3. PROCESSES MANAGEMENT and SCHEDULING

3.1. Process Management

3.1.1. Process or Task

A process or a task is an instance of a program in execution. It is the smallest software that is individually schedulable by a kernel or an operating systems. The process consists of code and data and described by attributes and state. The code includes machine instructions and system calls. Attributes associated wit the process are assigned by the system programmer or the operating system itself. They contain for example software priority and access rights. The operating system views the execution of a process as an activity going through particular states
Each multiprogramming or multitasking operating system keeps track of all active processes and allocates the system's resources to them according to policies used to meet objectives of applications.

There are two different terms to denote the concept of a sequence of code in execution. They are process or task. The term process may be somewhat confusing when used to describe a multiprogramming environment, because multiprocessing is usually understood to mean a system with multiple hardware resources, i.e. processors, rather than a single processor with multiple processes running concurrently. Where necessary to distinguish it from multiprogramming, it is recommended to the term multiprocess to denote an operating system that supports concurrent execution of programs on a single processor without supporting memory and file management. This form of operation is also called multitasking. Multiprogramming is more general concept that denotes an operating system that provides memory and file management, in addition to supporting concurrent execution of programs.

3.1.2. Functions of Process Management

The basic functions of kernel to manage process are as follows :

- creating and removing (destroying) processes,
- controlling the execution of processes,
- handling exceptional conditions arising during execution of a process for example, interrupts,
- allocating hardware resources among processes, i.e. to schedule an allocation of CPU time to each process; and to perform the context switch, that is to save the status of the process that is currently using the CPU and restore the status of the process that is being allocated CPU time,
- providing communication among processes,
- to keep a record of the state of each process;

In most real-time operating systems the kernel dealing with the process management functions is split into two parts: a scheduler which determines which process is to run next and which keeps, and a dispatcher which performs the context switch.

A process is activated by the operating after receiving the corresponding system call, for example RUN. The operating system answers by creating a process. This activity consists of creation and initialisation of operating system data structures that are used to control and monitor the progress of the process. After creating a process, it becomes active and eligible to compete for system resources such a processor and I/O devices. A process cycles between periods of execution and of waiting for resources. When a running process id suspended, the operating system may schedule another process for execution. Each active process is an individually schedulable activity. A number of active processes are different in various stages of execution at any point in time. A particular process usually go through various states, such as running or suspended, a number of times before

terminating and departing the system. The operating system dynamically tracks the state of each process and records all the state changes as they occur. This information is used for scheduling and other resource allocation decisions made in response to demands of active processes.

In addition to the static process structure that contains its data and program code, each process have certain attributes that assists the operating system in managing it. Process attributes include current state, scheduling priorities, access rights, and other information. Depending on the type of operating system and the application environment, division of process to be performed into processes that will be executed as independent processes and the initial assignment of process attributes may be performed by the operating system or the programmer. In other words, there are either system-defined or programmer-defined processes.

System-defined processes are commonly encountered in general-purpose multiprogramming systems, such as time-sharing. In this approach, each program required execution is treated by the operating system as an independent process. Thus, in response to the user's RUN command, the system creates a process to execute the program. Initial values of process attributes, such as scheduling priority and access rights, are assigned by the operating system at the process creation time on the basis of the related user's profile and system defaults. Processes created in this way are usually transient in the sense that they are destroyed and disposed of by the system after each run.

Programmer-defined processes are by programmers when high performance or explicit control of system activities is desired. System programs such as parts of operating systems, and real-time systems are common examples of programmer-defined processes. After dividing the application into several modules, programmers define each modules. A parent process is then added to create the environment for and to control execution of individual application processes. During program development or at run-time by embedding the related systems calls, the programmer can assign initial values of process attributes and control various aspects of process execution.

All concurrent processes sharing single resource compete with each other for allocation of system resources needed for their operation. Another fundamental relationship among concurrent processes is the cooperation. This relation holds in systems in which activity is carried out by a group of related processes, each performing a specific function. Cooperating processes exchange data and synchronisation signals necessary to perform their activities. They typically share some common attributes, in addition to interacting with each other. In many cases they have parent-children relationship. Children processes usually inherit attributes from their parents at their creation time. But these attributes may be changed at run-time by invoking the system calls. A parent process normally terminates only after all its children have done so. Both competition and cooperation of resources require proper operating system support. Competition requires careful resource allocation and protection in terms of isolation of different address-spaces. Cooperation depends on the existence of mechanisms for controlled access to shared data and the exchange of synchronisation signals. These are demanding and often conflicting requirements that must be addressed when designing multiprogramming operating systems.

3.1.3. Process States

With one processor only one process can be running at any given time and hence the other processes must be in some other state. The states, the names given to the states, and the transition paths between the different states vary from operating system to operating system. A typical state diagram is given in Figure 3.1.1

The various states are as follows (names in parentheses are commonly used alternatives):

- *existent* (dormant, off): the operating system is aware of the existence of this process, but the process has not been allocated a priority and has not been made runnable. This state is sometimes left out because it denotes processes that are not known to and therefore not tracked by the operating system. All process templates awaiting activation, as well

as the programs not yet submitted to the operating system, may be regarded as dormant.

- *ready* (runnable, on): there may be several processes in this state. The attributes of the process and the resources required to run the process must be available for the process to be placed in the Ready state. Ready process possesses all resources needed for its execution except for the processor. Processes are usually assigned with ready state immediately after creation. All ready processes are waiting to have the processor allocated to them by the operating system in order to run. An operating system or kernel module, called a scheduler, selects one ready process for execution whenever the operating system or the kernel has the control of the processor and is about to return the processor to a new process.

- *running* (active): this is the process which has contro1 of the processor. It will normally be the process with the highest priority of the processes which are ready to run. Running process has all resources needed for its execution, including the processor. On a single processor system, at most one process may be running at any point of time. The running process executes its sequence of machine instructions and may call upon the operating system to perform services such as I/O or synchronisation via signal exchange on its behalf. Depending on the particular scheduling policy in effect, the operating system may return control to the running process after performing the service, or the operating system may schedule another process, if any is ready to run.

- *suspended* (waiting, locked out, delayed): the execution of processes placed in this state has been suspended because the process requires some resource which is not available or because the process is waiting. for some signal from the plant, for example input from the analog-to-digital converter, or because the process is waiting for the elapse of time. Such processes are normally left out from competition for execution until the related suspending condition is removed. The running process may become suspended by invoking an I/O routine whose results it needs in order to proceed or by awaiting a signal that is not yet produced. The operating system then records the reason of the suspension, so that it may resume the process at a later time when the suspending condition is removed by the actions of some other process or by arrival of an external event. If the operating system does not return to the process in execution when an interrupt invokes the operating system, then the running process has been preempted by a higher priority process. The preempted process remains in the ready state rather than in the suspended state since the only resource it lacks is the processor.

- *non-existent* (terminated): the operating system has not as yet been made aware of the existence of this process, although it may, be resident in the memory of the computer.

At any time active processes are in different states. The states of all processes and resources in the system is called as the global system state. In response to both internal and external events, the processes may rapidly change their states and form another global state of the system. A multi-programming or multi-tasking operating system continuously tracks changes of the global state that occur as a result of both internal and external events. Using the global state as an input, the operating system employs its scheduling algorithms to decide how to allocate resources to the requesting processes in such a way that the resulting system performance satisfies the design objectives.

When a process is suspended through the execution of a system call or an interrupt handling, all data, which may be changed by other processes must be saved in an area belonging to the suspended process. When the process becomes running again the saved data must be restored. This procedure is called process context switch.

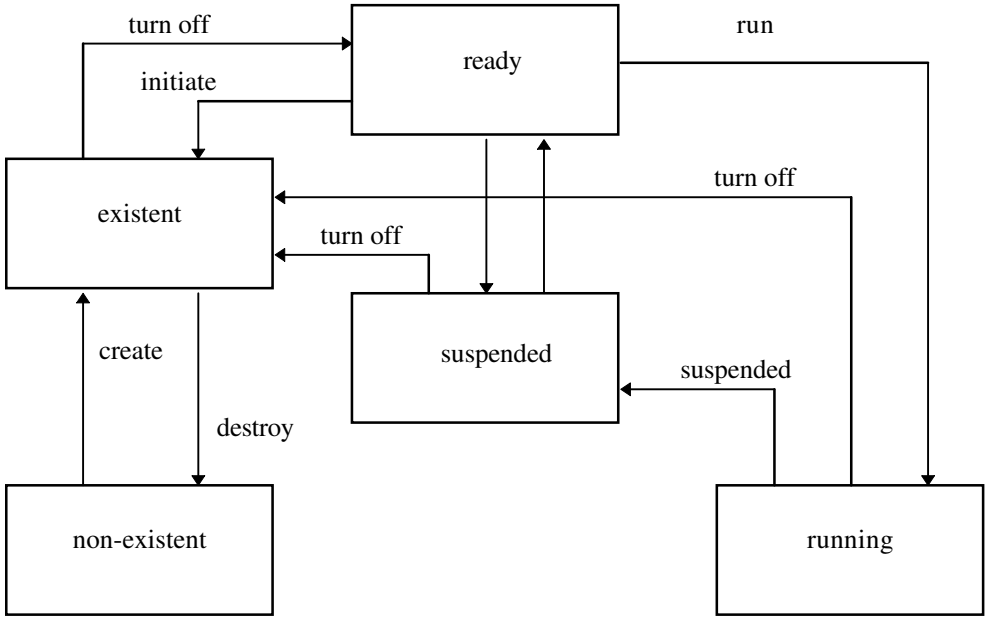

Figure 3.1.1. Process state transition

3.1.4. Process Environment

Each process environment contains the program code, the stack and the heap. The operating system groups all information that it needs about a particular process into particular data structures. The information about each process is held in two blocks of memory by the operating system. They are the process table and the process descriptor. The last block is referred to by various names: task descriptor (TD), process descriptor (PD), task control block (TCB) or task data block (TDB). Whenever a process is created or initialised the operating system creates a corresponding process descriptor to serve as its run-time description during the lifetime of the process. When the process terminates, its process descriptor PD is released to the pool of free spaces from which new process descriptors are drawn. The existent state is distinguished form other states because an existent process has no process descriptor. A process becomes known to the operating system and thus eligible to compete for system resources only when it has an active process descriptor PD associated with it. The process descriptor, as a data structure may be viewed as a record with fields for different static and dynamic process attributes.

The information held in the process descriptor will vary from system to system, but will typically consist of the following:

- process identification (ID);
- process priority (P);
- current state of process;
- area to store volatile environment (or a pointer to an area for storing the volatile environment),
- other information.
- pointer to next process in a list.

Heap and stack are used to store volatile environment of a process (or a pointer to an area for storing volatile environment).

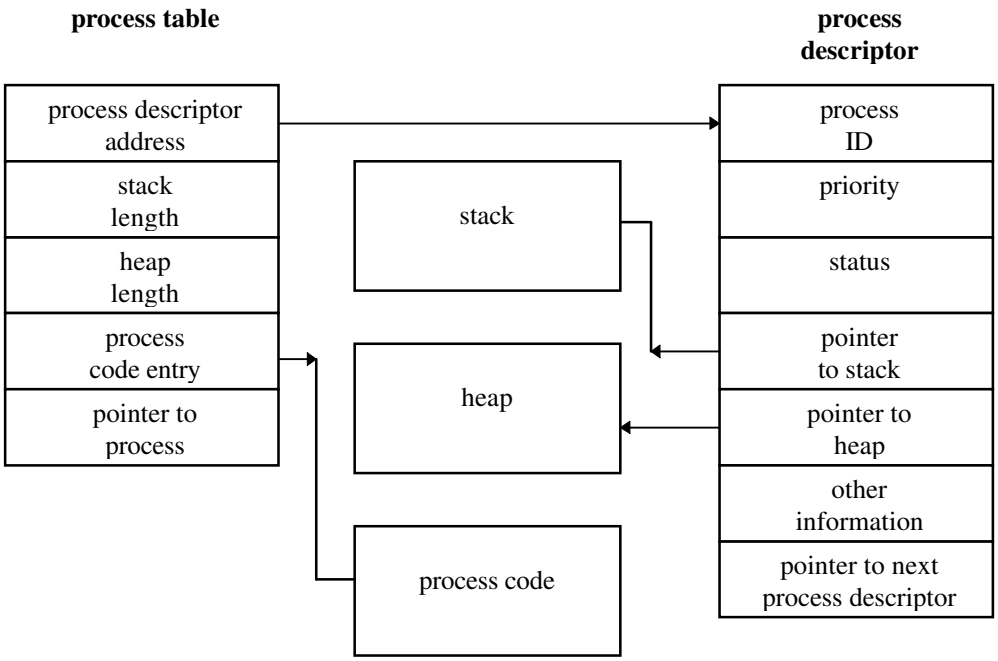

Figure 3.1.2. Process environment

Other information in the list refer to additional information about the process that a process descriptor may contain, such as a process activation record (including all registers and status flags), a list of requested and allocated resources, and accounting data. Accounting information is recorded by some systems in order to charge individual users for the processor time, I/O operations performed. Embedded and real-time systems generally do not incorporate this feature in order to ensure the precious execution cycle.

