1. Fundamentals

#### Mutual exclusion

A multi tasking operating system allows the sharing of resources between several concurrently active processes. This does not imply that the resources can be used simultaneously. The use of some resources is restricted to only one process at a time. For others, for example a re-entrant code module, several processes can be using them at the same time. .The restriction to one process at. a time has to be made for resources such as input and output devices, otherwise there is a danger that input intended for one process could get corrupted by input for another process. Similarly problems can arise if two processes-share a data area and both processes can write to the data area.

In abstract terms mutual exclusion can be expressed in the form;

*remainder 1*

*pre protocol (necessary overhead)*

*critical section*

*post protocol (necessary overhead)*

*remainder 2*

where

remainder 1 and remainder 2 represent sequential code that does not require: access to a particular resource or to a common area of memory.

critical section is the part of the code which must be protected from interference from another process.

pre protocol and post protocol called before and after the critical sections are code segments that will ensure that the critical section is executed so as to exclude all other processes.

To benefit from concurrency both the critical section and the protocols must be much shorter than the remainders, so that the remainders represent a significant body of code that can be overlapped with other processes. The protocols represent an overhead which has to be paid in order to obtain concurrency.

It is implicit in concurrent programming that there is `loose connection' or `low coupling' between processes. Low coupling increases reliability since an error in one process which causes an abnormal termination of that process should not, if there is low coupling, cause other processes in the system to fail. In abstract terms this can be expressed as the requirement that an abnormal termination in the code forming the remainder should not affect any other process. It would be unreasonable to demand that a failure of the protocol or the critical sections did not affect another process, since the critical section represents the code by which communication or sharing of a resource with another process is taking place.

In considering solutions to mutual exclusion problems it is normal to assume that a number of so-called primitive instructions exist. The correctness of the primitives is assumed to be guarantied by the language or operating system supporting them. A basic assumption is that a primitive forms an indivisible instruction and hence the process invoking a primitive is guarantied not to be pre-empted during the execution of the primitive. At the operating system level the basic primitive instruction is the machine code instruction and there is reliance on the CPU hardware to support the indivisibility of an instruction. Furthermore there is also reliance on the hardware implementation of mutual exclusion on the basic access to memory. For example, in a common memory system there will be some form of arbiter which will provide for mutual exclusion in accessing an individual memory location.

**Semaphore**

The most widely used form of primitive for the purposes of mutual exclusion is the binary semaphore. A binary semaphore is a condition flag which records whether or not a resource is available. If, for a binary semaphore **s**, s=1, then the resource is available and the process may proceed; if s = 0 then the resource is unavailable and the process must wait. To avoid the processor wasting time while a process is waiting for a resource to become available there has to be a mechanism for suspending the running of a process when it is waiting and for recording that the process is waiting for a particular semaphore. A typical mechanism for doing this is to associate with each semaphore a queue (often referred to as a condition queue) of processes that are waiting for a particular semaphore. It is assumed that such a queue is created and a semaphore is declared and processes can be removed from and added to the queue.

There are only three permissible operations on a semaphore, *Initialise, Secure*, and *Release*, and the operating system must provide the following procedures:

Initialise (s: BinarySemaphore, v: INTEGER): set semaphore **s** to value of **v** ( v=0 or 1 ).

Secure(s): if s =1 then set s : = 0 and allow the process to proceed, otherwise suspend the calling process.

Release(s) : if there is no process waiting for semaphore s then set s : =1, otherwise resume any process that is waiting on semaphore s :

The operations *Secure(s)* and *Release(s)* are system primitives which are operation carried out as indivisible operations and hence the testing and setting of the condition flag are performed effectively as one operation.

Given an example that considers a process which wishes to access a printer. In Figure 4.2.1.1. the underlying operations which take place as several processes attempt to access the same resource (assumed to be a printer) are shown. The binary semaphore, *printerAccess*, is initialised to the value 1 in step 1. As part of this process a three-item record with the semaphore value set to 1 and the pointers to the head and tail of the semaphore queue set to null is created. In step 2, Process A performs a *Secure(printerAccess*) operation and the semaphore value is set to 0: Since there was no other process waiting, Process A is allowed to use the resource. Sometime later Process A suspends and Process B performs a *Secure(printerAccess),* step 3. Since *printerAccess*=0 it cannot continue and is added to the semaphore queue by inserting pointers to the process descriptor for Process B, in the semaphore control block. If Process C now performs a *Secure(printerAccess),* step 4, then the pointer in the process descriptor for Process B is filled in with the address of the TD for Process C and the tail pointer entry in the semaphore control block is filled in to point to Process C. When Process A performs the *Release(printerAccess)* operation, step 5, Process B is removed from the semaphore queue and then obtains access to the resource. At step 6, Process B performs the Release(*printerAccess)* operation and hence Process C is allowed to run and at step 7 when Process C performs Release(printerAccess) the value of the semaphore is set to l.

