5. REAL-TIME KERNEL AND OPERATING SYSTEM

Software design is simple if details of the lower levels of implementation on a specific computer can be hidden from the designer. An operating system for a given computer converts the hardware into a virtual machine with characteristics defined by the operating system. Operating systems were developed to assist both designers and operators in using computers.

The traditional approach is to incorporate all the requirements inside a general purpose operating system. Access to the hardware of systems and the I/O devices is through the operating system. In many real-time and multi-programming systems restriction of access is enforced by hardware and software traps. The operating system is constructed, in these cases, as a monolithic monitor. In operating systems running only single jobs access through the operating system is not usually enforced. However, it is a good programming practice and it also facilitate portability since the operating system entry points remain constant across different implementation. In addition to control and support the basic activities, operating systems provide various utility programs, for example loaders, linkers, assemblers, debuggers as well as run-time support for high-level languages.

A general purpose operating system will provide some facilities that are not required in a particular application, and to be forced to include them adds unnecessarily to the system overheads. Usually during the installation of an operating system certain features can be selected or omitted. A general purpose operating system can thus be `tailored' to meet a specific application requirement.

Recently operating systems which provide only a minimum kernel or nucleus have become popular; additional features, written in high-level languages can be added to it. In this type of operating system the distinction between the operating system and the application software becomes blurred. The approach has many advantages for applications that involve small, embedded systems.

In a simple operating system that has a general structure, the command processor provides a means by which the user can communicate with the operating system from the computer console device. Through it the user issues commands to the operating system and it provides the user with information about the actions being performed by the operating system. The actual processing of the commands issued by the user is done by the BDOS (Basic Disk Operating System) which also handles the input and output and the file operations on the disks. The BDOS makes the actual management of the file and input/output operations transparent to the user. Application programs will normally communicate with the hardware of the system through system calls which are processed by the BDOS. The BIOS (Basic Input Output System) contains the various device drivers which manipulate the physical devices and this section of the operating system may vary from implementation to implementation as it has to operate directly with the underlying hardware of the computer. For example, the physical addresses of the peripherals may vary according to the manufacturer; these differences will be accommodated in the coding of the BIOS. Devices are treated as logical or physical units. Logical devices are software constructs used to simplify the user interface; user programs perform input and output to logical devices and the BDOS connects the logical device to the physical device. The actual operation of the physical device is performed by software in the BIOS.

Access to the operating system functions is by means of subroutine calls and information is passed in the processor registers of the machine. Functions cannot be called directly from most high-level languages and this provides isolation between the operating system and a programmer using a high-level language. The isolation is deliberate; it is an example of information hiding. The connection between the high-level language and the operating system is made by the compiler writer through the provision of run-time support routines which convert the operating system into the virtual machine described by the high-level language. The isolation is not complete in that it is possible to call assembly co-routines from high-level languages and to pass parameters between the high-level language code and the assembly code. This does, however, require detailed knowledge of the system.

Again information hiding is used in that the details of physical implementation on the processor and of. the I/O devices are hidden within operating system and hence operations are performed on the operating system virtual machine.

5.1. Real-Time Operating Systems

Confusion can arise between multi-user or multi-programming operating systems and multi-tasking operating systems. The *multi-user operating system* ensures that each user can run a single program as if they had the whole of the computer system for their program. Although at any given instance it is not possible to predict which user will have the use of the CPU, or even if the user's code is in the memory, the operating system ensures that one user program cannot interfere with the operation of another user program. Each user program runs in its own protected environment. A primary concern of the operating system is to prevent one program, either deliberately or through error, corrupting another.

In a *multi-tasking operating system* it is assumed that there is a single user and that the various tasks cooperate to serve the requirements of the user. Cooperation requires that the tasks communicate with each other and share common data. In a good multitasking operating system task communication and data sharing will be regulated so that the operating system is able to prevent inadvertent communication or data access (that is arising through an error in the coding of one task) and hence protect data which is-private to a task (note that deliberate interference cannot be prevented, the tasks are assumed to be co-operating).

A fundamental requirement of an operating system is to allocate the resources of the computer to the various activities which have to be performed. In a *real-time operating system* this allocation procedure is complicated by the fact that some of the activities are time critical and hence have a higher priority than others. Therefore there must be some means of allocating priorities to tasks and of scheduling allocation of CPU time to the tasks according to some priority scheme. A task may use another task, that is it may require certain activities which are contained in another task to be performed and it may itself be used y another ask. Thus tasks may need. to communicate with each other. The operating system must have some means of enabling tasks either to share memory for the exchange of data or to provide a mechanism by which tasks can send messages to each other. Also tasks may need to be invoked by external events and hence the operating system must support the use of interrupts. Similarly tasks may need to share data and they may require access to various hardware and software components; hence there has to be a mechanism for preventing two tasks from attempting to use the same resource at the same time.

In summary a real-time multi-tasking operating system has to support the resource sharing and the timing requirements of the tasks and the functions can be divided as follows:

- task management: the allocation of memory and processor time (scheduling) to tasks,
- memory management: control of memory allocation,
- resource control: control of all shared resources other than memory and processor time,
- intertask communication and synchronisation: provision of support mechanisms to provide safe communication between tasks and to enable tasks to synchronise their activities.

In addition to the above the system has to provide the standard features such as support for disk files, basic input/output device drivers and utility programs. The overall control of the system is provided by the task management module which is responsible for allocating the use of the processor. This module is often referred to as the monitor or as the executive control program (or more simply the executive). At the user level, in addition to application tasks or system tasks are also used since in many operating systems some operations performed by the operating system and the utility programs run in the memory space allocated to the user or applications - this space is sometimes called `working memory.

From the user's viewpoint the two most important features of task management are how to create a task, that is make its existence known to the real-time operating system, and what scheduling strategy or strategies the real-time operating system supports. Task creation is largely a function of the interface between the operating system and high-level programming language.

5.2. Real-Time Kernel

5.2.1. Fundamentals

A real-time operating system has a kernel inside of it. In addition, it usually has a "shell" (operator interface), file I/O services, application load and management services, etc. A kernel, on the other hand, does not assume the availability of a keyboard, monitor, or disk drive. Instead, it is targeted as black boxes which do not have these amenities and it is assumed to run from ROM. Also, a kernel is normally linked with the application code to form a single, executable program.
The kernel usually denotes the lowest layer of an operating system which is directly above the hardware. A typical kernel may, among other things, provide the services listed below to other software layers of an operating system and to system users:

- *Booting and start-up of the initial system configuration.*
- *Process management.* Functions, such as creation and deletion of processes, delaying of processes, manage processes.
- *Interrupt management.* Typically, vectoring of interrupts, saving and restoring of state, and management of priority levels, including selective level enabling and disabling
- *Interprocess communication and synchronisation.* A set of mechanisms and related primitives necessary to support both functions

In addition, a kernel usually handles some supporting activities necessary to provide the services visible to other OS layers. Examples of such internal kernel operations include system time-base management and CPU allocation by means of basic process scheduling. Typically, real-time kernels provide the following additional services:

- memory management,
- timing,
- i/o management,
- error management.

In an embedded application a run-time version of a kernel may be all the system software that is provided on the target machine, whereas in a complex system the kernel is typically enclosed by additional layers of an operating system. The kernel must be one of the most stable parts of an operating system. Otherwise, any change in the interface specification of a kernel primitive may result in changes in all upper layers that call on it. Being closest to hardware, a kernel may be. thought of as the first-level functional extension of the bare machine.