The pointer to the next process is included in the process descriptor because process descriptors are usually held in a linked list structure. In this case the actual process descriptor can be located anywhere in the memory and hence the operating system is not restricted to a fixed number of processes. Having a list structure, moving a process between lists, reordering the lists, creating and deleting processes can all be performed simply by changing the corresponding pointers. As a result, there is no need to copy or move the process descriptors themselves.

Once the process descriptor is constructed for a newly created process, it is filled with the programmer defined attributes found in the process template or specified as the parameters of the system call that creates the process. Default values for other fields are usually assigned by the operating system at this time. Whenever the process is suspected, the contents of the processor registers are saved in its activation record for subsequent restoration when the process is scheduled to run again. The state field is used by the operating system to track each and every state change of the process for as long as its process descriptor exists.

A state of a process is just a component of the global system state, which encompasses all processes and resources. In order to keep track of all processes, the operating system maintains lists of process descriptors classified by the current state of the related processes. In general, there is a ready list, which contains process descriptors of all ready processes and a suspended list. The reason for including the pointer to the next process in the list above is that the process descriptors are usually held in a linked list structure. The executive keeps a set of lists, one for each process state.

The advantage of the list structure is that the actual process descriptor can be located anywhere in the memory and hence the operating system is not restricted to a fixed number of processes as was often the case in the older operating systems which used fixed length tables to hold process state information. With the list structure moving processes between lists, reordering the lists, creating and deleting processes can all be achieved simply by changing pointers. There is no need to copy or move the process descriptors themselves.

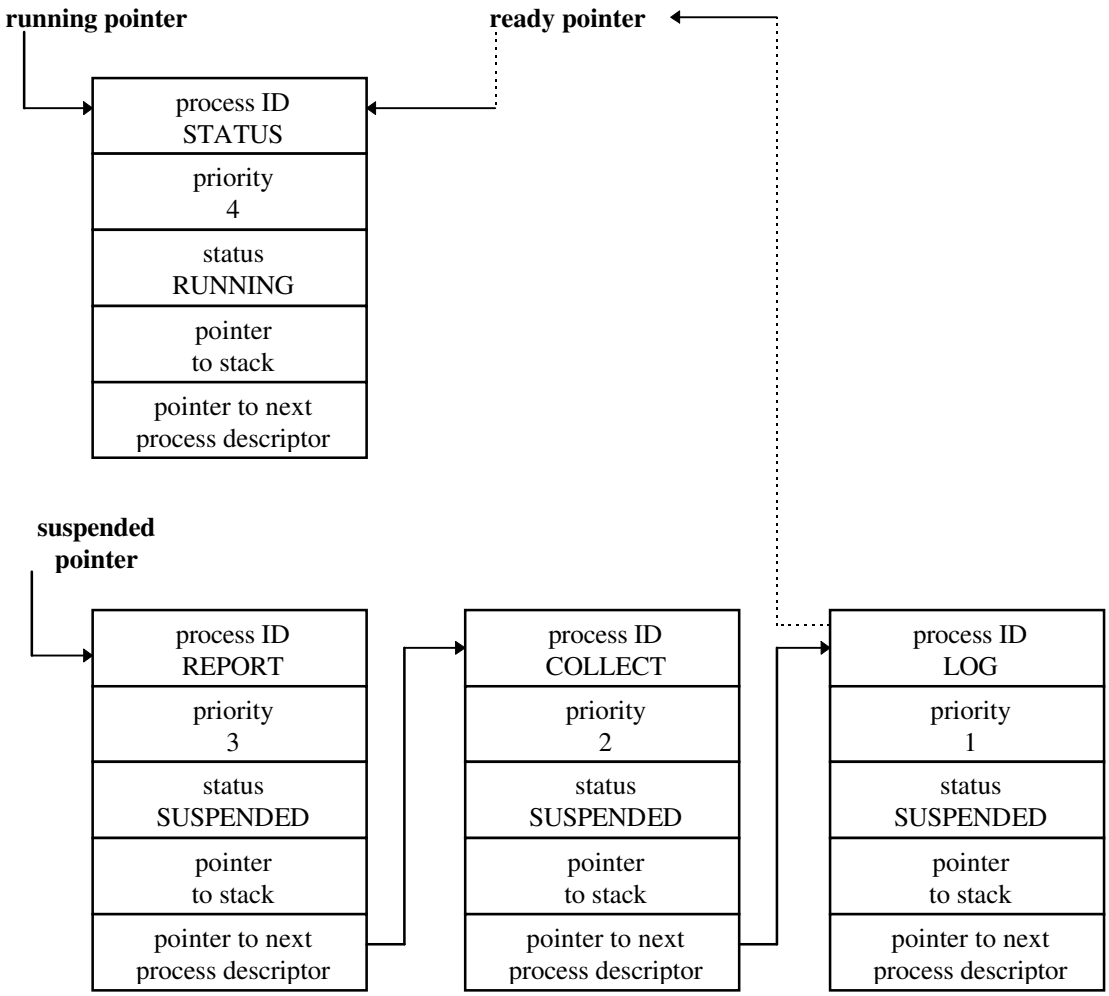

Figure 3.1.3 Processes' states before state change

By means of these lists, the operating system forms pool of processes in similar states that are likely to be examined by the operating system resource allocation procedure. For example, the scheduler should search for the next process to run only in the ready list. Operating system performance may be improved by sorting and updating these lists in ways that are most suitable for the routines that are known to operate on them.

Membership of a process descriptor in a given list is granted on the basis of the current state of its process. Whenever a process changes its state, the operating system reacts by placing the process's process descriptor in the list that corresponds to its new state. Thus, process descriptor are bound to processes but not to lists. A process descriptor is moved between lists as often as its process changes its state. Because of the frequent insertion and deletion of process descriptors, operating system process lists are almost always implemented as linked lists, sometimes with multiple pointers, in order to speed up their processing. The ready list is assumed to be sorted by process priority in order to facilitate scheduling. The suspended list is managed in FIFO order.

In Figure 3.1.3. process STATUS is running, processes REPORT, COLLECT and LOG are suspended. The process REPORT waits signal from the process STATUS. The process COLLECT waits signals from both process STATUS and process REPORT. The process LOG waits for a completion of I/O operation. The dashed line indicate the process descriptor operation necessary to reflect a change of the system's global state.

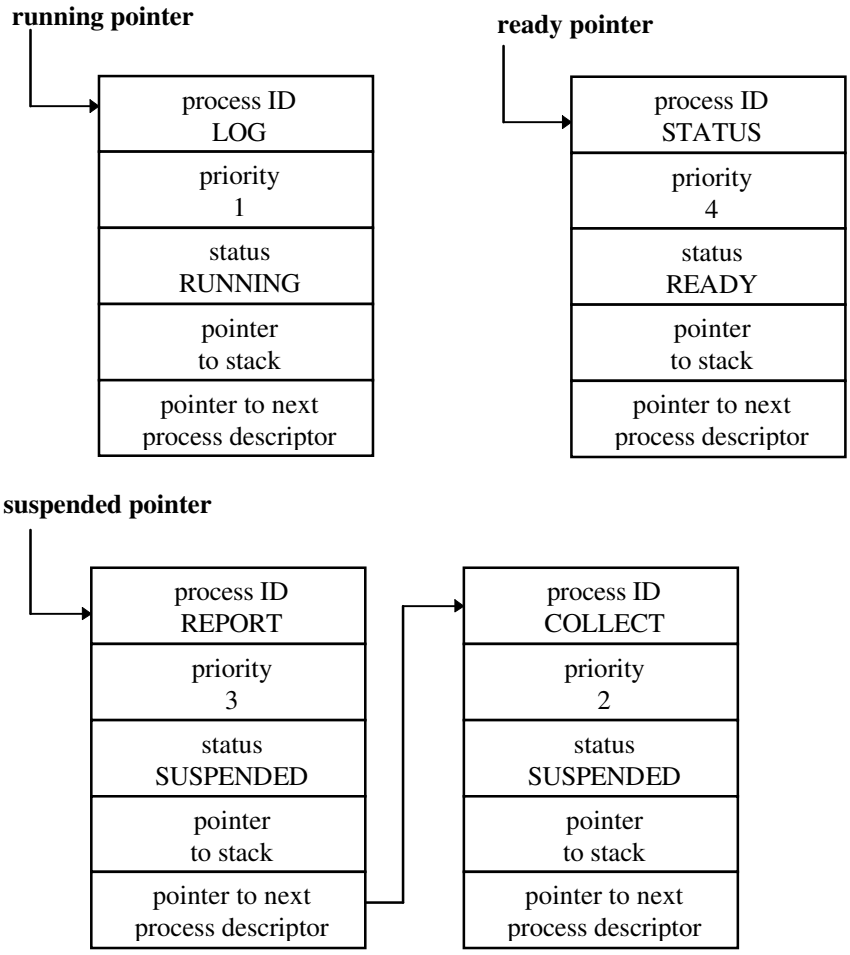

Figure 3.1.4. Processes' states after state change

In particular, when the I/O operation is completed, the corresponding interrupts alert the operating system to that event. Because the process LOG suspending condition has been removed the operating system makes the process LOG ready by moving its process descriptor into the ready list in Fig. 3.1.4. Then the scheduler is invoked to select the next running process and the process LOG is chosen because it has the highest priority. This preempts the process STATUS, the running process, whose process descriptor is moved into the ready list. Then process LOG becomes the running process.

Whenever a transition from one user process to another has to be made, the operating system intervenes and updates the system list accordingly. The first step is to record the state of the running process that is about to become suspended or preempted. The run-time state of a process typically includes the program counter, stack pointer, processors status word and all other registers used by the process. Unlike the interrupt context save, the full state must be recorded because there is no way for the operating system to know which particular registers the running process may use in its next running state. This information is stored in the activation record of the process descriptor, whose other fields are also updated at this time: state, accounting information, reason for suspension, etc. The process descriptor is then moved to the ready or suspended list, depending on whether a preemption or request for a resource that cannot be granted, resulting in suspension, has caused the process switch. In any case, the scheduler is then run to select another process for execution. The corresponding process descriptor is then used to create a running environment for the selected process. The activation record is used to set the processor registers, the interrupt mask register is set to reflect the priority of the new process, and then control is transferred to the process.

The process switching is a considerably more complex and overhead prone operation than the context switch. Given its complexity and relatively high frequency of occurrence, process switching can significantly affect performance of the operating system. For this reason, the speed of the process

switching must be high in performance-oriented system, such as real-time systems. A hardware scheme is often employed to speed up process switching is to have multiple, structurally identical sets of processor registers. The minimum is two, one for the operating system and another for user processes. A dedicated bit in the processor status word indicates the current mode of operation, supervisor or user, and the corresponding set of active registers. This scheme may be expanded in multiprocessor systems by providing multiple processors. up to one per process. Such a may system eliminates the problem of process switching, at least the overhead attributable to preemptions by allocating a dedicated processor to each active process. The user-supervisor scheme is quite common in contemporary architectures because it also provides the added benefits of system protection and address isolation.

3.1.5. System Calls for Process Management

The operating system or the kernel provides run-time services, or system calls for process management. These system calls may be invoked by user process, either directly, via supervisor calls embedded in the user's code, or indirectly, via commands typed by the user and translated into system calls.

The status of the various processes may be changed by actions within the operating system for example a resource becoming available or unavailable or by commands from the application processes.