**The Monitor**

Although the binary and general semaphore provide a simple and effective means of enforcing mutual access they have one weakness: in use they are scattered around the code. Each process that requires access to a particular resource has to know the details of the semaphore used to, protect that resource and to use it. The onus on correct use is placed on the designer of each process. In a small system this causes few problems but in larger, more complex systems where the processes involved may be divided between several people use of the semaphore becomes more. difficult and the probability of. introducing errors increases.

An alternative solution which associates the control of mutual exclusion with the resource rather than with the user process is the monitor. A monitor is a set of procedures that; provide access to data or to a device. The procedures are encapsulated inside a module that has the special property that only one process at a time can be actively executing a monitor procedure. It can be thought of as providing a fence around critical data. The operations which can be performed on the data are moved inside the fence as well as the data itself. The user process thus communicates with the monitor rather than directly with the resource.

Two procedures, WriteData and ReadData, provide access to the data. These procedures represent gates through which access to the monitor is obtained. The monitor prevents-any other form of access to the critical data. A process wishing to write data calls the procedure *WriteData* and as long as no other process is already accessing the monitor it will be allowed to enter and write new data. If any other process was already using either the *WriteData* or *ReadData* operations then the process would be halted at the gate and suspended, since only one process at a time is allowed to be within the monitor fence.

The advantage of a monitor over the use of semaphores or other, mechanisms to enforce mutual exclusion is that the exclusion is implicit; the only action required by the programmer of the process

requiring to use the resource is to invoke the entry to the monitor. If the monitor is correctly coded then an applications program cannot use a resource protected by a monitor incorrectly.

### 4.3.2. Intertask Communication

Issues of synchronisation and communication can be divided into three areas:

synchronisation without data transfer;

data transfer without synchronisation; and

synchronisation with data transfer.

#### 4.3.2.1. Process Synchronisation Without Data Transfer

Frequently one wishes to be able to inform another process that an event has occurred, or to set a process to wait for an event to occur. No data needs to be exchanged by the processes: A mechanism that enables this to be done is the so-called signal. A signal **s** is defined as a binary variable such that if s =1 then a signal has been sent but has not yet been received.

Associated .with a signal is a queue and the permissible operations on a signal are:

***Initialise****(s: Signal : v: INTEGER)* set **s** to the value of v (0 or 1 ).

| | |
|---|---|
| *Wait(s)* | if s=1 then s : =0 else suspend the calling process and place it in the condition queue **s**. |
| *Send(s)* | if the condition queue **s** is empty then s : =1 else transfer the first process in the condition queue to the ready queue. |

Clearly a signal is similar to a semaphore; in fact the difference between the two is not in the way in which they are implemented but in the way in which they are used. A semaphore is used to secure and release a resource and as such the calls will both be made by one process; a signal is used to synchronise the activities of two processes and one process will issue the send another process the wait (Note that a signal is sometimes implemented such that if a process is not waiting it has no effect, that is the receipt of a signal is not remembered.)

In practice two important additions to the basic *signal* mechanism are required: one is the facility to check to see if a process is waiting to send a signal, and the other is to be able to restrict the length of time which a process waits for a signal to occur. In a real-time applications it is rarely correct for a process to be committed to wait indefinitely for an event to occur. The standard Wait(s) commits a process to an indefinite wait.

#### 4.3.2.2. Data Transfer Without Synchronisation

RTOSs typically support two mechanisms for the transfer or sharing of data between processes: these are the pool and the channel.

**Channel** supports communication between producers and consumers of data. It can contain one or more items of information. Writing to a channel adds an item without changing items already in it. The read operation is destructive in that it removes an item from the channel. A channel can become empty and also, because in practice its capacity is finite, it can become full.

**Pool** is used to hold data common to several processes, for example tables of values or parameters which processes periodically consult or update. The write operation on a pool is destructive and the read operation is non-destructive.

It is normal to create a large number of pools so as to limit the use of global common data areas. To avoid the problem of two or more processes accessing a pool simultaneously mutual exclusion on pools is required. The most reliable form of mutual exclusion for a pool is to embed the pool inside a monitor. Given that the read operation does not change the data in a pool there is no need to restrict read access to a pool to one process at a time.

Channels provide a direct communication link between processes, normally on a one-to-one basis. The communication is like a pipe down which successive collections of items of data - messages - can pass. Normally they are implemented so that they can contain several messages and so they act as a buffer between the : processes. One process is seen as the producer of information- and the other as the consumer. Because of the buffer function of the channel the producer and consumer processes can run asynchronuously.

There are two basic implementation mechanisms for a channel:

queue (linked list); and

circular buffer.

The advantage of the queue is that the number of successive messages held in the channel is not fixed. The length of the queue can grow, the only limit being the amount of available memory. The disadvantage of the queue is that as the length of the queue increases the access time, that is the time to add and remove items from the queue, increases. For this reason and because it is not good practice to have undefined limits on functions in real-time systems queues are rarely; used.

The circular buffer uses a fixed amount of memory, the size being defined by the designer of the application. If the producer and consumer process's run normally they would typically add and remove items from the buffer alternately. If for some reason one or the other is suspended for any length of time the buffer will either fill up or empty. The processes using the buffer have to check, as appropriate, for buffer full and buffer empty conditions and suspend their operations until the empty or full condition changes.