5.2.1.1. Types of kernel

Single processor kernel. Any kernel contains data structures that represent processes and three basic kinds of routines: interrupt handlers, the primitives themselves, and a dispatcher. We focus here just on the parts of a kernel that implement processes. There are three basic ways to organise a kernel:

- as a monolithic unit in which each kernel primitive executes as an atomic action;
- as a collection of specialised processes that interact to implement kernel primitives; e.g. one may handle file I/O, and another, memory management; or
- as a concurrent program in which more than one user process may be executing a kernel primitive at the same time.

Each process is represented in the kernel by a process descriptor. When a process is idle, its descriptor contains all the information needed to execute the process. This includes the address of the next instruction the process will execute and the contents of processor registers.

The kernel starts executing when an interrupt occurs. Interrupts can be divided into two major categories: external interrupts from I/O devices and internal interrupts (also called traps) triggered by the executing process. When an interrupt occurs, the processor automatically saves state information so that the interrupted process can be resumed. Then the processor enters an interrupt handler; there is typically one handler for each kind of interrupt.

To invoke a kernel primitive, a process causes an internal interrupt by executing a machine instruction variously named a supervisor call (SVC) or trap. the process passes an argument with the SVC instruction to indicate what primitive is to be executed and passes other arguments in registers. The SVC interrupt handler first saves the full state of the running process. Then it calls the appropriate primitive, which is implemented within the kernel by a procedure. When a primitive completes, it in turn calls the processor scheduler. The scheduler selects a process for execution and then loads its state.

To ensure that the primitives are executed atomically, the first action of an interrupt handler is to inhibit further interrupts; the last action of the scheduler is to enable interrupts. When an interrupt occurs, further interrupts are inhibited automatically by the hardware; the kernel reenables interrupts as a side-effect of loading a process state.

Multiprocessor kernel. Hence, it can be stated that a multiprocessor kernel provides an object-oriented environment for applications. This environment consists of objects such as processes and semaphores and services such as signal and test provided by the kernel.. The multiprocessing paradigm encourages you to view your application as a collection of interacting objects rather than as a set of sequential operations. The object model turns out to be a better model for most real-time systems. The other models simply do not deal well with the complexities of multiple, simultaneous events which are typical in real-time systems. Multiprocessing, on the other hand is ideal for such systems, just define a process for each part !

Breaking a large job into smaller processes and then performing the processes one by one is a technique we all use in our daily lives. This divide and conquer approach is equally applicable to writing real-time systems software. A multiprocessing kernel takes this one step further by allowing the real-time system software to actually run as multiple processes. This has several advantages:

- Small processes are easier to code, debug than is monolithic software.
- A multiprocessing kernel provides a uniform, pre-defined interface between functions implemented as processes, thus minimising hidden dependencies between functions.
- The uniformity provided by a kernel is especially important if processes are created by different programmers.
- A pre-emptive multiprocessing kernel allows processes handling urgent events to interrupt less urgent processes.
- New features can easily be added by adding new processes.

Basically, a pre-emptive, multiprocessing environment is compatible with the way real-time software is created and is a natural environment for the same software to run in.
A shared-memory multiprocessor has two or more processors and at least some memory that is accessible to every processor. It is relatively straightforward to extend the single processor kernel to a multiprocessor kernel. The main changes we need to make are to store kernel procedures and data structures in shared memory, to access the data structures with mutual exclusion when that is needed to avoid interference and to change scheduler to exploit the multiple processors. There are some subtleties, however, that result from the special characteristics of multiprocessors.

We assume that internal interrupts (traps) are serviced by the processor that was executing the process that caused the trap and assume that every processor as an interval timer. For now, we also assume that each kernel operation and process can be executed by any processor. When a processor

is interrupted, it enters the kernel and inhibits further interrupts on that processor. This makes execution in the kernel indivisible on that processor, but it does not prevent other processors from simultaneously executing in the kernel. To preclude interference between the processors we could make the entire kernel as a critical section. However, this is a poor choice for two reasons. First, it unnecessarily precludes some safe concurrent execution. In particular, only access to shared data structures such as the free and ready lists is critical. Second, making the entire kernel into a critical section results in unnecessarily long critical sections. This decreases performance because it delays processors trying to enter the kernel, and it increases memory contention for the variables that implement the kernel critical section protocol.

The performance potential of multiprocessors has proven to be elusive in practice. One important reason is that current multiprocessor operating system structures and facilities tend to be warmedover rehashes of their single processor predecessors. Multiprocessors create the opportunity and the need to re-think various aspects of operating system design. Topics have included:

- Lock management techniques for shared-memory multiprocessors.
- Data structures for high-performance threads.
- Kernel organisation for high-performance interprocess communication on single processors and multiprocessors.
- Kernel abstractions providing effective support for the user-level management of parallelism.
- The relationship of processor architecture and operating system design.

At the heart of much of this work is the observation that the basic role of the kernel must change as we move to multiprocessor systems.

On multiprocessors, the opposite is true: threads, thread scheduling, and interprocess communication should be managed entirely at user-level, with the *kernel merely providing address spaces and occasionally reallocating processors* among them.

Distributed kernel. If all process interaction is by means of message passing, it is possible to execute a concurrent program on a distributed system such as a multicomputer or network of workstations. Here we show how to extend the shared memory kernel to support distributed execution. The basic idea is to replicate the kernel - placing one copy on each machine - and to have the different kernels communicate with each other using network communication primitives.

In a distributed program, each channel is stored on some machine. For now, assume, that a channel can have any number of senders but that it has only one receiver. Then the logical place to put a channel's descriptor is on the machine on which the receiver executes. A process executing on that machine accesses the channel as in the shared-memory kernel. However, a process executing on another machine cannot access the channel directly. Instead, the kernels on the two machines need to interact.

The kernel on each machine contains descriptors for the channels stored on that machine and for the processes that execute on that machine. As before, each kernel has local interrupt handlers for supervisor calls (internal traps) timers and input-output devices. The communication network is a special kind of input-output device. Thus each kernel has network interrupt handlers and each kernel contains routines that write to and read from the network.

When a kernel primitive - executing on behalf on an application process - needs to send a message to another machine, it calls kernel procedure net-write. This procedure has three arguments: a destination processor, a message kind and the message itself. First, net-write acquires a buffer, formats the message, and stores it in the buffer. Then, if the writing half of the network controller is free, an actual write is initiated. Otherwise the buffer is inserted on a queue of write requests. In either case, net-write returns. Later, when a write interrupt occurs, the associated interrupt handler

frees the buffer containing the message that was just written; if its write queue is not empty, the interrupt handler then initiates another network write.

Input from the network is typically handled in a reciprocal fashion. When a message arrives at a kernel, the network read interrupt handler is entered. It first saves the state of the executing process. Then it allocates a new buffer for the next network input message. Finally the read handler unpacks the first field of the input message to determine the kind and then calls the appropriate kernel primitive.

The net-read-handler services three kinds of messages: SEND, CREATE-CHAN and CHAN-DONE. These are sent by one kernel and serviced by another. The kernel's dispatcher routine is called at the end of net-write-handler to resume execution of the interrupted process. However, the dispatcher is not called at the end of net-read-handler since that routine calls a kernel primitive - depending on the kind of input message - and that primitive in turn calls dispatcher.
For simplicity we assume that network transmission is error free and hence that messages do not need to be acknowledged or re-transmitted.