The information that has to be stored in order to continue running a process that has been suspended by the scheduler for some reason or other comprises:

1. information required for volatile environment,
    - CPU register contents;
    - stack pointer;
    - program counter.
2. process data:
    - process stack;
    - process general work area (heap).

Typical commands or system calls are:

CREATE (process ID, attributes)

In response to this call, the operating system creates a new process with the specified or default attributes and identifier. The process cannot create itself because it would have to rerunning in order to invoke the operating system and it cannot run before being created. In response the CREATE call, the operating system obtains new process descriptor from the pool of free memory, fields with provided and/or default parameters, and insert the process descriptor into the ready list. Thus making the created process eligible to run. Some of the parameters that must be defined during the process creation include:

- level of privilege, such as system or user,
- priority,
- size and memory requirement,
- maximum data area and/or stack size,
- memory protection information
- access rights,
- other system-dependent data.

To create a process the operating system tries to find it in the process table. If it cannot be found, the operating system loads a file into memory using the requested process name as a file name. This file contains the program code and the process table of the particular process. After the process has been loaded into or located in the process table, the process descriptor is assigned to the new process. The process descriptor is automatically initialised and maintained. The process need not be concerned about the process descriptor's existence or contents. The process table contains the size of both the heap and the stack. The operating system allocates a memory area of the required size from the free memory space for them. In next phase the volatile environment of the process is set up providing the proper addresses and values. If the process uses initialised variables and/or pointers they are copied from the program code area to the proper addresses in the heap area.

The typical error messages, implying that the process was not created as a result of this call, include wrong ID (illegal, or already active), no space for process descriptor)usually transient, the call may retried later), calling process not authorized to invoke this function, etc. In the case of a temporary lack of process descriptor space, the user should alert the system manager to increase the size of the memory pool.

EXECUTION(process ID)

There is not a particular system call that starts the execution of a process. Once the program code has been loaded and the process has been created, it is ready for execution, i.e. it is in the READY queue. The process starts to compete with other processes for the processor.

The execution of process starts with system call that takes the next process out of READY queue and initiates execution. If there is no process in the READY queue the operating system waits for an interrupt and then checks the queue again.

DELETE(process ID)

Invocation of this call causes the operating system to destroy and remove the designated process from the system. Unlike CREATE, a process may delete itself or another process. The operating system reacts by reclaiming all resources allocated to the specified process (attached I/O devices, memory) closing files opened by or for the process, and performing whatever other housekeeping is necessary. Following this process the process descriptor is removed from the list. This makes the designated process dormant. The DELETE call is normally invoked as part of an orderly program termination. The process descriptor of the corresponding process is also deleted from the list of existing processes and inserted into the list of NON-EXISTENT processes.

Almost all multiprogramming operating systems allows processes to terminate themselves, at least provided none of their spawned processes is active. Operating systems differ in their attitude toward allowing one process to terminate others. The issue is one of convenience and efficiency versus system integrity. Allowing uncontrolled use of this function provides a malfunctioning or a malevolent process with the means of wiping out all other processes in the system. On other hand, terminating a hierarchy of processes in a strictly guarded system, where each process can only delete itself and the parent must wait for children to terminate first, could be a lengthy operation indeed. The usual compromise is to permit deletion of other process, but to restrict the range to members of the family, to lower-priority processes only, or some other subclass of processes.

Possible error return from the DELETE call include a child of this process is active (should terminate first), wrong ID (the process does not exist), and calling process not authorised to invoke this system call.

ABORT (process ID)

This a forceful termination of a process. While a process could conceivably abort itself, the most frequent use of this call is to remove a malfunctioning process from the system. The operating system performs much the same actions as DELETE, except that is usually furnishes a register and memory dump, together with some information about the identity of the aborting process and the reason for action. This information may be provided in a file, as a message on terminal, or as an input to the system crash analyser utility. Obviously, the issue of restricting the authority to abort other processes is even more pronounced when implementing the ABORT call.

Error returns for ABORT are practically the same as those listed in the DELETE call.

FORK(process ID)

Another method of process creation is the FORK system call. It was originally introduced as primitives for multiprocessor systems. It is used to split a sequence of instructions into two concurrently executable sequences. After reaching the identifier specified in FORK, a new process (child) is created to execute one branch of the forked code while the creating (parent) process continues to execute the other. FORK usually returns the identity of the child to the parent process and the parent can use that identifier to designate the identity of the child whose termination it wishes to wait for when invoking a JOIN operation.

JOIN(process ID)

JOIN is used to merge the two sequences of code divided by FORK. It is available to a parent process or synchronisation with a child. The JOIN is used to synchronise the caller with termination of the named process.

The relationship between processes created by FORK is rather symbiotic in the sense that they execute from a single segment of code and a child usually initially obtains a copy of the variables of its parent.

SUSPEND(process ID)

This system call suspends the designated process indefinitely and placed in the suspended state. It does, however remain in the system. A process may suspend itself or another process when it is authorised to do so by virtue of its level or privilege, priority, or family membership. When the returning process suspends itself, it - in effect - voluntarily surrenders control to the operating system. The operating system responds by inserting the process's process descriptors into the suspended list and by updating the process descriptor state field accordingly.

Suspending a suspended process usually has no effect, except in systems that keep track of the depth of suspension. In such systems, a process must be resumed at least as many times as it was suspended in order to become ready. To implement this feature, a suspend-count field has to be maintained in each process descriptor.

Typical error returns include process already suspended, wrong ID, and caller not authorised.

RESUME(process ID)

This call resumes the process, which is presumably suspended. Obviously, a suspended process cannot resume itself, because the suspended process depends on a partner process to issue the RESUME. The operating system responds by inserting the target process's process descriptors into the ready list, with the state updated. In systems that keep track of the depth of suspension, the operating system first increments the suspend count and moves the process descriptor only when the count reaches zero.

The SUSPEND/RESUME mechanism is convenient for relatively primitive and unstructured forms of interprocess synchronisation. It is often used in systems that do not support exchange of signals.

Error returns include process already active, wrong ID, and caller not authorised.

DELAY(process ID, time)

This call suspends a process for the duration of the specified time period. The time may be expressed in terms of a number of system clock ticks (system-dependent and not portable) or in standard time units (seconds, minutes). A process may delay itself or, optionally, delay some other process. The action of the operating system in handling this call depend on processing interrupts from the programmable interval timer.

The delay is very useful system call for implementing timeouts. In this application, a caller initiates an action and delays for the duration of the timeout. When the delay (timeout) expires, control is given back to the calling process, which tests the outcome of the initiated action. Two other varieties of time delay are cyclic rescheduling of a process at given time intervals and time-of-day scheduling, where process is run at the specific time of day.

The error returns include illegal time intervals or unit, wrong ID, and caller not authorised.

CHANGE_PRIORITY(process ID)

The priority of the process is set to a defined value. Considering this new priority the process is allocated processor. The run-time modification of a process's priority may be used to increase or decrease a process's ability to compete for system resources. The priority of a process should rise or fall according to the relative importance of its momentary activity and thus make scheduling more responsive to changes of the global system state. This may be violated by lower-priority processes, and the whole system may be corrupted by processes competing with the operating system itself. For these reasons, the authority to increase priority is usually restricted within a certain range.

The error returns include caller not authorised for the requested change and wrong ID.

GET_ATTRIBUTE(process ID, attributes)

The current status of a process with a given ID is read from its process descriptor. This is normally the only way for a process to find out what its current attributes are, because it neither knows where its process descriptor is nor it can access the protected operating system space where the process descriptors are usually allocated. This call may be used to monitor the status of a process, its resource usage and accounting information.

The error returns include no such attribute, wrong ID, and caller not authorised.

GET_ID

The attribute could be the ID of the process currently executed by the processor. It may be helpful in reentrant code to find out which process is currently running.

3.2. Scheduling

The design of the real-time systems enables the software designer to separate considerations of process scheduling from those of process design. The processes think that they have sole use of their own processor and memory. In a one process-per-processor set, the processes can be thought of as running continuously subject only to synchronisation requirements; in this case the scheduling problem does not arise. However in both multiprocessor and multitasking environment allocation of CPU/memory must be organised within the system.
It was pointed out earlier that the kernel performs a number of functions. Probably the most important of them is process scheduling. In simple terms, the scheduler defines the state of processes within the system - running, ready, suspended or deleted. It also includes the dispatching function, i.e. loading a new process into the processor. Scheduling decisions are made at frequent intervals, these being based on a set of scheduling rules or strategy. For embedded systems, the primary objective is to ensure that all processes are completed within specific time frames. Processor throughput is much less important. Unfortunately, a multitasking design solution brings it with a time overhead. The result is that the amount of time available for executing the application is reduced. Therefore, there is a need for a scheduling strategy which gives efficient processor utilization with minimum overhead.

Scheduling Problem

The purpose of process scheduling is to organize the set of processes ready for execution by the processor system (more precisely, to organize them so that performance objectives are met). Thus it is essentially an optimization problem. The order or arrangement of these processes is called a 'schedule'. For real-time embedded systems, our primary criterion is to ensure that all processes meet their deadlines. Scheduling has to answer the following questions :

When will be the CPU/memory scheduled ?

How the CPU/memory will be re-scheduled ?

Which new process will be chosen ?

Most commercial computer operating systems attempt to maximise the use of computing resources (disc drives, tape units. etc.). Accordingly, the priority of a certain process at a particular time may be strongly influenced by what resources it will use. Clearly, it is desirable to run a process that uses a disc unit at a time when the disc unit would otherwise be idle. However, once the system is expected to operate in real-time, the prime consideration must become that of adequate response to its environment. What is regarded as an adequate response, will depend on the application. Rea1-time considerations must take precedence over the economic use of system resources. The primary aim must be to design a solution to the scheduling problem that will ensure an adequate real-time response from the systems.

A schedule can be either feasible or optimal:

- *feasible schedule* orders processes so that they all meet their deadlines;
- *optimal schedule* is one which ensures that failures to meet process deadlines (i.e. late results) are minimized.

One of the most common ways of ordering processes in software is to connect them as a linked list. Their relative importance determines their position within the list, that at the top having highest priority. When a context switch takes place, this one is loaded (dispatched) into the processor system. The scheduling problem is to decide exactly where processes should be placed within the ready list. These decisions are made using a scheduling algorithm or policy which takes into account various process characteristics (attributes).

Process characteristics and scheduling

In real-time systems two characteristics of processes are considered to be of primary interest: criticality or importance; and timing. However, overall performance is also significantly affected by a number of other factors, including:

- communication and synchronization of processes
- mutual exclusion requirements, and
- fault-tolerance.

Process importance is frequently a subjective issue, whereas timing is objective (at least, the values are objective even if they have been arrived at by subjective reasoning). These points apply to both periodic and aperiodic processes. A periodic (synchronous) process is one which is activated at fixed, regular time periods. Its period is the time interval between two consecutive activations. Typical of periodic processes are closed-loop control, data monitoring and display updating. Aperiodic processes are those which are activated (made ready) by some particular event, and are a one-shot operation. Operations such as alarm detection and serial data handling are representative of these. If the events are generated outside the software system, then they must be implemented as interrupts.

The essential timing attributes of processes that define the scheduling procedure are as follows:

- worst-case computation time T_{Cw},
- deadline T_D.

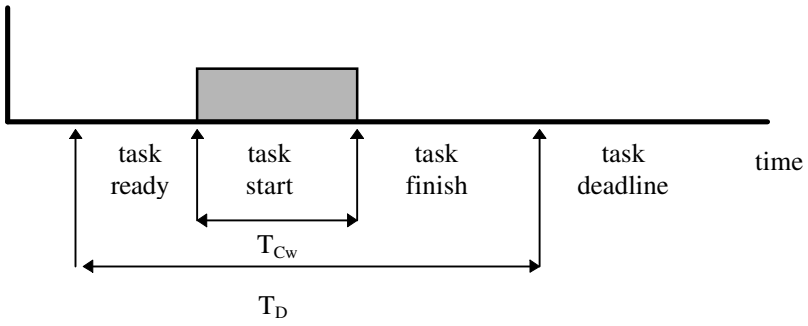

Figure 3.2.1. Process basic time attributes

The timing attributes of periodic processes also include process period time T_P.