There is an example; an alarm scanning process which for a period of time produces data at a rate much greater than that at which the logging process can print it out. A buffer is needed to store the data until the consuming process is ready to take it. The system is shown in Figure xx. It is assumed that the buffer is bounded, that is of finite size, and that the operation of storing an item of data in it is performed by the call Put(x) and an item of data is removed by the call Get(x).

Since the buffer is of finite size it is necessary to know when it is full and when it is empty. The following function calls are used:

*Full* - which returns the value true if the buffer is full;

*Empty* - which returns the value true if the buffer is empty.

When the producer and consumer are formed by separate processes which share a common buffer area. The producer operates in an endless cycle producing some item **x** and waiting until the buffer is not full to place **x** in the buffer. The consumer also operates in an endless cycle waiting until the buffer is not empty and removing item **x** from the buffer. This solution is not satisfactory for two reasons:

1. the *Put(x)* and *Get(x)* are both operating on the same buffer and for security of the data simultaneous access to the buffer cannot be allowed - the mutual exclusion problem;
2. both the producer and the consumer use a `busy wait' in order to deal with, the buffer full and buffer empty problem.

The first problem can be solved using the semaphore, with the operations secure and release. The second problem can be solved by using the signal mechanism described above.

Because of the need to test for empty and full and to suspend the process if one or the other condition appertains then although transfer of a data item is not synchronised there is process synchronisation on the buffer full and buffer empty conditions. In this example the critical code is enclosed between secure and release operations but it is essential that the *bufferAccess* semaphore is released before executing the *Wait(nonFull)* or *Wait(nonEmpty)* primitives. If this is not done the system will deadlock. For example, if the *Producer* execute *Wait(nonFull)* while holding the access

rights to the buffer then the buffer can never become non-full since the only way it can is for the *Consumer* to remove a item of data, but the *Consumer* cannot gain access to it until it is released by the *Producer.*

Both semaphores and signals can be generalised to allow a semaphore or a signal variable to have any non-negative integer value - in this form they are :sometimes referred to as counting semaphores.

#### 4.3.2.3. Synchronisation With Data Transfer

There are two main forms of synchronisation involving data transfer: The first involves the producer process simply signalling to say that a message has been produce and is waiting to be collected, and the second is to signal that a message is read and to wait for the consumer process to reach a point where the two processes can exchange the data.

The first method is simply an extension of the mechanism used in the example in the previous section to signal that a channel was empty or full. Instead of signalling these conditions a signal is sent each time a message is placed in the channel. Either a generalised semaphore or signal that counts the number of send and waits, or a counter, has to be used.

#### Liveness

An important property of a multi-tasking real-time system is Iiveness. A system (or a set of processes) is said to possess liveness if it is free from

livelock,

deadlock, and

indefinite postponement.

*Livelock is* the condition under which the processes requiring mutually exclusive access to a set of resources both enter *busy wait* routines but neither can get out of the *busy wait* because they are waiting for each other. The CPU appears to be doing useful work and hence the term livelock.

*Deadlock is* the condition in which a set of processes are in a state such that it is impossible for any of them to proceed. The CPU is free but there are no processes that are ready to run.

The detection of deadlock or the provision of resource sharing commands in such a way as to avoid deadlock is the responsibility of the operating system.

*Indefinite postponement is* the condition that occurs when a process is unable to gain access to a resource because .some other process always gains access ahead of it.

### 4.3.2. Resource sharing and deadlock

All processes in the system uses system resources. At the least, these resources consist of a processor and storage for the program code. More usually the resources would include disc files and peripheral devices. Whenever two or more processes compete for the use of a set of resources, a possibility of deadlock exists. Imagine that process A has the exclusive use of resource R and requests use of resource Q. Process B has the exclusive use of resource Q but requests use of R. Neither process can continue - they are deadlocked.

Operatings systems app1y one of three strategies to solve the deadlock problem :

1. prevent deadlock by ensuring that the conditions necessary for deadlock cannot occur;
2. detect deadlock when it occurs and instigate recovery action;
3. anticipate that deadlock may occur and try to avoid the situation

The simplest strategy is to demand that any process be allocated all of its resources at one time (normally when it starts). This approach avoids the possible deadlock, but it does not allow for the

optimum use of system resources. Complex deadlock-avoidance algorithms have been designed and applied, in an attempt to allow more efficient sharing of resources betvween the competing processes. However, in a real-time system, the processing time required for the more sophisticated algorithms may prove unacceptable.

To use the simplest strategy the ideal process for allocating the resource is the high-level scheduler. The configuration and status of resources in the system would be described by system data structures. Each of these data structures (pools) would carry out its resource allocation process via dialogue with the resource monitors. In a real-time system, the resource-allocation responsibilities of the scheduler would focus on avoiding deadlock situations. Obviously, time limits permitting; it would be helpful if the scheduler could also cater for the optimum usage of system resources.

The dispatcher would be a simple process primarily concerned with swapping the volatile environment of the process that is indicated in the process descriptor pool as the next to be run. The other three processes would use their particular knowledge of system status to manipulate the process descriptors so that they reflect the current priority requirements.