5.2.1.2 Kernel structure

Monolithic kernel. An operating system is a very large piece of software; for example, Sun's UNIX kernel (the basic functions) occupies over a megabyte. The system calls can be partitioned into those which invoke file storage services, those which request terminal I/O, those which request network communication, and so on. The term subsystem is often used for the major internal modules of operating systems. As operating systems grow in size a great deal of research effort was devoted to considering how best they should be structured. One proposal was to impose a strictly layered structure on these functions introduced above.

Layered kernel. The idea is that each layer in the hierarchy creates a single abstraction. Any layer uses the services provided by lower layers and provides a service for the layers above it. An advantage is that a change at a given level does not affect lower levels. In practice, systems have not been designed with their functions in a single strict hierarchy. This is because of the difficulty of choosing one specific layering of functions. For example, a memory fault could not be reported to the operator's console because the console manager was placed above memory in the hierarchical structure.

Certain modules fall naturally into a hierarchical relationship, and this is exploited in system design. For example, a module concerned with driving a disk comes naturally "below" a module concerned with providing a file service. A layered structure is used for communication software. If a concurrent system is to be built on distributed hardware many of the component computers may have very limited functionality. A component of a process control system may be dedicated to monitoring the process; a component of s distributed system may be dedicated to providing a file storage service or to function as a gateway between networks, concerned only with communications handling. Such components do not need to run a complete operating system, neither would the many programming support utilities provided for human users be needed there. For this reason an open operating system structure is advocated.

Microkernel. The idea here is that those functions which are needed universally, by every component of a system, form a minimal kernel, sometimes called a lightweight kernel or microkernel. These names tend to be used now rather than "open operating system", partly to avoid confusion with the term "open system" which was also coined in the 1970's for use in communications standards. In the latter case, the term means that a system is open to communication in a heterogeneous world. To give an example, if a computer has no local disks because storage is provided across the network, then modules concerned with storage need not be present in that component of the system..

A small kernel has the advantage of being easier to engineer and maintain. Also, the services that are implemented above rather than within the kernel can easily be recompiled and loaded and may

therefore be changed (or omitted) in certain configurations. Processes that are executing kernel functions run at a higher priority than user-level processes. They made be privileged to carry out certain functions that are forbidden to user-level processes.

There has been a good deal of experimental work on what should be included in a minimal kernel. Every computer has at least one processor, some memory and at least one device, for example a terminal or network connection. The aim is to implement efficient mechanisms in the kernel and as much as is reasonable at user level. If you are sure that you always want a particular function to run as fast as possible, whatever applications the kernel is to support, you should include it in the kernel. You then give it priority over every function you might ever implement at user level. This is a difficult decision to make. There is a trade-off involved between making a service part of the kernel, and therefore fast but inflexible, and keeping the kernel as small as possible for maintainability and low overhead. Any particular function may provide a slower service when it is located at user level than when it is provided as part of the kernel.

The kernel overhead is the time spent in the kernel between an event occurring and the user level process that it is interested in it being scheduled. In real-time systems it is essential to be able to bound this time; the requirement is to be able to express the maximum possible delay rather than to obtain high speed as such. A minimal kernel makes this feasible. The requirements of the applications that run above a minimal kernel can be expressed by indicating the relative priorities of these user-level processes. With good design, the minimal functionality of the kernel should lead to low overhead and the potential for high performance which is essential for many network-based server computers. Any inefficiency in a kernel design affects every application, however high its priority.

The kernel provides efficient basic mechanisms; policies are expressed flexibly at user level. Programmers can control the relative priorities of user-level software. To summarise, a microkernel-based system has the following potential advantages:

- The system configuration can be tailored for dedicated servers.
- A small kernel is easier to engineer and maintain.
- The services which now run above the kernel are easier to engineer, tune, update and maintain.
- The time spent executing the kernel can be bounded.

A potential disadvantage is that a given service will respond more slowly to its clients if it is implemented at user level instead of inside the operating system.

A microkernel should be able to form the basis for a traditional operating system which provides a programming support environment. A problem in moving to a new operating system design is that a vast amount of software has been written to run above the old operating system. Ideally, a microkernel should provide a set of abstractions from which higher-level services can be built. It should be possible to build a conventional operating system interface above a microkernel and run existing software above that. This approach has been taken by several microkernel projects. The UNIX emulation can take UNIX system calls and translate them into calls on the microkernel services.

There could be a comparison between microkernel (for example Mach and QNX) and monolithic (traditional Unix) operating systems. The basic notion of a microkernel consists of devolving as much functionality as possible into processes rather than the kernel itself; different systems take different approaches to implementing this. For example, some systems (such as Mach) leave device drivers in the kernel, and place higher-level services (such as file systems) outside; others (such as QNX) move device drivers outside of the kernel. For example, most modern monolithic kernels now implement multiple threads of execution and fine-grained parallelism. Architecturally, this approach

begins to appear similar to a microkernel with several kernel-space processes working from shared memory.

As an aside, people often complain that the Mach system can't be a `real' microkernel, because it is so large (at least, this is the argument most frequently cited). However, the automatically-generated code stubs contribute very significantly to the size of the kernel, and that some size reduction would be likely if the stub generator produced better code. As mentioned above, the leaving of device drivers in the kernel also contributes to Mach's size. Debating microkernels versus monolithic kernels on the basis of kernel size misses the central, architectural point. In the same way as the point of a RISC processor is not to minimise the instruction count, but rather to make a different trade-off between what is implemented in the processor instruction set and what is implemented in other ways, the microkernel architectural issue is to determine which services are implemented in the microkernel, and which services are implemented external to that microkernel. By making appropriate choices here, the goal is to enhance various OS attributes in a manner that might not be addressable with a monolithic kernel OS. System attributes such as performance, flexibility, real-time, etc. are all variables which are taken into

Adaptable kernel. How it is possible to increase the performance for resource-intensive applications, such as multimedia and databases? The key idea of an adaptable kernel architecture: an extensible microkernel, in which application-specific code can be embedded. An application can define new resources, which are split into a user-level component and a kernel-level component. The kernel contains mechanisms to permit the embedding of application-specific code (the kernel-level components) in a way that is completely protected.

5.2.1.3. Implementation of the kernel

The exact implementation of the kernel and the details of the internal algorithms used can have a profound bearing on the resources the kernel needs to run. A traditional assembly language implementation produces a monolithic object that must be loaded into the target system in its entirety. A more modern approach involves writing as much of portability and maintainability of the kernel and provides performance advantages, since the overhead of a trap interface to the kernel is avoided. The disadvantage of this approach is that, if all kernel services are used, more code space will be required. But note that since the kernel is structured as a library, only the system calls you actually use get linked into your target image. So if your application does not use all of services - and very few applications use all of the services provided by a kernel - the version of kernel that gets linked into an application will be smaller.

Evaluating the RAM required to support a kernel is less straightforward since it is highly application dependent. For instance, if you have six tasks in your system, the kernel must create six Process Control Blocks (PCBs). These PCBs hold information such as the processes' priorities, current states, and stack pointers. Depending on the implementation, the PCB may also include space for pointers that will link it into the process ready list or other lists maintained by the kernel. Other kernel objects such as partitions and message queues also require control blocks. Each process must also have its own stack. The stack is used primarily by the application but also by the kernel to process system calls and to save the registers of the task when the task is not executing.