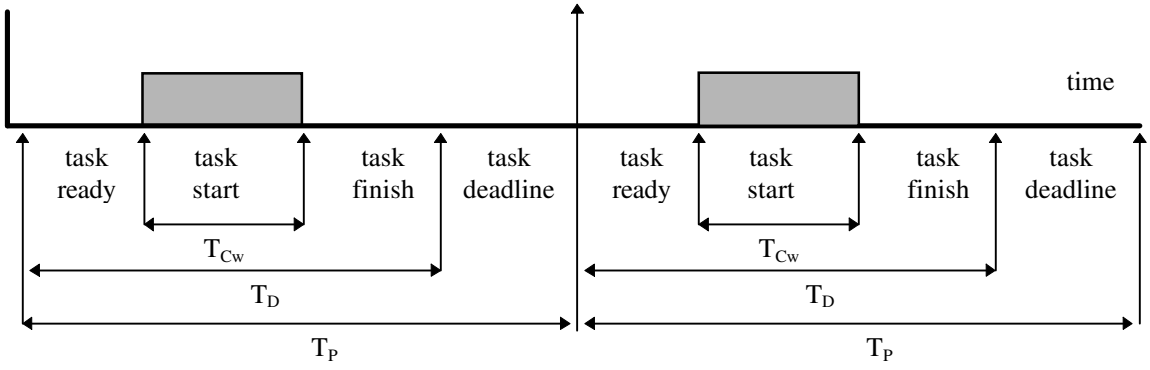

Figure 3.2.2. Periodic process - time attributes

Many real-time application processes have a 'hard' execution deadline that must be met, otherwise the real-time systems can fail with possible severe consequences. Unfortunately, criticality and time may often conflict. Therefore, in practice, an approximate analytic or simulation model should be used to resolve the issue. To guarantee that processes will meet their deadlines, most RTEX scheduling policies use worst-case computation times. Timing tools have been developed for predicting the deterministic bounds of computation times for processes. They are based on timing schema that are specified at the application source level.

A feasible process schedule is easier to attain with periodic rather than aperiodic processes. Unfortunately, scheduling optimization is particularly difficult to achieve in the presence of a mix of periodic and aperiodic processes. Even more unfortunately, this is the normal situation in hard embedded systems. Wherever possible, it makes sense to turn aperiodic processes into periodic processes. This can be done by replacing interrupt-driven operations with signal polling. Clearly, there is a limit to what can be done because of the polling overhead, but this is a matter of individual system design.

It is not sufficient to consider only the process timing attributes; we also have to allow for process interaction. Thus communication and synchronization constraints must be taken into account owing to precedence and exclusion relations between the application processes. Precedence relations exist between processes when some processes require information produced by others. Exclusion relations exist when some of them exclude others to prevent errors caused by simultaneous access to shared resources.

Computational complexity of the process scheduling problem

Ideally, reliable scheduling strategies used by RTEXs should incorporate all possible process constraints. The problem is that such strategies cannot be implemented in polynomial time, because a problem having n parameters is solvable in polynomial time if, as n is increased, the computation time increases polynomially. For a single processor this time increases exponentially, i.e. it is an NP complete problem in the strong sense. For real-time systems, the time overhead of a software-based executive must be minimized. Therefore, most practical RTEXs rely on scheduling strategies which depend only upon some of the process constraints. Application programmers frequently use ad hoc rules when applying scheduling strategies. The result is that timing (deadline) performance cannot, in practice. be guarantied. One potential solution to the scheduling time overhead is to remove all such functions from the process processor, and install them on a co-processor. The resulting processor system consists of a process processor(s) unit and a kernel co-processor unit. This co-processor takes over all kernel functions. Using this approach, the kernel time overhead can be reduced for two reasons:

- process processors are dedicated only to the execution of the application processes, and
- the co-processor can use hardware-based technique to perform process management.

Time overheads and scheduling

All scheduling decisions are taken at periodic intervals, called the 'tick' time. At such times, the application process is suspended and the scheduler is run. Its function is to decide which process should next be executed by the processor. In Figure 3.2.3. process A is running when the first tick (tick 1) occurs. However, the decision of the scheduler is that this process should be allowed to continue. Hence it is re-installed on the processor after the scheduler process. In contrast, after the second tick (tick 2), process A is left in a suspended state and process B becomes the running process. From this description, it is clear that the scheduler process reduces the amount of time available for processing of the application. Therefore we must strive to:

- minimize scheduler execution time, and
- minimize the tick frequency.

These two points are closely interlinked with scheduling policies (strategies).

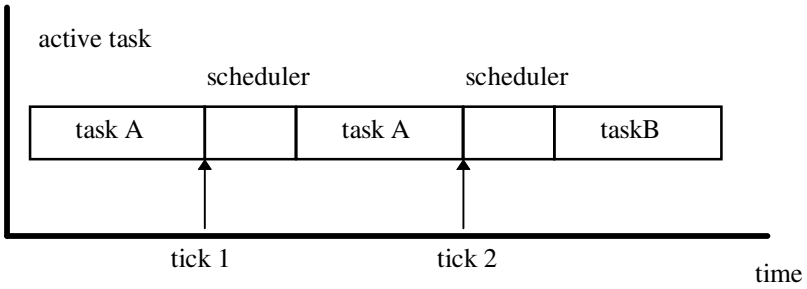

Figure 3.2.3. Process switching

This diagram, unfortunately, is somewhat simplistic. To have a proper understanding of the time overhead, it is necessary to examine operations in detail. Specifically, it is required to examine in Figure 3.2.4 what occurs at the tick point and on completion of the scheduling process.

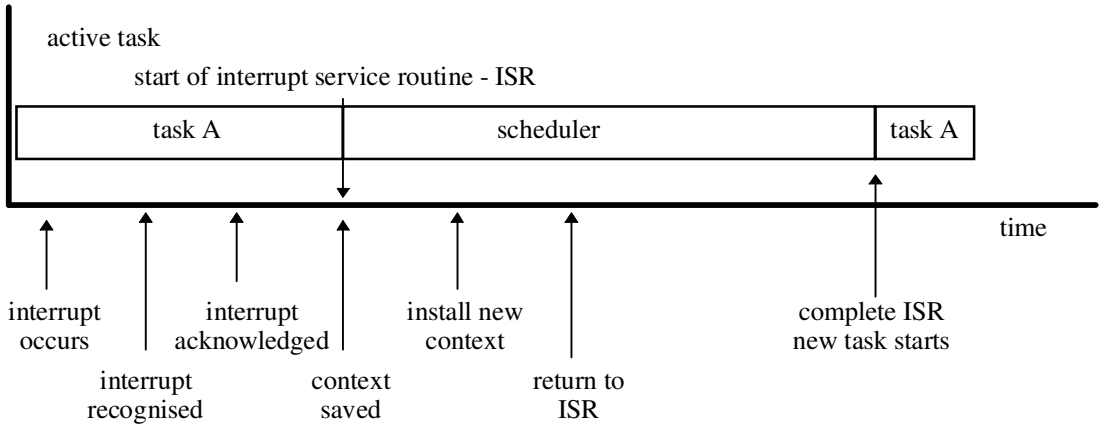

Figure 3.2.4. Context switch - detailed activities

The tick is normally produced by a hardware timer which generates an interrupt to the processor. Thus the first event is the tick interrupt line going active - the arrival of the interrupt. This may be followed by a time delay before the processor recognizes that an interrupt is present. Normally, any such delays are due to the interrupt being disabled (which can lead to very unpredictable timings).

After recognizing the interrupt, the processor completes the current instruction, saves essential register information, acknowledges the interrupt, and vectors to a specific address (precise details vary with CPU type). This is the start of the interrupt service routine, and the time taken to reach this point is defined as the 'interrupt latency'. Following this the 'context' of the running process is saved. The context is the collection of items which defines the current state of a process. Typically, it includes the values of:

- CPU registers
- maths co-processor registers
- memory management registers, and
- dynamic variables and pointers to dynamic data structures.

Once these are saved, scheduling decisions are made, and the process which is next to execute is identified. Its context is then loaded into the processor system, and a 'return from interrupt' is performed. The total time taken for all these actions is called the 'context switch time'.

It can be seen from Figures 3.2.3. and 3.2.4. that context switching has two distinct components: manipulating data (a simplistic but useful) term), and calculating process schedules. Let us consider each of these in turn.

The time taken to manipulate data is very processor specific, and is highly related to clock speeds. Further, it also depends upon exactly how much information is moved around the system. Therefore,

interrupt latency and context switch times are meaningless unless the platform they apply to is also specific. The time depends both upon the scheduling policy being used and the efficiency of its implementation. Although the numbers vary, they are all in the tens of microseconds range where context switching is done in software. Thus, the overhead of the RTEX is (sensibly) only likely to be a problem in very fast systems, moderately fast systems which have a large number of process, or in systems having complex scheduling algorithms.

The scheduling strategy in real-time system must also ensure that the system adjusts itself to an overload situation in a correct manner. As an example, take a message switching system that receives and stores messages. Once a message is received in its entity by the message switch, it is then re-transmitted down a selected path. Ideally, the message switch should never refuse incoming messages. Therefore, in an overload situation, the process reading in and storing messages must gain a high priority compared with the path selection and transmitting processes. This dynamic adjustment to overload is a necessity in all real-time systems in cases of heavy overload, involve the system in temporary abandonment of some of its less vital activities. It is necessary for the designer to know what is the desired acceptable or tolerable behaviour of the system under overload conditions. The formulation of suitable dynamic priority allocation algorithms is his most powerful in implementing these overload characteristics.

3.2.1. Priority structures

Priority Selecting a process to be run is a function of the relative importance of all eligible processes. Obviously the process selected should be that which is most important or vital for the system's operation. The scheduling problem involves deciding which is the most important process. It is convenient to think of process importance in terms of priority - a more important process having a higher priority than a less important one. Priorities are not necessarily fixed. The more flexible the allocation of priorities in a system, the more flexible will be the system's response to changing environmental conditions.

In a real-time system the designer has to assign priorities to the processes in the system. The priority will depend on how quickly a process will have to respond to a particular event. An event may be some activity of the process or may be the elapsing of a specified amount of time. Most RTOSs provide facilities such that processes can be divided into three levels of priority. They are as follows:

Real-time Priority Levels. A process's priority is dependent on how important it is for the process to run immediately upon the occurrence of a particular event. This event may be something occurring in the real-time system, or simply the passing of a certain amount of time. The more stringent the deadline the process has to meet, the higher its priority. Three levels of priority can be identified.

1. *Interrupt level:* at this level are the service routines for the processes and devices which require very fast response - measured in milliseconds. It holds the service routine for those devices and situations that require instant or near-instant service. One of these processes will be the real-time clock process and clock level dispatcher.
2. *Clock level:* at this level are the processes which require repetitive processing, such as the sampling and control processes and processes which require accurate timing. The lowest-priority process at this level is the base level scheduler.
3. *Base level:* processes at this level are of low priority and either have no deadlines to meet or are allowed a wide margin of error in their timing. Processes at this level may be allocated priorities or may all run at a single priority level - that of the base level scheduler.

Figure 3.2.5 shows a process priority level against time. In this idealised diagram, it is assumed that each process has a unique priority and that the dispatcher takes zero time to do its job. The diagram represents the following sequence. Initially, a base level process is running when interrupt occurs. The clock interrupt handling process takes over and carries out any necessary processing before

handing over to the highest-priority clock level process that is currently runnable. When this process finishes, a lower-priority clock level process takes over until it, in its turn, completes. The highest-priority base level process is then allocated the processor until it temporarily loses use of the processor to an interrupt handling process. In due course, the next clock interrupt occurs and the cycle is repeated.

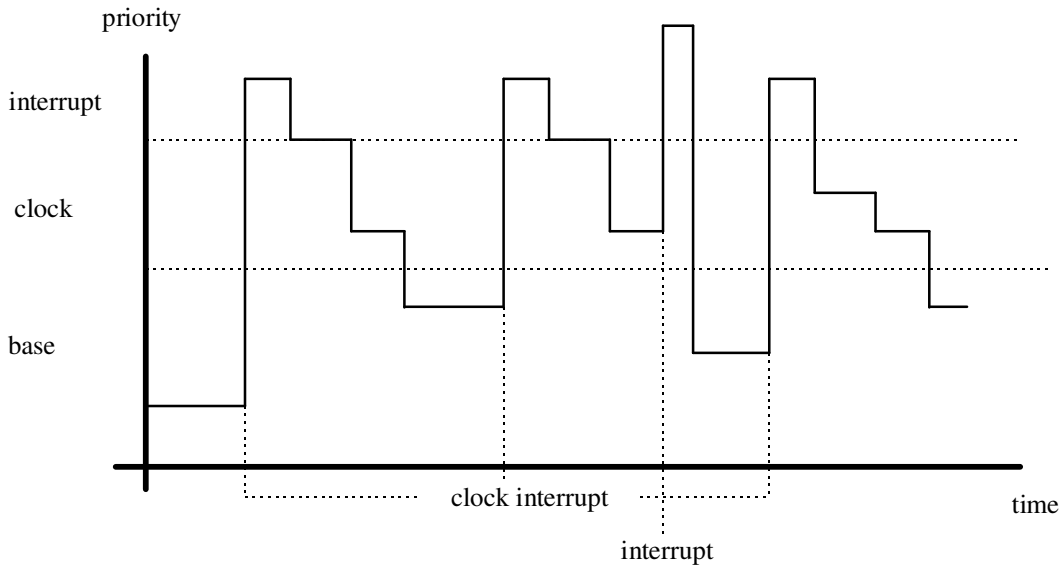

Figure 3.2.5. Priority versus time

3.2.1.1. Interrupt Level

It is at this level that the division between when to re-schedule and which to re-schedule becomes somewhat blurred. An interrupt forces a re-scheduling of the CPU to the corresponding interrupt-handling process. The system has no control over the timing of this re-scheduling. For this reason the amount of processing carried out at this level must be kept to a minimum. Normally, an interrupt level process will only do sufficient processing to avoid loss of data and status information and will then forward the resulting information to a lower priority process for further servicing.

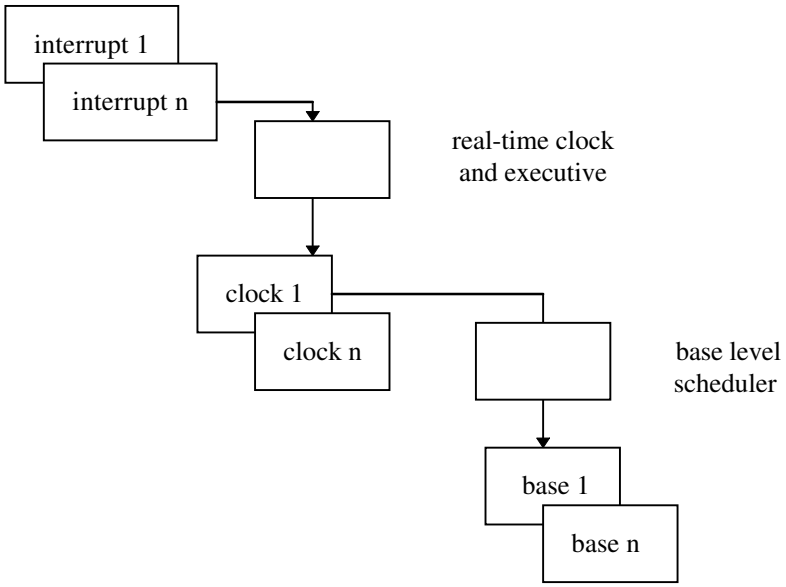

Fig. 3.2.6. Priority levels in real-time systems

The interrupt forces a re-scheduling, and therefore the interrupt level processes must conduct a form of process swapping. The interrupt processes are designed to store whatever parts of the interrupted process's volatile environment that may be required to users (normally a number of registers). The they carry out the necessary processing, replace the volatile environment and exit.

Even within the interrupt level, a priority structure will be necessary. The priority levels within the interrupt processes are defined by the relative time-critical nature of the interrupting devices. High-speed devices - for example, disc units - will need the ability to pre-empt the interrupt level processes associated with a slower device. The priority scheduling will normally be imposed by the hardware priority mechanism available on most modern processors.

In general, these mechanisms work as follows. The hardware includes a number of interrupt lines, each associated with an interrupt priority level. The CPU includes a register in which the current priority is stored (the processor priority). Associated with each interrupt line is a pair of memory locations (interrupt vector). The first of these locations holds the address of the start of the interrupt handling process for that interrupt line/priority level. When a device wishes to interrupt, it puts a signal on an interrupt line. If the processor priority is currently greater than the priority of the interrupted line, the interrupt is ignored. If not then the current program counter is stored in the second location of the interrupt vector, the processor priority becomes that of the interrupting line and the value in the first location of the interrupt vector is placed in the program counter. When the interrupt handling process completes, it lowers the processor priority and replaces the program counter with the value previously stored in the second location of the interrupt vector. Note that interrupt handling processes may be interrupted by higher-priority interrupts.

The return from interrupt to the interrupted process is implied by the RTEX, providing a stacking mechanism for several levels of interrupt activity. However, a return could also be forced via the dispatcher which may consequentially wish to re-schedule the processes after doing some essential housekeeping.

Interrupts force a rescheduling of the work of the CPU and the system has no control over the timing of the rescheduling. Because an interrupt-generated rescheduling is outside the control of the system it is necessary to keep the amount of processing to be done by the interrupt handler to a minimum. Usually the interrupt handler does sufficient processing to preserve the necessary information and to pass the information to a further routine which operates at lower- priority level, either clock level or base level. Interrupt handlers have to provide a mechanism for process swapping, that is they have to save the volatile environment. After completing the interrupt handler it either will simply restore the volatile environment and hence will return to the interrupted process or it may exit the scheduler.

Within the interrupt level there will be processes with different priorities. As a result, there will have to be mechanism for preventing interrupting processes with higher level interrupts by processes with lower level interrupts.

3.2.1.2. Clock level

The real-time clock handler, which is an interrupt level process, will be run at some time interval, usually determined by timing requirements of application processes. Typical values are one to two hundreds ms. Each clock interrupt is called as a tick and represents the smallest time interval known to the system. The function of the clock interrupt handler is to update the time-of-day clock in the system and the transfer control to the dispatcher.

The Real-time Clock. One of the interrupt-level processes is the real-time clock handler. A real-time clock is a hardware device which interrupts the processor at a regular rate. Virtually all real-time systems include such a device. The reason for its inclusion is fourfold.

- *First,* not all activities carried out by a real-time system are a response to some external stimulus or interrupt. Very often, physical constraints (for example, lack of available interrupt lines) will prevent the device or mechanism that is being controlled from interrupting the

processor. Therefore, in order to ascertain the need for servicing, it becomes necessary for the real-time system to inspect the status of the device at regular intervals. This technique, called polling, is used widely from communication systems to control systems. The interval between servicing will, of course be dependent on the performance characteristics of the device. Ideally, system control should be centred on the CPU. Interrupting devices command the attention of the CPU and are therefore exercising a degree of control. By regularly scanning the status of the devices. the CPU avoids the need for interrupt and retains full control of the system. It is necessary, then, for the system to be aware of the passing of real-time so that it may activate device-servicing routines at relevant times.

- *Second*, many devices controlled by the system will operate extremely slowly compared with the speed of the CPU. This it especially true of electromechanical devices such as motors and relays. It will therefore be necessary for the process to be able to wait a certain amount of time before continuing.
- *Third.* it will often be necessary for the system to have a knowledge of time in terms of a 24-hour day. In this way exact timing of certain events can be handled, i.e. displayed, stored and recalled.
- *Fourth*, the real-time clock enables time-deendent scheduling - for example, periodic polling or sampling.

The real-time clock interrupting at regular time intervals provides the system with a knowledge of, and thus the ability to measure the passing of time. The rate at which a clock interrupts will depend upon the timing accuracy required by the system. It will usually be defined by the required rate of activation of the most often activated device handler. Typical values would be 20, 50 or l00 ms.

Clock Level Processes. Clock priority level processes are those which must run at strictly regular intervals. The real-time clock interrupt handling process, when activated will normally update the time-of-day counter (some memory location) and transfer control to the dispatcher, having first left an indication that a real-time clock interrupt has just occurred. It is then up to dispatching mechanism to select one of the clock level processes that must run at that clock `tick'.

Some of the clock level processes will require accurate timing for their activation (the very reason that they are at clock level). If the processes requiring the most accurate timing are always run first after a real-time clock interrupt, then they will be initiated at a regular rate. Those scheduled later will have some `'jitters' caused by the varying running times of the previously run processes. Here, the priority of the process is decided by its sensitivity to variations in timing.

It is possible, in certain circumstances, for processes running at clock level to take longer than the period between clock interrupts. Rather than let these momentary overloads disrupt the other processes that require high timing accuracy, it is possible to split the processes into two priority classes. The high-priority processes that guarantied to finish within the clock interrupt period run first. Thereafter, the lower-priority clock level processes run at the risk of being interrupted by the real-time clock on its next tick.

The scheduler selects which process is to run at a particular clock tick. Clock level processes at this level are divided into two categories: cyclic, delay

Cyclic processes. These are processes which require accurate synchronisation with the outside world. Their priority reflects the accuracy of timing required for the process. Those processes which require high accuracy are being given the highest priority. Processes of lower priority within the clock level

Example 3.2.1.2 There are three processes A, B and C required to run at 20 ms, 40 ms and 80 ms intervals corresponding to 1 tick, 2 ticks and 4 ticks, if the clock interrupt rate is set at 20 ms. If the process priority order is set as A, B and C with A as the highest priority then the processing will be shown in Figure 3.2.1.3. Using a single CPU it is not possible to have all the processes starting at the

clock tick. All but one of the processes will be delayed relative to the clock tick; however, the interval between successive invocations of the process will be constant (if the execution time for each process is a constant value). If the priority order is now rearranged so that it is C, A and B then the activation diagram is as shown in Figure xx and every fourth tick of the clock there will be a delay in the timing of processes A and B. In practice there is unlikely to be any justification for choosing a priority order C, A and B rather than A, B and C. Usually, the process with the highest repetition rate will have the most stringent timing requirements and hence will be assigned the highest priority.

A further problem which can arise is that a clock level process may require a longer time than the interval between clock interrupts to complete its processing (note that for overall satisfactory operation of the system such a process cannot run at a high repetition rate).

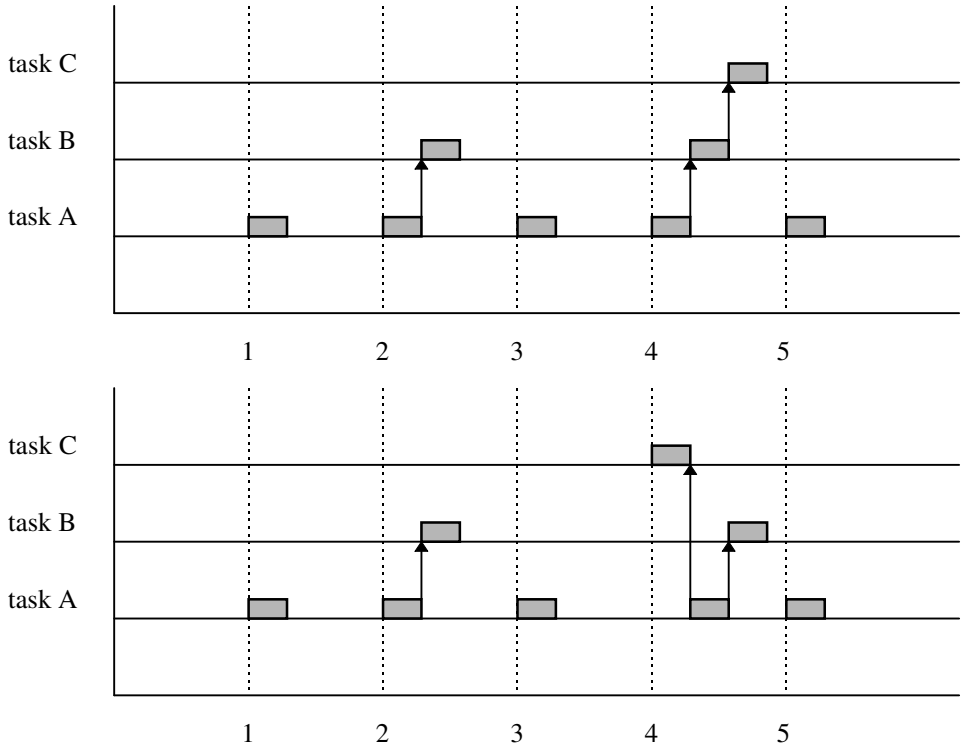

Figure 3.2.7. Timing diagram of cyclic processes

Example 3.2.1.2. Assume that in Example 3.2.1.1. process C takes 25 ms to complete, process A takes 1 ms and process B takes 6 ms. Assumed that process C is allowed to run to completion. Process A will be delayed by 11 ms at every fourth invocation. It is normal therefore to divide the cyclic process into high-priority processes which are guarantied to complete within the clock interval and lower-priority processes which can be interrupted by the next clock tick.