5.2.2. Basic Services of the Kernel

The basic services and their features of the kernel are as follows:

1. *Booting and start-up of the initial system configuration* All kernels provide some form of this service, but operating systems differ greatly in the details of how system initialisation is actually performed
2. *Process-management* Obviously, the first necessary function is to create a process. Suspending and resuming of processes may be indirectly accomplished at the user level by means of primitives, so a minimal kernel need not provide specialised system calls for that purpose. In

order to support clock-driven forms of scheduling as well as the requirements of real-time applications, the kernel should provide a system call to delay execution of a process for a specified period of time.

3. *Interrupt-management* Given that messages are also used to synchronise interrupt-service routines with hardware interrupts it is sufficient to provide some services for manipulation of the interrupt levels, such as enabling of a level by system call(s).
4. *Interprocess communication and synchronisation* Since messages are to be used for both purposes, some sort of SEND and RECEIVE operations should be implemented in order to provide the most general many-to-many mapping between senders and receivers, The kernel should manage mailboxes and support at least their creation by system call(s).

Initialisation Whenever a computer system is started, the operating system must initialise itself and the hardware and create the environment for the initial configuration of processes. When all this is done, control may be passed to the first user process. From that point on, the system becomes event-driven in the sense that the behavioural patterns of executing processes, together with the occurrences of events that cause rescheduling, dictate its further state transitions. In general, an operating system acts both as a watchdog by intervening to do the housekeeping necessary to record and complete the required state transitions and as a controller charged with maintaining a certain level of performance. Allocation of resources and more sophisticated scheduling policies are usually delegated to higher-level layers of an operating system, leaving mostly housekeeping and basic mechanisms to kernels.

Process management. The kernel should provide flexible mechanisms to be used as building blocks by higher software layers. In particular, a kernel should provide mechanisms and implementation of policies such as those governing resource allocation. By choosing event-driven scheduling, it may appear that kernel violates this principle. However, this is not the case, because some basic scheduling must be done by the kernel in order to control the execution of processes. In real-time systems, the priority-based scheduling offered by the kernel may be adequate for application to all processes. In systems with variable requirements, operating system processes and interrupt-service routines are often scheduled on the basis of their priority, whereas user processes and utilities may be handled by another scheduling algorithm implemented by an upper-level layer of the operating system. The tools for implementing additional scheduling mechanisms, such as time-slicing, can be provided in kernel by building on the existing support for interval-timer and system time-base management.

The kernel provides only the means for creation of objects, such as mailboxes and processes, but not for their termination. These functions, as well as other system services such as the remaining process-management system calls should be part of a kernel which is to serve as a foundation for a complex operating system. On the other hand, in some embedded applications where the size or the overhead of the executive may be critical the kernel may be adequate as it is required. Thus, system processes and real-time processes tend to be permanent, contrary to the transient processes common in program-development systems. In order to make the kernel useful in such different environments while keeping it as simple as possible, the kernel structure must be flexible as much as possible.

The kernel can keep track of processes and their individual state changes by maintaining a process descriptor for each active process. The initial form of a process descriptor can be established at the process-creation time, which is indicated by invocation of the CREATE system call. Static process attributes, such as initial priority and starting address, are commonly entered into its process descriptor at this time.

A newly created process is usually made ready and its process descriptor is placed into the Ready list to await selection by the scheduler. Following a burst of CPU activity, the running process may pass control to the operating system or be pre-empted as a result of the occurrence of an interrupt. In either case, the context of the running process must be saved, say, in the related process descriptor,

for subsequent restoration when the process is scheduled to run again. In a multiprocess system, a number of active processes and stages when they are active may change at any time. Their collective state can be tracked by the operating system by means of system lists of process descriptors. As a result, the kernel ought to include some form of Ready and Suspended lists.

In order to support the delay function, the kernel must track of the elapsed time. This is usually accomplished by means of a dedicated interval timer that is initialised and serviced by the operating system. Once started, the timer generates interrupts in approximately regular intervals. If vectored to the operating system, these interrupts may serve as the system time base.

Communication and synchronisation. The interprocess synchronisation and communication mechanisms define the basic features and properties of real-time systems. One of the primary considerations to select these mechanisms is that the selected set of service (and thus system primitives) is functionally complete in the sense of supporting both interprocess exchange and synchronisation. Messages, which may serve both interprocess synchronisation and communication in centralised as well as distributed systems, represent the best solution.

A message facility may be implemented in several different ways depending on the particular forms of naming, copying, and buffering used in a given system. One form of the message exchange is the exchange via mailboxes. In order to provide the asynchronous communication of senders and receivers required for signalling and handling messages buffers are used at the kernel level. In order to avoid the issue of problems of protection and the dynamic memory-management, message-passing in the kernel may be implemented by passing a pointer rather than by copying the contents of a message from the sender's to the receiver's space. The message facility can also be used to signal the arrival of hardware interrupts to the user-provided interrupt-service routines. Providing a single mechanism to handle both interprocess interactions and interrupt simplifies the kernel. This feature may be provided by dedicating a mailbox to each potential hardware source of interrupt, such as terminals, disks, etc. Whenever an interrupt with priority higher than that of the running process is asserted by an I/O port, the kernel should do the preliminary low-level processing and send a system-signalling message to the related mailbox. A process wishing to service interrupts from a given source may do so by receiving the corresponding system messages via RECEIVE calls on the specific mailbox.

Thus the kernel must also perform the necessary housekeeping operations, such as interrupt-level management by means of masking interrupts according to the priority of the running program, context-switching, and some sort of interrupt acknowledgement.

5.2.3. Functions of the Kernel

5.2.3.1. System Start-up

Whenever a computer system is "cold-started," after being powered or a system crash, at least a portion of the operating system must be into main memory and given control of the CPU. This activity is called system booting or bootstrapping of an operating system. Typically, the hardware initially transfers control to a known address where a starting routine in the ROM is placed. This routine, called the bootstrap loader, can be used to bring the rest of the system gradually to main memory, for instance, from the secondary storage. In disk-based systems, the core portion of the operating system is often placed at a known address, called the boot block or boot area, of a known system-disk drive. Thus, the bootstrap-loader routine can include a form of a disk driver whose primary function is to load and activate the initialisation section of the operating system. This section can, in turn, load into main memory the rest of the operating system and complete the initialisation process. In memory-based systems the starting routine can simply transfer control to the operating system, which itself may be residing in ROM.

In any case, once in memory, the operating system typically initialises hardware and system data structures, creates the initial configuration of system and user processes, and transfers control to one of them. From that point on, control of the processor is transferred between active processes and the

operating system in response to and in accordance with the changes of system state and the resource-allocation policies in effect.

In embedded systems and in many applications, however, there may not be a single-user process which should be present and therefore activated automatically by the operating system in every installation. Thus, a more general solution is for the operating system to create a known process user-definable but system-initiated, whenever the system is started. At least a portion of the program for this process can be defined by the system manager or a programmer to include create calls for the initial configuration of user objects, such as processes and mailboxes. This process is always the first user process to receive control from the operating system. It is also usual to facilitate creation of a large number of objects by structuring this process so that it can process a set of user-provided object-definition tables. In disk-based systems, this function is often accomplished by having this process routinely execute commands provided in a known file consisting of object-activation commands. The system manager can edit this file or define its entire contents and thus have. the desired processes created whenever the system is started.

If the kernel is specified as a memory-based system, the code of processes to be run under it, is in memory when control of the processor is initially transferred to the kernel. The starting routine of kernel should thus initialise the hardware needed for the

proper operation of kernel. This may include initialisation of the programmable interrupt controller. In order to be as hardware-and configuration-independent as possible, the kernel should not initialise any other I/O ports that may be available in a specific system configuration. This task may best be accomplished by the individual interrupt-service routines themselves when they eventually receive control of the processor for the first time. The hardware initialisation is done with interrupts off, so that no I/O may take place until the system is ready to handle I/O properly.