Delayed processes. There is, however, another class of processes that use the real-time clock interrupt. These processes wish to delay their activities for a fixed amount of time. Priority of these processes is within the base level.

These processes either delay their activities for a given period of time or there must be a fixed delay between their successive execution. The processes which wish to delay their activities for a fixed period of time, either to allow some externa1 event to complete, for example, a relay may take 20 ms to close or because they only need to run at certain intervals, for example, to update the operator display, usually run at the base level. When a process requests a delay its status is changed from runnable to suspended and remains suspended until the delay period has elapsed.

One method of implementing the delay function is to use a queue of process whose function is to handle processes DELAYED. This queue is an ordered list of process descriptors, the process at the front of the queue being that whose next running time is nearest to the current time. When a process

delays itself it calls an executive process which calculates the time when the process is next due to run and inserts the process descriptor in the appropriate place in the queue.

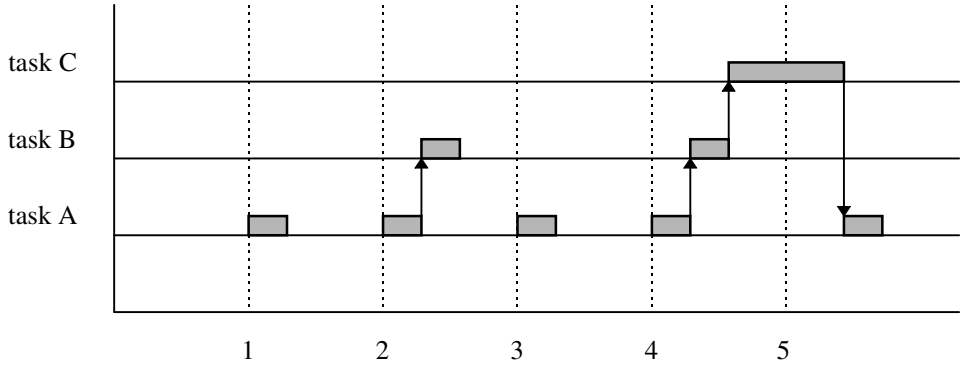

Figure 3.2.8. Timing diagram of delayed processes

A process running at the clock level checks the first process in the DELAYED queue to see if it is time for that process to run. If the process is due to run it is removed from the DELAYED queue and placed in the RUNNABLE queue. The process which checks the DELAYED queue may be either the dispatcher which is entered every time the real-time clock interrupts or another clock level process which runs at less frequent intervals, say every 10 ticks, in which case it is part of the base level scheduler. Many real-time operating systems do not support the cycle operation and the user has to create an accurate repetitive timing for the process by using the delay function.

A common method of implementation this delay capability is to introduce a `clock queue' of process descriptors. When a process wishes to be delayed for a certain length of time, it calls upon an executive subroutine to remove its process descriptor from the runnable list and place it on the clock queue. The clock queue is a list of process descriptors of processes that have requested suspension. The process descriptor is placed in the list in a position relative to the time at which re-activation is required. When the real-time clock interrupts, the interrupt handling process updates the time-of day counter. It then checks to see if this time corresponds to the time of the re-activation of the first process(es) in the clock list. If so, the process is made runnable by transferring the process descriptor to the queue of runnable processes.

This repetitive checking of the clock list may be considered an unacceptable overhead for the real-time clock handler. After all, the interrupt handlers are meant to be short! To reduce this overhead, a special clock level process can be constructed. This process designed to be activated every so many clock ticks, does the checking and process descriptor swapping. The period between successive activations of this process will define the minimum integral delay interval possible in the system.

3.2.1.3. Base Level

Once the processing requirements of clock level processes have been met, the remaining time in a clock period is available for use by the base level processes. The processes at the base level are initiated on demand rather than at some predetermined time interval. The demand may be user input from a terminal, some process event or some particular requirement of the data being processed. The way in which the processes at the base level are scheduled can vary; one simple way is to use time slicing on a round-robin basis. In this method each process in the runnable queue is selected in turn and allowed to run until either it suspends or the. base level scheduler is again entered. For real-time work in which there is usually some element of priority this is not a particularly satisfactory solution. It would not be sensible to hold up a process, which had been delayed waiting for a relay to close but was now ready to run, in order to let the logging process run.

Priority Allocation among Base Level Processes

The priority assigned to a process may be fixed for the life of the system affecting the system at a particular time.

Fixed Priority Process priority can be fixed and assigned at the time the process is created. Its priority would then be a function of the urgency of the job it must perform and often of the cos' or availability of the system resources that it uses.

Unfortunately, it is very difficult for a designer accurately to forecast what will be a satisfactory mix of priorities for base level processes. This problem is aggravated by the dynamic and unpredictable nature of the system's environment. Knowledge and experience can be obtained by examining similar systems, but if this previous experience is not available then other methods must be used to gain information on system behaviour. This can be done either by analysing simulation models of the system, or by monitoring actual system behaviour under different priority mixes and environmental conditions.

Variable Priority - high-level scheduler - The problem of forecasting can be avoided if process priorities are varied dynamically, dependent on the behaviour of the system when it is in operation. The high-level scheduler can be pressed into service to fulfil this function.

The high-level scheduler is an executive process that has the ability to alter process priority. It is activated at regular intervals and uses its knowledge of overall system status to re-allocate process priority. The calculation of new process priorities is an overhead on the system's workload, and so the algorithm that it uses should be as simple as possible. An example of a useful algorithm would be to give a higher priority than normal to a process whose input channel was growing excessively large. For commercial, batch-based operating systems, the algorithm chosen would be one that would tend to maximise the use of system resources or throughput of user jobs. For a real-time system however the algorithm must ensure that the time-critical processes are carried out. Therefore, when allocating a priority to one process, it must take into consideration process deadlines. More complex priority allocation algorithms will take into consideration the deadlines of all the processes in the system.

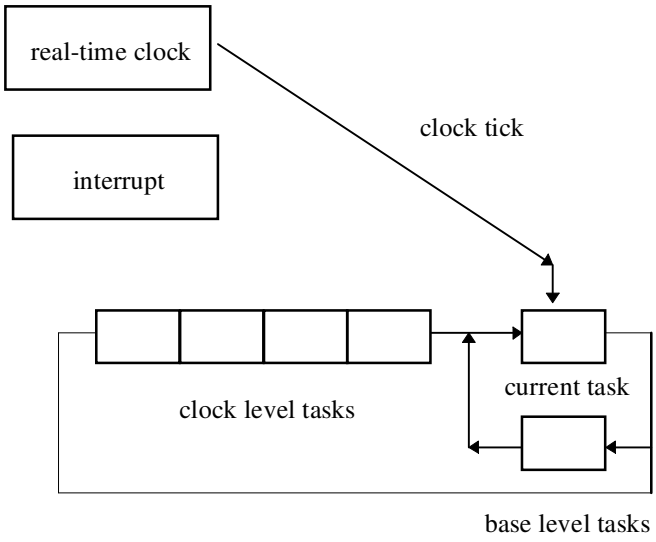

Figure 3.2.9. Scheduling scheme

Most real-time systems use a priority strategy even for the base level processes. This may be either a fixed level of priority or a variable level. The difficulty with a fixed level of priority is in determining the correct priorities for satisfactory operation. The ability to change priorities dynamically allows the system to adapt to particular circumstances. Dynamic allocation of priorities can be carried out using a high-level scheduler or can be done on an ad hoc basis from within specific processes. The high-level scheduler is an operating system process which is enable to examine the use of the system resources. It may for example check how long processes have been waiting and increase the priority of the processes which have been waiting a long time. The difficulty with the high-level scheduler is that the algorithms used can become complicated and hence the overhead of running can become significant.

Alternatively priorities can be adjusted in response to particular events or under the control of the operator. For example, alarm processes will usually have a high priority and during an alarm condition processes such as the log of plant data may be delayed with the consequence that the output of the log lags behind real. So that the log can catch up with real time quickly it may be advisable to increase, temporarily, the priority of the printer output process.

3.2.1.4. Types of schedulers

There are three types of scheduler

- the long-term scheduler
- the medium-term scheduler
- the short-term scheduler

The long-term scheduler selects the next resource intensive, for example processor time, memory, I/O devices, low-priority programs that may be used to keep the system resources busy during periods of low activity. These programs contain all the necessary data and commands for their execution. They may also contain programmer-assigned or system-assigned estimates of their resource needs. The primary objective of the long-term scheduler is to provide a balanced load of low-priority programs, such as processor-bound and I/O-bound to the short-term scheduler. When the processor utilisation is low, the scheduler may admit more processes to increase the number of processes in a READY queue and with it the probability of having some useful work awaiting processor allocation. The invocation of the long-term scheduler is both system and workload-dependent, but it generally much lower than that of the other two types of schedulers.

The medium-term scheduler Being executed a running process may be suspended making an I/O request or issuing a system call. The suspended process cannot make any progress until the related suspension condition is removed. It is sometimes worth to remove such processes from the main memory in order to make space for other processes. Saving the suspended process on back storage is called swapping. The medium-term scheduler performs the swapping of processes. Once the condition resulted in suspension is removed, the medium-term scheduler attempts to allocate the required amount of memory and swap the process in and make it ready. In order to work properly, the medium-term scheduler must be provided with information about the memory requirement of the corresponding process. This is known because the actual size of the process may be recorded at the of swapping and stored in the related process's descriptors.
The medium-term scheduler control the suspended-to-ready transition of the swapped processes. It may be invoked when memory space is available after removing a process or when the number of ready processes falls below a specified limits.

The short-term scheduler It allocates the processor among the ready processes that are in the memory, i.e. they are resident. Its main objective is to maximise system performance in accordance with the chosen set of criteria. It performs a ready-to-runnng state transitions. It is invoked as often as once for each process switch to select the next process to be run. The short-term scheduler is invoked whenever an event changes the global state of the system. Given that any such change could result in suspension of the running process or in suspension of one or more ready processes, the short-term scheduler should be run to determine whether such significant changes have indeed occurred and if so, to select the next process to be run. This rescheduling may be invoked by the following events:

- clock ticks
- interrupts and completing I/O operations
- some system calls
- sending and receiving signals.

Whenever one of these events occurs, the operating system invokes the short-term scheduler to determine whether another process should be scheduled for execution.
Large systems may use all three types of schedulers. Small, medium-sized or special-purpose systems may have only one or two of them. A long-term scheduler is normally not found in systems without support for resource intensive low-priority processes. The medium-term scheduler is used only when swapping is applied by the operating system. When more than one type of scheduler exists in an operating system, proper support for their interaction is very important.

3.2.2. Scheduling Strategies

Figure 3.2.2.1 gives a comprehensive view of the scheduling policies of interest to the real-time community. It can be seen that there are two major groupings:

- simple and
- priority-based.

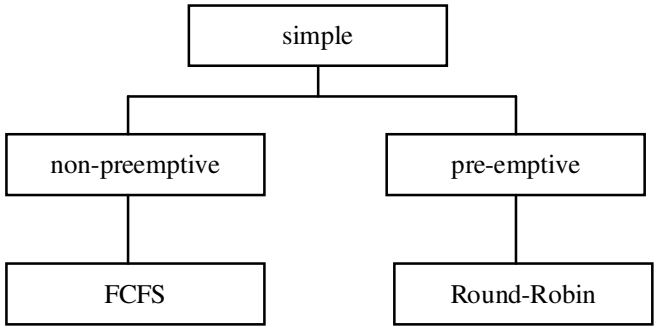

Figure 3.2.10. Simple scheduling algorithms

Simple scheduling strategies are those where all processes are considered to have equal standing. Non-preemptive techniques are those which always allow executing processes to run to completion. By contrast, when using pre-emptive policies, the running process can be swapped out at any time the scheduler decides to do so.

In **priority-based scheduling strategies**, processes are assigned priority values based on one or more specific process attributes. The position of processes in the ready queue is defined by their priority number. Processes may, of course, be allocated equal priorities. Where this happens, the sub-group so formed is scheduled using one of the simple strategies.

The majority of existing RTEXs use a priority scheduling mechanism. Processes are allocated a priority level and at the end of a predetermined time slice the process with the highest priority of those ready to run is chosen and is given control of the CPU. This may mean that the process which is currently running may continue to run.

Process priorities may be fixed - a static priority system - or may be changed during system execution - a dynamic priority system. Dynamic priority schemes can increase the flexibility of the system, for example they can be used to increase the priority of particular processes under alarm conditions. Changing priorities is, however, risky as it makes it much harder to predict the behaviour of the system and to test it. There is the risk of locking out certain processes for long periods of time. If the software well designed and there is adequate computing power there should be no need to change priorities - all the necessary constraints will be met. If it is badly designed and/or there are inadequate computing resources then dynamic allocation of priorities will not produce a. reliable system.

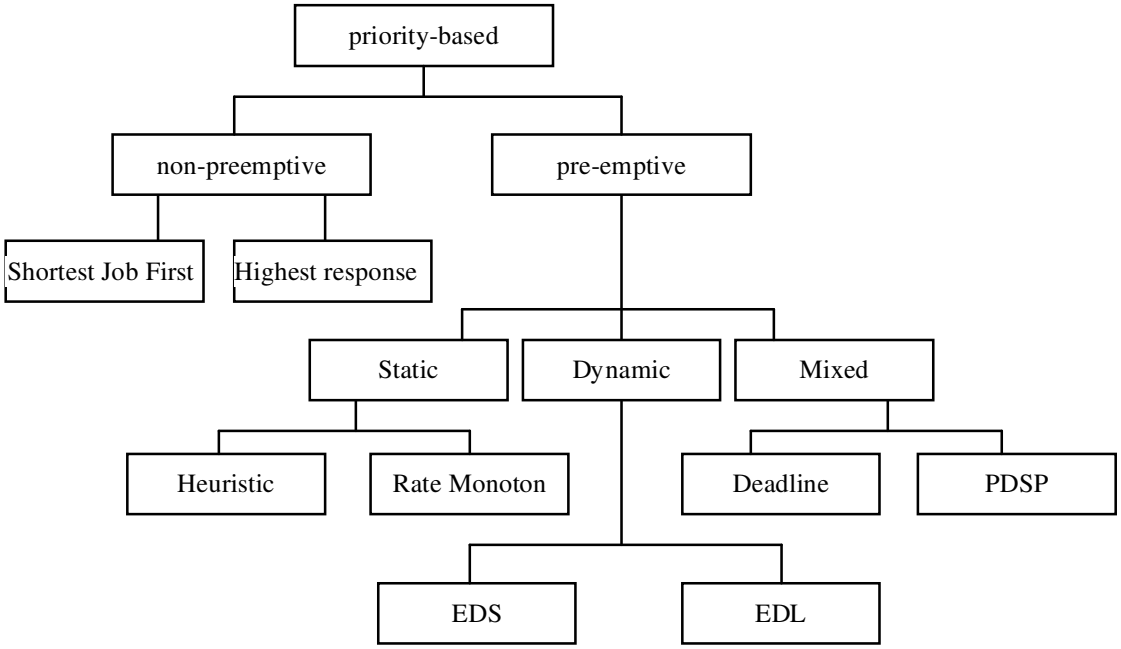

Figure 3.2.10. Priority-based scheduling algorithms

Whatever scheduling strategy is adopted the process management system has to deal with the handling of interrupts. These may be hardware interrupts caused by external events, or software interrupts generated by a running process. An interrupt forces a context switch. The running process is suspended and an interrupt handler is run. The interrupt handler should only contain a small amount of code and should execute very quickly. When the handler terminates either the process that was interrupted is restored or the scheduler is entered and it determines which process should run. The RTOS designer has to decide which approach to adopt.

Cyclic scheduling strategy allocates the CPU to a process in turn. The process uses the CPU for as long as it wishes. When it, no longer requires it the scheduler allocates it to the next process in the list. This is a very simple strategy which is highly efficient in that it minimises the time lost in switching between processes. It is an effective strategy for small embedded systems for which the execution times for each process run are carefully calculated, often by counting the number of machine instruction cycles for the process, and for which the software is carefully divided into appropriate process segments. In general this approach is too restrictive since it requires that the processes have similar execution times. It is also difficult to deal with random events using this method.

Pre-emptive scheduling strategy. In order that the currently running processes reflect the relative process priorities in effect at a particular time there is a need for a pre-emptitive scheduling strategy. A scheduling strategy is pre-emptitive if a process wanted to be activated, and it has a higher priority than the process currently running then the current process will be stopped so that the higher-priority process may proceed. For example; if a base level process is running, and a device produces an interrupt, the relevant interrupt level service process does not wait for the base level process to finish, but will be executed immediately before the base level process is resumed.

There are many pre-emptive strategies. All involve the possibility that a process will be interrupted - hence the term pre-emptive - before it has completed a particular invocation. A consequence of this is that the executive has to make provision to save the volatile environment for each process, since at some later time it will be allocated CPU time and wil1 want to continue from the exact point at which it was interrupted. This process is called context switching. The simplest form of pre-emptive scheduling is to use a time-slice approach, sometimes called a round-robin method. Using this strategy each process is allocated a fixed amount of CPU time - a specified number of ticks of the clock - and at the end of this time it is stopped and the next process in the. list is run. Thus each

process in turn is allocated an equal share of the CPU time. If a process completes before the end of its time slice the next process in the list is run. immediately.

Non-pre-emptive scheduling strategy. The alternative, a non-pre-emptive approach, would imply that the processes be they interrupt clock or base level, run to completion without interruption. Upon completion, each process would transfer control to the dispatching mechanism.

However, that some re-schedulin by the real-time executive can happen when external events cause interrupts and when kernel support is requested through system calls. A simple system is illustrated in Figure xx. The dispatcher runs the base level processes serially, up to a clock `tick' interrupt when the clock level processes are run sequentially, and then returns to the base level after completion of all clock level processes. Other interrupt level processes hardware or software may occur, provided that the service routines are not time-critical and are short and infrequent enough to guarantee successful completion all processes at clock and base 1evels Such a simple system implies a limited degree of re-emption but in a very controlled manner by selective enabling of the interrupt facility.

Significant advantages are obtained when using this latter technique. Since the processes are not interruptable, poor synchronisation does not give rise to the problem of corruption of shared data. Shared subroutines can be implemented without producing re-entrant code or implementing lock and unlock mechanisms. However, the main drawback with this approach is that while the current process is running the system is not responsive to changes in the environment. Therefore, system processes must be extremely brief if the real-time response system is not to be impaired.

3.2.3. Scheduling Algorithms

First-come-first-served (FCFS). This policy is the simplest scheduling strategy to be found, Thus it does not invoke any process constraints. Ready-to-run processes are organized in a list, that at the top being executed first. When a process becomes ready, it is added to the end of the ready list. Thus processes execute in the order in which they are readied - first come, first served. It is non-pre-emptive, each process being allowed to run to completion.

FCFS is, by itself, highly unsuitable for hard embedded systems for two reasons: first, its responsiveness can be quite poor - when a process is readied it has to join the back of the existing ready queue, which may lead to a long wait before it gets executed; second, it has very unpredictable timings we never know how long it is going to take before it is set running (although a worst-case estimate may be made). However, it is effective in some cases when used within other scheduling strategies. VRTX, for instance, is an example of a commercial RTEX that adopts this policy for scheduling subsets of processes having the same scheduling priority.

Time Slicing on a Round-Robin Basis. The policy is the pre-emptive version of the FCFS technique. Processes are still arranged in the ready list in the order in which they become ready-to-run, but when they are set running they execute within a fixed time slice. If a process is still running at the end of its time slice, it is forcibly removed from the processor. Its replacement is the process which was at the head of the ready queue - the pre-empted process is sent to the end of the ready queue. As the process set is executed, it gradually works its way to the top of the queue. When one more installed on the processor it resumes execution from the point of interruption.

A scheduler requires a time management function to implement this policy - real-time clock tick handler. The time slice is proportional to the period of the clock tick. As pointed out earlier, the time slice length is critical. It must not be too small, but unfortunately the consequence is frequent context switching.

This is relatively straightforward approach. At each clock period, one of the base level processes is selected to be run for the remainder of the period. This selection is done on a round-robin basis, a new process being selected each period. If the selected process, when activated, has no work to carry out, it returns control to the dispatcher immediately after it is activated. The dispatcher then choses

the next process in line. Similarly, if the chosen process terminates before the end of the clock period, it returns control to the dispatcher.

Of course this method takes no account of the relative priorities of the base level processes. An implicit priority scheme can be introduced by giving higher-priority processes extra clock periods when their turn comes round on the round-robin. An alternative method is to set up a round-robin where higher priority processes have more than one place on the round-robin and are therefore selected more often.

Re-scheduling the processes ever clock period may prove too high an overhead and so the dispatcher may re-schedule after every so man clock ticks. In order to implement the round-robin method it is only necessary to provide the dispatcher with a circular list of runnable processes, and include within the dispatcher sufficient intelligence to maintain the round-robin.

Accordingly to the basic philosophy that systems should always be as simple as possible, then the round-robin method provides the most applicable strategy. In many situations, a simple policy such as this will be sufficient. The designer should introduce more complex methods only when simple methods have been shown to be inadequate. Even if this is the case, simple methods should not necessarily be rejected. Considerable speed improvements in the processes themselves can often be obtained by the use of more efficient algorithms, simplifying processes by removal of unnecessary frills and constraints, and by judicious use of low-level code.

It should be slightly greater than the average worst-case process computation time. If it is much greater it results in excess processor idle time (low utilization). If less, processes require multiple time slices to complete, and thus response time suffers. Round-robin scheduling has been widely used in slow and/ or soft real-time systems. For fast, hard applications its responsiveness is generally unsatisfactory. However, like FCFS, it can be effective when used within other scheduling policies. VRTX, for example, also offers round-robin scheduling for subsets of equal priority processes. When time-slicing is enabled, this sub-set executes in round-robin fashion, the time duration being programmer-defined.

Shortest-job-first. This policy is a priority-based alternative to the FCFS scheduling strategy. It uses a single criterion in defining priority, process execution time. Within the ready list, the process with the shortest (worst-case) computation time is allocated the highest scheduling priority. As a result it is placed at the front of the ready queue, waiting for service by the processor. However, the policy is non-preemptive, and the current executing process is always allowed to complete. Observe that this is a static scheduling policy - priorities are fixed.

This strategy, compared to FCFS and round-robin policies, produces a better average response time. However, by itself it is unsuitable for use as a main (reliable) scheduling strategy of real-time schedulers. It can, though, be applied within some other (priority-based) policy.

Highest-Response-Ratio-Next. The strategy overcomes some of the weaknesses in the shortest-job-first policy. The schedule order of processes depends on two criteria: process execution time; and process waiting time (i.e. the amount of time a process has been readied but waiting to be executed).

Priority is calculated as follows:

$$P = \frac{T_W + T_C}{T_C} \qquad (3.2.1)$$

where P - priority,

T_w - time a process has been ready and waiting, and

T_C - process computation (execution) time.

The process having the highest priority is placed at the front of the ready queue. Note that this policy is also non-preemptive.

Heuristic Scheduling. This policy is often called ‘static priority scheduling’. It proceeds from the assumption that each process has associated a fixed (static) priority. This defines its importance for the application. The processes are connected in order of priority in the ready list, the highest being on the top. This is a pre-emptitive policy; thus at reschedule time the 'running process will be replaced (pre-empted) if a higher priority process is ready. Process importance is evaluated heuristically by application designers. They are fully responsible for the performance of the application; therefore the policy belongs to the class of 'subjective' scheduling approaches.

This policy is simple, easy to use and generally seems to be effective. As a result, it is provided by most commercial RTEXs (e.g. RMK, VRTX, VxWorks, Venix, etc.).

Rate Monotonic Scheduling (RMS). This policy considers a single process timing criterion, process period. Ready processes are arranged in order dependent or their period; that having the shortest period is placed at the front of the ready queue, being allocated highest priority, The next process is the one which has the second shortest period, and so on. Thus, as we go down the ready list, process periods always increase (a monotonic sequence). In this strategy, the most frequent processes execute first. A running process may be pre-empted if a process with a shorter period is already ready.