The kernel must initialise its own data structures before passing control to the first user process. These include all system lists and pointers, such as Ready .and Delayed lists, and global-system data. At this point, the initial configuration of user processes and their related mailboxes can be created.

In the kernel-based systems control must be passed to this section of code whenever the system is started (booted). It turns interrupts off in order to initialise hardware, the programmable interrupt controller in particular. Other hardware and I/O ports are normally initialised by their related servicing drivers. With hardware prepared for normal operation, the kernel initialises its global variables.

Recalling that head pointers to the Ready and Delayed lists are structured as process descriptors nodes for efficient processing, pointers to these structures are initialised by a special operator.

Before passing control to a user process for the first time, the kernel creates the user-specified initial configuration of processes and mailboxes. The initial configuration is defined by means of a table consisting of static process descriptors and mailbox descriptors. The configuration table contains pointers to the table of static process descriptors of initial processes and contains a pointer to the table of addresses of initial mailboxes.

The initialisation process invokes CREATE system call to create each process specified by means of the initialisation table. With the static process descriptors of initial processes occupying adjacent positions in initial processes, the successor to the just processed static process descriptor can be reached by incrementing the pointer by the length of an static process descriptor. A similar approach is used to create the initial set of mailboxes. Finally, dispatch routine is invoked to schedule one of the created processes and to surrender control of the processor to it. As a part of this process, the interrupt system is enabled for the level of priority of the scheduled process.

5.23.2. Process-Management

Process Descriptor. A process descriptor usually contains a process's attributes and other information that the operating system needs about a particular process. For example; entries of

process descriptor include current status, priority, and allocated resources. It also contains the pointer fields necessary to link it into system lists corresponding to the current status of its related process. A process descriptor supports storing the context of its process when it is suspended or pre-empted.

Hardware interrupt-processing sequences and environment-switching instructions often use the stack for storing and retrieving the context they control. Thus, stack stores that part of a process's context that must be saved by software. For reasons of security, it is customary to separate the system stack used by the operating system from the stack accessible to user processes. One way to accomplish this is to save and restore the respective stack pointer whenever a transition from a user's process to the operating system, or vice versa, takes place. Some computer systems facilitate this process by providing multiple stack pointer registers in hardware, sometimes augmented with automatic switching between them when executing instructions that make transitions between User and System modes. In this way the integrity of the operating system's stack may be preserved.

However, maintenance of two separate stacks in a system does not solve the problem of a user process disturbing the stack that it shares with other user processes. In particular, if a stack is shared by several user processes, each process making a transition from the Running state must leave the stack exactly as it found it when dispatched. If a running process may be pre-empted by an external event, then it is difficult to use the stack, shared by several processes for storing their temporary variables. A much safer approach is to dedicate a separate stack to each user's process. In this way, whenever a user process is running, the stack pointer is set to that process's private stack area. Whenever a process switch takes place, the departing process's value of the stack pointer may be stored in its process descriptor. Likewise, the hardware stack pointer may then be set for the new running process by restoring the saved stack pointer value from the related process descriptor.

In this case any stack error affects only the corresponding process. In addition, each user process need not worry about matching the number of stack pushes and pops before passing control of the processor to the operating system and, ultimately, to another user process.

Static Process-Descriptor The creation of a process basically consists of initialisation of its process descriptor. In the kernel, systems programmers may provide the initial values of process attributes by means of a special data structure, the static process descriptor (SPD). The static process descriptor contains a process's attributes, such as name, priority, and indication of some special hardware requirements, if any. If the memory allocation of the kernel is static, process that wishes to create another process must provide sufficient memory space for the process descriptor and the stack of the new process. Addresses of these memory areas and the programmer-assigned process attributes, such as name and priority, are combined into a data structure called the static process descriptor (SPD). It stores the starting address of the related process, the lowest address of the reserved stack area and its maximum size, respectively. It also contains a pointer to the memory area reserved for the process descriptor.

When used in small embedded systems, the kernel can easily make the often useful distinction between static and dynamic data so that they may be placed in ROM or RAM, respectively. The resulting code is often called ROM-able, and it is of great interest in embedded systems with no secondary storage. A static process descriptor for a process belonging to an initial system configuration is static in the sense that it is defined at system-development time and never modified at run-time. Thus, the static process descriptors can safely be placed in ROM. Placing the static data, constants, and the code of the kernel and user processes in ROM greatly simplifies booting and restarting of the system after a malfunction or a program failure. In addition, code and data in ROM are more secure in the sense that they cannot be erased by a running process.

In any case, with its code in memory and the static process descriptor formed, a process can be created by means of the CREATE system call.

System lists. The following lists are to be provided in the kernel :

- existing list
- ready list
- delayed list
- waiting-for-message (process descriptors, one list per mailbox)
- waiting messages (messages, one list per mailbox)
- mailboxes (mailbox descriptors).

The kernel data structures include one each of the following lists: Existing, Ready, and Delayed. In addition, there is potentially one list of messages and one list of processes per each active mailbox in the system. The process descriptor of a created process is placed into the Existing list and the Ready list. The process may subsequently move between Ready, Running, Delayed, and Waiting-for-message states. Whenever a process changes its state, the corresponding kernel routines record the resulting state of the system by moving the related process descriptor between the corresponding system lists. Processing of an event that results in a change of the system state thus typically causes a system object, such as a process descriptor, to be removed from one list and placed into another. The operating system lists are implemented using a linked-list representation. A process descriptor must provide link fields for the links required by various system lists. There is a link field assigned to the Existing list. Since Ready and Suspended states of a process are mutually exclusive, at any time a given process descriptor may participate only in either the Ready list or one of the two forms of the Suspended list (Delayed or Waiting-for-message). To handle these lists a single multipurpose link field in a process descriptor is used. Each process descriptor has one link field per each type of list. As a result, when a process descriptor is moved from one list to another, it is sufficient to update the pointers of adjacent nodes accordingly without ever having to copy the contents of a process descriptor from one place in memory into another.

Ready list and its handling. While a process's membership in other lists varies with its state transitions, a process remains in the Existing list from its creation until it terminates. Typical operations on this list are searching for a given process, for example, to check the validity of a parameter supplied to a system call, and reading of a process's attributes, such as priority. The Ready list is used by several kernel routines, and the efficiency of certain operations on it may have a significant impact on the overall performance of the kernel-based systems. The Existing list contains process descriptors of all active processes. It may also include all other system lists, such as Ready, within it. For instance, the Ready list may be obtained from the Existing list simply by regarding each process descriptor whose status field is set to READY as also belonging to the Ready list. An alternative is to implement a separate Ready list that contains only the PCBs of processes in the Ready state.

The Ready list is updated whenever a process is suspended (removal of the related process descriptor) and made ready (addition of the related process descriptor). The Ready list is searched for the highest-priority node whenever a new process needs to be scheduled. Thus, the kernel routines .perform three types of operations on the Ready list:

- insertion of a node,
- deletion of a node, and
- searching to find a node with specific attributes.

If the Ready list is included within the Existing list, changes of state from Ready-Suspended and Suspended-Ready can be recorded in place without the need to move the related process descriptor. If an operating-system pointer points to the target process descriptor when a state change needs to be performed all that is needed is to update the corresponding process descriptor’s status field. In this case a new process can be scheduled by searching all process descriptors and selecting the one with the highest priority among those whose status field is set to READY.