The rate monotonic strategy is based on certain assumptions about application processes:

- all processes are equally important
- a process computation time is constant and does not vary with time
- processes for which hard deadlines exist are periodic
- a process deadline is defined by the process period
- aperiodic processes have no hard deadlines
- processes have no precedence or exclusion relations
- processes may be pre-empted
- embedded system has one processor.

One of the claimed advantages of rate monotonic scheduling is the ability to achieve high processor utilization whilst still meeting deadlines. A schedule is said to fully utilize the processor if its assignment is feasible, but any increase in the run-time of any of the processes then makes it infeasible. Now, be careful in interpreting this statement; full utilization has a very specific definition here. Because a set of scheduled processes fully utilize the processor, this does not mean that the processor is working to full capacity (100% processor utilization).

A schedule of n ready to-run processes with the rate monotonic priority order, the least upper bound to processor utilization is :

$$U=n(2^{1/n}-1) \qquad (3.2.2)$$

The least upper bound of the utilization factor is the minimum of the utilization factors over all schedules that fully utilize the processor. A process set using rate monotonic scheduling will meet their deadlines if the following condition is satisfied:

$$\sum_{i=1}^{n} \frac{T_C(i)}{T_P(i)} \le U \qquad (3.2.3)$$

where $T_C(i)$ - the worst-case computation time of a process i

$T_P(i)$ - the period of a process i and

U - the worst-case utilization bound.

For the rate monotonic policy this is $n(2^{1/n}-1)$. It is a quantity that decreases monotonically from 0.83 when n = 2 to ln2 = 0.693 as n increases to infinity.

The major weaknesses of RMS are that:

- processes are considered to be periodic and independent
- aperiodic processes are effectively neglected, and
- process deadline and period are considered to be synonymous.

Earliest Deadline as Soon as Possible (EDS). In this scheduling strategy priority is defined using a single criterion, time to deadline (process deadline). A process will be assigned the highest scheduling priority if its current deadline is the shortest (nearest), and placed at the front of the ready queue. The process with the longest (furthest) deadline is placed at the end of the ready queue. It should be clear that deadline values change during program execution. Thus this scheduling method belongs to the class of dynamic policies. It has its origins in the deadline driven scheduling algorithm, and makes the same assumptions as RMS.

In the real-time literature, the deadline driven strategy is often called the earliest deadline as soon as possible (EDS) scheduling policy.

The necessary and sufficient condition for the feasibility of a set of n processes scheduled by using the deadline driven scheduling policies as follows:

$$\sum_{i=1}^{n} \frac{T_C(i)}{T_P(i)} \leq 1..00 \qquad (3.2.4)$$

where $T_C(i)$ - the worst-case computation time of a process i and

$T_P(i)$ - period.

A schedule of "n" processes is feasible if and only if condition (3.2.4) is fulfilled. The bound 1.000, used in this condition, is the worst-case utilization bound for the deadline driven policy.

Earliest Deadline as Late as Possible (EDL). The EDL scheduling principle uses the criterion of process 'laxity', i.e. spare time in Figure 3.2.3.1. A simple illustration is given here. Assume that two processes are readied at the same instant. It can be seen that, taking into account the time frames set by the deadlines, process 2 has less spare time than process 1. EDL assigns highest priority to the process having least spare time (least laxity). Thus in this example, process 2 takes priority over process 1.

EDL is an alternative to EDS. It also invokes the same assumptions as the rate monotonic principle, but it is a more complex algorithm. For EDL, once a process is readied, time to deadline reduces as time elapses. For EDS, laxity also changes in the same way. However, laxity is also affected when process computation is carried out. The general expression defining the laxity of a process i is:

$$laxity(i, t) = \max(T_D(i) - T_C(i) - t, 0.000) \qquad (3.2.5)$$

where $T_D(i)$ - deadline,

$T_C(i)$ - computation time and

t - the current run-time.

Note that the second max function parameter is fixed to 0.000 because a laxity value must never be negative. It is obvious that laxity is a dynamic attribute; thus this is a dynamic scheduling policy.

Deadline scheduling and the priority ceiling protocol. The assumptions made by the rate monotonic and earliest deadline policies that:

- aperiodic processes can be neglected when making scheduling decisions

- all processes are independent, i.e. they do not have precedence or exclusion relations, and that

- no account is taken of process importance - all are considered to be equally important,

are unrealistic.

Various extensions have been proposed to deal with the aperiodic process issue, including that of the aperiodic server algorithm. However, this still leaves us with the second two items, which may lead to process blocking problems. This comes about in a number of ways, the two main ones being:

- process synchronizations involving more than two processes (in Ada, a Rendezvous), and

- attempted simulations access to protected resources.

If processes have to queue first-in-first-out while waiting for synchronization, a low priority process may precede one of a higher priority. Thus the higher priority process gets blocked until the first, lower priority one is serviced. This is called priority inversion.

Where resources have to be protected, they are accessed only via critical sections of code. To prevent contention and mutual deadlock between processes, such resources may be accessed by only one process at a time. That is, access is mutually exclusive. Once a resource is being used, other processes seeking to use it are forced to wait, irrespective of priorities. Thus a low priority process may again block out one of higher priority.

However, this description hides the true magnitude of the problem of blocking. With the existing arrangement, a process using a shared resource may get suspended. Thus any processes which are blocked are now subjects to delays introduced by the suspension. We must prevent this from happening wherever possible.

These problems can be overcome by applying the priority ceiling protocol. This protocol actually consists of two subprotocols: priority inheritance and priority ceiling.

In the first subprotocol, a low priority process in a critical section has its priority raised temporarily. It is elevated to the priority of the highest priority process which it is currently blocking (priority inheritance). Thus, only processes of higher priority than this can force it to suspend (which is a correct response when using priority scheduling).

In the second subprotocol, a priority ceiling is defined for every critical section. The priority of the ceiling has a single value - the same as that of the highest priority process which may enter that particular critical section. Any process working within its critical section executes at its own defined priority - until a second process attempts to access the shared resource. At this point, it has its priority raised to the ceiling value, so preventing suspension. It can, of course, be suspended by processes of even higher priority. However, by definition, these do not require use of the shared items, hence there is no priority inversion problem.

These two subprotocols guarantee that each scheduled process will only have to wait for, at most, one lower scheduling priority process to finish in a critical section. The sufficient conditions for feasible scheduling a set of n periodic process ordered is:

the rate monotonic protocol

$$\sum_{i=1}^{n}\left(\frac{T_C(i)}{T_P(i)}+\frac{T_B(i)}{T_P(i)}\right)\leq n(2^{1/n}-1) \qquad (3.2.6)$$

the earliest deadline as soon as possible protocol;':

$$\sum_{i=1}^{n}\frac{T_C(i)+T_B(i)}{T_P(i)}\leq 1.00 \qquad (3.2.7)$$

where $T_C(i)$ - the computation time of a process i

$T_P(i)$ - its period, and

$T_B(i)$ - the longest possible time spent waiting for a lower priority process to finish execution in a critical section.

The priority ceiling protocol has two important properties: it overcomes the problem of chained blocking, and it is deadlock free.

Integration of process importance and deadline scheduling

The main criteria for estimating the performance of a scheduling policy include:

Responsiveness. Even under worst-case conditions of processor loading, process response times must be acceptable.

Schedulability. A process schedule must be feasibly ordered.

Stability. In some cases, it may be impossible to meet all process deadlines. In such circumstances, specified essential deadlines must meet their timing requirements.

The scheduling policies discussed so far have tended to concentrate on the first two criteria. The third aspect has not been taken into account. Two factors contribute to instability: first, the presence of aperiodic processes; second, the conflict between process importance and response requirements.

A scheduling policy, to achieve higher scheduling stability, integrates these two constraints - importance and timeliness - treating all application processes as pre-emptive aperiodic types. In this strategy, a process schedule is initially ordered so that all process deadlines can be met. If at some later time it turns out that processes may miss their deadlines, a revised schedule of ready processes is formed. This is made feasible by removing a sufficient number of lower importance processes from the schedule. If the real-time system on which the application runs is distributed, those 'rejected' processes can be dispatched to another node.

However, there is a major drawback. There is an assumption, that is most unrealistically, that application processes have no precedence or exclusion relationships. A scheduling strategy that upgrades this model is the predictable dynamic scheduling policy (PDSP). This is based on the EDS principle combined with the dynamic priority ceiling protocol. The policy is applicable to aperiodic processes, and considers all process characteristics listed earlier, except the fault-tolerant constraints.

Using this policy, a schedule is ordered according to the earliest deadline as soon as possible principle. When a running process requires access to a shared resource it uses the dynamic priority ceiling protocol. Each time a process is readied, the following (sufficient) scheduling condition is checked:

$$T_D(l) - \sum_{i=1}^{l} T_C(i) - T_B(l) \geq t \qquad 1 \leq l \leq N \tag{3.2.8}$$

where $T_D(l)$ - the deadline of a process l,

$T_C(i)$ - its computation time

$T_B(l)$ - the length of the largest section in the list of critical sections that can block the process l, and

t - the current run-time of the application.

Processes are indexed according to their deadlines, low values having higher priority. The process index of “i” is less than that of “j”'s if “i”'s deadline is closer, or equal, to the time “t” than that of “j”.

If condition (3.2.8) fails when l is equal to m, $1 \leq m \leq N$, the scheduler has to exclude some processes from the subset of the ready queue. At first, a process with the lowest criticalness and the largest

laxity is removed from the subset. If the condition for the 'reduced' schedule in the ready queue is still violated, the process of removing is repeated until it is satisfied.

Actually, processes that have to be removed from the feasible schedule are only signed as 'rejected'. In a distributed embedded computer, those processes could possibly by transferred to another system node. Such a policy, though, is extremely unlikely to be adopted in a hard, fast system. However, bear in mind that condition (3.2.8) evaluates the task's worst-case computation times. Therefore, it is still possible that some of the rejected processes will, in practice, meet their deadlines.

3.2.4. Scheduler and Real-Time Clock Interrupt Handler

The real-time clock handler and the scheduler for the clock level processes must be carefully designed as they run at frequent intervals. Particular attention has to be paid to the method of selecting the processes to be run at each clock interval. If a check of all processes were to be carried out then the overheads involved could become significant.

System Commands Which Change Process Status

The range of system commands affecting process status varies with the operating system. Typical states and commands are shown in Figure xx and full details of these commands are given in Table xx. Note that this system distinguishes between processes which are suspended awaiting the passage of time - these processes are marked as delayed - and those processes which are waiting for an event or a system resource - these are marked as locked out.

The system does not explicitly support base level processes; however, the lowest four priority levels of the clock level processes can be used to create a base level system. A so-called free time executive is provided which if used runs at priority level n - 3 where n is the lowest-priority process number. The free time executive is used to run processes at priority levels n - 2, n -1 and n; it also provides support for the chaining of processes. The dispatcher is unaware of the fact that processes at these three priority levels are being changed; it simply treats whichever processes are in the lowest three priority levels as low-priority processes. Processes run under the free time executive do not have access to the system commands.

Dispatcher

The dispatcher/scheduler has two entry conditions:

- the real-time clock interrupt and any interrupt which signals the completion of an input/output request;
- a process suspension due to a process delaying, completing or requesting an input/output transfer.

In response to the first condition the scheduler searches for work starting with the highest-priority process and checking each process in priority order. Thus if processes with a high repetition rate are given a high priority they will be treated as if they were clock level processes, that is they will be run first during each system clock period. In response to the second condition a search for work is started at the process with-the next lowest priority to the process which has just been running. There cannot be another higher-priority process ready to run since a higher-priority process becoming ready always pre-empts a lower-priority-running process.

The system commands for process management are issued as calls from the assembly level language and the parameters are passed either in the CPU registers or as a control word immediately following the call statement.
In designing a real-time system it is important to know how the scheduler searches for work. In many systems the scheduler searches in strict priority order and hence the overheads in terms of the time spent searching for work will be increased if some of the highest priority processes rarely run. A careful assessment of process priority is required and particular attention will have to be paid to alarm

action processes. Such processes are normally accorded high priority; however, it is hoped that they will rarely be required.