A separate Ready list requires updating of the pointers necessary to remove a process descriptor or to insert a process descriptor when the corresponding process makes a Ready -> Suspended or Suspended -> Ready transition, respectively. The number of active processes is likely to be strictly smaller than the total number of processes in a given system, with the possibility of being considerably smaller when a relatively large number of processes is suspended for some reason or another. As a result, implementation of a separate Ready list in the kernel is likely to be more efficient than its embedded alternative. The time of searching can be reduced further, to constant time, when a separate version of the Ready list is sorted by the decreasing priority of participating processes. In this case, the first process descriptor in the Ready list always belongs to the highest-priority ready process, and scheduling is reduced to setting a system pointer to it.

Insertions into the Ready List. A kernel routine inserts a process into the Ready list. This procedure is called by other kernel services whenever a process needs to be "made ready." This operation is quite frequently needed by other parts of the kernel. This procedure inserts a process descriptor, passed as a parameter, into the Ready list such a way that preserves its priority ordering. In particular, the new process descriptor is inserted after all those of higher or equal priority and ahead of all those of lower priority than its own. The processes of equal priority are placed in the Ready list in a first-come, first-served order. The parameter passed to this procedure is a pointer to the process descriptor to be processed. To find a proper place in the Ready list where the submitted process descriptor is to be inserted, the procedure searches the Ready list for the first process descriptor of lower priority. If one is found, the new process descriptor is inserted before that node in order to preserve the priority ordering of the Ready list. The status field of the new entry is set to ready, and the global variable “running” is set to the first entry in the Ready list in case that inserts before the first node takes place.

There is the following problem in every multiprocess system is what the operating system should do when there is nothing to do. For example, this situation occurs when the running process suspends itself, by attempting to receive a message from an empty mailbox and no other process in the system is ready. As a result of the change of system state it is necessary to reschedule the system, thus the scheduler is invoked to select the next running process, but it finds the Ready list empty. In principle, the system has to wait until a process becomes ready and then pass the control to that process. To reschedule the system the scheduler has to test the state of the Ready list until it becomes non-empty as a result of the occurrence of an event that causes a process to become ready. This approach is ineffective in many systems where the scheduler executes at a higher priority than user processes in order to be protected from potentially harmful preemptions. While in a high-priority wait loop, the operating system may never be able to detect the lower-priority event needed to get the user processes back in motion. This problem is handled in many operating systems with the aid of a special “null” process that is created by the system as a part of the start-up sequence. The “null” process is designed to never terminate and to always be ready. As a result, the Ready list is never empty and the problem is solved. Moreover, execution of the scheduler, a time-critical part of the system, can be made faster by eliminating the need to test for emptiness of the Ready list. The simplest way to implement the “null” process is as a busy-wait loop. The “null” process monopolises the processor and therefore must be the lowest priority process in the system. All other events that have higher priority will thus primped “null” process and be recognised and processed by the system. Being executed only when the processor would otherwise be idle makes the “null” process an excellent vehicle for gathering performance statistics. Namely, by keeping track of the time spent executing it, the processor utilisation by user processes and by the operating system can be measured. Another useful activity that can be delegated to the “null” process is to execute various diagnostic programs and thus assist in monitoring the system integrity. .

The purpose of “null” process is to ensure that the Ready list is never empty. Since priority ordering of the Ready list and event-driven scheduling result in the scheduler being reduced to a single assignment in the course of process dispatching, the dispatch routine selects the next process to run

in the kernel. Thus, existence of the this process makes the execution of dispatch routine faster, since it does not have to test for emptiness of the Ready list.

Process creation. Creation of a process entails initial formation of the process descriptor, typically followed by moving the process into the Ready state: When a process does not exist any more, its process descriptor is normally returned to the pool of free process descriptors.

A run-time validity check of process names passed as parameters to system calls is a useful feature in an operating system. For example; a request for an indefinite suspension or forceful abortion of a process normally returns an error indication if the specified process is non-existent, perhaps because it terminated or was never created. Authentication of process names can be conveniently performed when a list of all non-dormant processes in the system is maintained by the operating system. Although the validity of process names is not checked by the kernel, such a functional extension is made possible by maintaining the list of Existing processes. The elements of this list are the process descriptors, which can be added at the process creation time and removed when the related process departs the system.

The CREATE system call associates a process descriptor with the newly created process, enters the initial values of the process attributes, such as process priority, and enters the process descriptor into Existing processes and Ready lists. After creation a process no rescheduling is done in the kernel, and control is always returned to the calling process. As a result, the newly created process may become the running process only upon occurrence of an event that causes the scheduler to be invoked, for example; a timer interrupt.

The CREATE procedure takes one parameter, the address of the static process descriptor of a process to be created. The CREATE procedure starts by forming a process descriptor in the area pointed to by the static process descriptor. While being created, a process descriptor does not belong to any system list. To create a stack; first, the top of the stack must be calculated and placed into the stack field of the process descriptor. This operation can be machine-dependent in the sense that the direction of stack growth, from higher to lower memory addresses, or vice versa, may vary from one computer system to another. Assuming the common case of downward growth, the initial value of stack can be obtained by adding the stack length to the lowest stack address, which are provided by the static process descriptor.

An important task of the CREATE system call is the preparation of the newly formed process descriptor so that control of the processor can be passed to a new process in a manner that is consistent with the way in which control is transferred to regular processes. A process that is not running always has the address of its next instruction (the value of the program counter) on its private stack. When such a process is resumed, dispatch routine eventually surrenders control to it by emulating a subroutine return with the process's program counter on top of the stack. For reasons of uniformity and efficiency, it is desirable to make both regular and new ' processes look the same to dispatch routine in this regard. This can be accomplished by having CREATE place the starting address of the new process onto the process's stack.

In particular, if the starting address of the process is pushed onto the process's stack, the subroutine return used by dispatch routine works for all processes. For this reason, procedure CREATE actually places the process's initial starting address, supplied by the static process descriptor, onto its private stack. In addition, the stack field of the process descriptor is set to point to the "pushed address" as the current top of stack, thus making it both unnecessary and even impossible for dispatch routine to differentiate between newly created and recurring processes. Naturally, this process is machine-dependent, and it may require some assembly language code.

The next activity of CREATE is to calculate the process's interrupt mask on the basis of its software priority indicated in the static process descriptor. The interrupt mask should have the proper format to be directly writeable into the hardware interrupt-mask registers in order to enable only the hardware levels of priority higher than that of the process in question. Obviously, this interrupt mask

is machine-specific. The interrupt mask is used by the dispatch routine while creating the environment for the new running process. Rather then being provided in a separate process descriptor field, the interrupt mask may be calculated by dispatch routine at run-time.

The CREATE system call also places the newly formed process descriptor into the Existing list. Since it is not sorted, insertions into this list are made at its front. The process descriptor is then inserted into the Ready. These list manipulations are performed with interrupts off in order to protect the global system pointers involved in the process. It is important that no scheduling is performed at the process-creation time. Consequently, the newly created process will be considered for execution following the next event that causes the rescheduling.

Process delay. This system call delays a process for a specified time. A process wishing to delay itself invokes the DELAY system call and specifies the duration of the desired delay. The operating system normally services such requests by means of two separate but cooperating activities. The calling process is first suspended and placed in the list of delayed processes, and the number of system clock-ticks corresponding to the duration of the delay is recorded in the related process descriptor. Secondly, whenever an interrupt from the interval timer is processed, the operating system decrements the number of remaining intervals in the corresponding process descriptors of all the processes in the Delayed list. Each process whose delay is found to be expired, by virtue of the related count becoming zero, is resumed by being placed into the Ready list.

After suspending the caller·and recording the number of time intervals (clock-ticks) to delay the caller, the DELAY system call causes dispatching of a new running process in a standard way. This system call initialises the programmable interval-timer for the desired duration of a system time unit, i.e. clock-tick. Following this, the corresponding interrupt level is enabled in order to allow recognition of timer interrupts. Processes whose delay has expired are then effectively resumed by being placed into the Ready list:

Delaying of a Process for a Specified Time An important process-management function of the kernel is delaying of a process for a specified period of time. Delaying of a process is performed by two separate but related routines; delay and check delay list. The first one is called by a user process wishing to be delayed. The calling process is suspended and placed on the list of delayed processes. The second one is actually an interrupt-service routine which decrements the remaining delay of the processes in the Delayed list and makes eligible for execution those processes whose delay is found to be expired.

The Delayed list. The DELAY procedure basically inserts entries into this list, whereas check delay list routine searches the Delay list and removes qualified entries. In most systems, the frequency of timer interrupts is higher than that of processes becoming delayed. Consequently, we should be primarily concerned with the efficiency of clock-tick processing.

The first option to be considered is whether the Delayed list should be sorted or not. Inserts into an unsorted list can be performed in constant time by simply appending new entries. The clock-tick processing with such a list, however, has a time complexity proportional to the total number of delayed processes. On the other hand, insertions into a sorted Delayed list, on the average, take operations. Clock-tick processing with this kind of list apparently has the same time complexity as in the previous case, because the remaining delays of all processes must be decremented upon each timer interrupt. Unfortunately, neither of the two approaches does much for reducing the overhead of the most frequent and time-critical operation. The problem lies with decrementing of remaining delays of all processes in timed wait. One way to solve this problem is to make delays relative to each other, rather than absolute in terms of remaining time units.

Alternatively, consider the sam m where d re recorded and processed relative to each other. If the Delayed list is sorted by increasing absolute delays, a requirement is satisfied. If an individual process's relative delay is expressed as the number of clock-ticks to wait after the delay of that process's predecessor expires. The predecessor's relative delay is, in turn, expressed in terms of its

predecessor's delay, In other words, relative delay of a process is the increment expressed in terms of clock-ticks that must elapse after expiration of all preceding relative delays in order for the process in question to leave the delayed state. Since the first process in the Delayed list has no predecessor, its relative delay is numerically identical to its absolute delay.

The primary advantage of relative delays is that only the remaining delay of the first process in the list needs to be decremented upon each timer interrupt. Since all other delays are relative and additive to it, no process descriptors of processes beyond the one at the head of the Delayed list need be processed. This results in a significant reduction of overhead of timer-interrupt processing. In particular, only one node is visited and processed per timer interrupt, which brings the time complexity of that operation down to constant time.

The DELAY operation. The procedure DELAY takes two parameters: the address of a mailbox, and the delay expressed in terms of system time units. The first parameter is ignored by DELAY. Specifying the duration of the desired delay in terms of the number of clock-ticks (system time units) is system-specific and generally not portable even within the same family of operating systems. The reason is that the time unit is often a system-generation parameter that may vary from installation to installation. The DELAY procedure acts only on those user calls whose desired delay is greater than 1 time unit. When called with an illegal argument, such as negative or zero time to wait, DELAY simply returns to caller with no error indication. For legal invocations, DELAY turns interrupts off in order to preserve the integrity of the Delayed list in the presence of concurrent updates.

The DELAY system call scans the list to find a place where the new entry is to be inserted. As with the Ready list, special processing for inserts at the beginning of the list and into the empty list are avoided by structuring the head pointer of the Delayed list as a process descriptor whose delay is 0. After visiting each node, the corresponding relative delay of the new entry is formed by updating the variable ticks. The related while loop terminates either when a place to insert is found or the last node is reached. In the latter case, depending on their relative delays, the new entry may have to be inserted before or after the last node.

If the relative delay of the new entry is smaller than that of the examined node, an insert before the examined node is performed. Accordingly, the relative delay of the examined node is updated and the roving pointer is backed-up to the node after which the insert is to be performed. Alternatively, the insert is to be made after the last visited node, and ticks is updated. The latter situation includes inserts after the very last node, as well as behind nodes with the same relative delay as that of the new entry. In this way, nodes with identical remaining delays are treated in FCFS order.

Once its place is found, the new entry is marked as a delayed process and inserted into the Delayed list by modifying the forward link of its predecessor and the back link of its successor, if any. Of course, the corresponding links of the process itself are set to its predecessor and successor, respectively. After updating its status and recording its calculated relative delay, ticks, the stack pointer of the delayed process is saved in its process descriptor and the process is suspended by being deleted from the Ready list. The resulting transient state cannot corrupt the kernel because it is made invisible to other routines by virtue of having the interrupts off. With processing of the significant event completed, the new running process is scheduled and dispatched.

Timer-interrupt processing. The delay list handler routine is effectively a process running under the kernel. It services interrupts from the interval timer by decrementing the remaining delay of the first process, if any, in the Delayed list. If the updated delay becomes 0, the top process and its potential successors with identical delay are removed from the Delayed list and placed into the Ready list. However, being a process, delay list handler must never be explicitly invoked by any routine or the associated timing it produces will be in error and the system may crash.

In general, when accurate timing is desired, the delay list handler process or its equivalent must have higher priority than any process which delays itself. The delay list handler process is assigned software priority 0, which is commensurate with the hardware level of priority it services.

Consequently, no process can primped the delay list handler in the kernel-based systems, and the high priority of delay list handler process provides for accurate tick keeping, subject to the accuracy and stability of the hardware clock.

Like most system and real-time processes, the delay list handler process consists of the initialisation part, which is executed only once, and the forever do part, which performs the normal processing. The first part of delay list handler contains a call to enable recognition of the corresponding hardware interrupts and then forms the address of the corresponding interrupt mailbox. An implementation of delay list handler should normally begin with the initialisation of the hardware interval timer so as to specify duration of a clock tick for the particular system configuration.

Once an interrupt occurs, delay list handler tests whether there are any delayed processes. If so, the remaining delay of the first one is decremented. Following this, a while loop is entered to remove all processes whose remaining delay is 0, if any. Each qualifying process is made ready and removed from the front position of the Delayed list. If more processes are left, the back pointer of the new first process is set to the list head. In general, when delay list handler adds one or more entries to the Read list the scheduler should be invoked to select the next process to run.

The delay list handler process continues by invoking RECEIVE to await another timer interrupt. In normal processing, the interrupt is not asserted until the full tick elapses, and delay list handler is suspended at the corresponding mailbox. The ensuing processing of the change of system state (the delay list handler process moving from the Running to the Waiting-for-message state) results in the rescheduling needed to take newly readied processes into consideration.

5.2.3.3. Interprocess Communication and Synchronisation

When a buffered-message facility is selected for the purposes of interprocess communication and synchronisation, the kernel must support management of mailbox and message objects. In addition, SEND and RECEIVE system calls need to be provided to system and user processes that wish to interact with other processes or system devices.

Mailboxes and messages. operating systems that support a mailbox-based exchange of messages often combine all information necessary for management of a , given mailbox into a data structure called a mailbox descriptor (MBD). A separate mailbox descriptor is usually dedicated to each .active mailbox. A mailbox descriptor may be formed and initialised at the message-creation time, indicated by execution of the corresponding system call. Not unlike the relationship between processes and process descriptors, the user's concept of a mailbox as a repository of messages is supported at the operating system level by creation and management of a related mailbox object.

When a running process sends a message to a specific mailbox by means of the SEND call, the message can be immediately delivered to a waiting process, if any. However, if there are no processes suspended at that mailbox by an unmatched RECEIVE, the message is left in the mailbox and the running process continues to execute. In this way, several messages can be accumulated in a given mailbox awaiting delivery to the receiver process. Since the kernel is to buffer outstanding messages; a portion of each mailbox descriptor may be used as a header of the queue of messages awaiting delivery at its related mailbox.

When a running process invokes a RECEIVE system call on a mailbox where no messages are awaiting delivery at that time, the calling process enters the Waiting-for-message state and remains suspended until arrival of. a message at the 'designated mailbox. Due to the nature of message exchanges in the kernel, it is possible that several processes may be awaiting delivery of messages at a given mailbox: Since each such :process may be resumed only when a message is delivered to it; mailboxes look like a good place for the matching of processes to messages and vice versa. The idea to queue outstanding messages at a mailbox may thus be extended to include the queuing of processes when receivers temporarily outpace senders of messages to a given mailbox.

The Waiting-for-message state of processes in the kernel can be handled by a layered version of the Suspended list, whereby process descriptors of waiting processes are placed at the particular mailbox descriptor where the message is to be received. In this way, the reason for suspension of a given process may be deduced from the identity of the mailbox where it is waiting.

A mailbox serves as a meeting place of processes and messages. The kernel keeps track of mailboxes by means of mailbox descriptors. A mailbox descriptor in the kernel must be structured in such a way that either messages (resulting from the execution of unmatched SENDs) or process descriptors of suspended processes (resulting from the execution of unmatched RECEIVEs) can be queued at it. Due to the nature of interprocess communication in the kernel, messages or processes, but not both; may be queued at a given mailbox descriptor at any time.

FIFO and priority queues for waiting processes and messages are widely used because FIFO is simple to implement an it ensures that, with the proper supply of messages, waiting processes cannot be subjected to indefinite postponement. The use of messages to signal interrupt arrivals in the kernel precludes the use of any queuing discipline other than FIFO. Some multiprocess operating systems use priority or processes, thus providing for potentially superior response time to high-priority processes than FIFO, but at the expense of the possibility of indefinite postponement and inconveniences when emulating semaphores.

In order to serve interprocess message communication, a mailbox must first be created by means of the corresponding system call in the kernel. Typically, a sender or the system-initialisation module performs this task. Creating a mailbox in the kernel is a fairly straightforward operation. It normally results in formation of the related mailbox descriptor capable of queuing messages or processes. When a mailbox descriptor is created, both queues are empty and their headers must be initialised to reflect that fact.

In general, however; requests to send messages to or to receive messages from a non-existing mailbox (deleted or- not created) should be denied by the operating system. Deletion of an existing mailbox is another service not offered by the kernel but which may be desirable in a-larger operating system: Both these extensions call for a list of all mailbox descriptors in the system. Entries to this list can be added whenever a new mailbox is created, and removal of entries can be made when the operating system is required to delete a mailbox. When only a validity check is desired, this list may be searched to find whether the target mailbox descriptor is in active use, say, by virtue of being created but not deleted. Requests to use active mailboxes can then be granted, and others can be denied, with a corresponding error code being returned to the calling process.

Mailbox descriptors and the mailbox list can be used in operating systems to check and declare affiliation of processes with active mailboxes by means of system calls. In such systems, a process may detach itself from a mailbox that it no longer uses. Similarly, a

departing process may be detached from all its mailbox holdings by the operating. Thus the operating system is enabled to detect unused mailboxes by keeping track of the number of processes actively attached to them. Mailboxes that are already not used can then be deleted by the operating system immediately upon detachment of the last related process.

A mailbox descriptor consists of head and tail pointers to the queue of messages and head and tail pointers to the Waiting-for-message queue of processes. Both sets of head and tail pointers are maintained in order to provide for efficient appending and removal of entries at opposite ends of the queues. When a mailbox is created, all queues are assumed to be empty and their respective pointers are set to nil. Since memory allocation is not performed by the kernel, a process wishing to create a mailbox must provide a record of the mail descriptor type and invoke a routine to initialise the mailbox descriptor.

Procedure that creates the mailbox takes one parameter, the address of the mailbox descriptor. This procedure initialises all pointers of the mailbox descriptor to nil and places the mailbox descriptor on

the Mailboxes list. No particular ordering of this list is required in the kernel, so the simplest and the most efficient method of adding entries to the front is followed. While the created mailbox descriptor is in the process of being inserted into the Mailboxes list , interrupts are turned off to ensure indivisibility of the operation.

Support for variable-length messages in the kernel requires some sort of message headers where the length of a given message may be recorded by the sender and referenced by the receiving process. The kernel provides buffering of messages awaiting delivery (sent but not yet received) it requires some means for linking them at a mailbox descriptor. The message header is probably the most convenient place for this link when unlimited message buffering per mailbox is required. Thus, each message consists of a fixed header and an optional data field of variable length. For example, empty messages used for synchronisation purposes normally consist of the header field only.

A message must be built in the sender's address space before it may be forwarded to a receiving process. Building of a message consists of placing data in the data field, if any, and setting the length field in the header accordingly. After setting the message type field, if necessary, the message is built, and then it may be sent by invoking the SEND routine. The link field of a message header should not be manipulated by the sending process. This field is used by the kernel for queuing messages at mailbox descriptors.

SEND message operations. The facility for sending messages, SEND needs two parameters: the identity of the mailbox (i.e. the mailbox descriptor) where the message is to be delivered and the message itself. When SEND is invoked by the running process, either processes or messages may be found waiting at the designated mailbox descriptor. When no processes are waiting, the message cannot be delivered immediately, and SEND places it at the mailbox descriptor before returning control to the caller, the running process in this case. Alternatively, if one or more processes are waiting, SEND delivers the message to the first in line.

Since delivery of a message results in both the running process and the process receiving the message becoming ready, the scheduler should be invoked to determine which process should receive control after the completion of SEND. Prior to that, the context of the running process must be saved in case it is pre-empted by the process receiving the message. In any case, the process receiving the message is made ready and its process descriptor is moved from the mailbox queue to the Ready list. Given the priority-based scheduling in the kernel, the next process to run is the highest-priority ready process. The scheduler in the kernel is thus reduced to finding the highest-priority process in the Ready list. After scheduling, SEND invokes the dispatch routine to establish the run-time environment for, and to transfer control to, the new running process.

SEND system call begins by saving the context of the calling process. The saved context is restored when control is eventually returned to the process that has invoked the system service. In the case of SEND, if the message cannot be delivered immediately and control is returned to the sender, the sender's context is restored. Alternatively, if the message is successfully delivered, its sender may or may not be the next process to be dispatched. Thus, SEND cannot blindly restore its caller's context, and the restoring of the proper context is left to the dispatcher routine, which learns the identity of the next running process from the scheduler. The details of the context save-restore operations are system-specific, and they depend on such things as the format of the system data structures, the type of instruction used to invoke the operating system, the amount of context switch (process switch) done by the underlying hardware, the memory-management and protection policies in effect, and the like.

The selection of the next running process within the body of SEND is defined by the following condition. When the message is delivered to a waiting process, there are only two viable candidates from which to select the next running process: the caller of SEND (previously running process) and the process that receives the message. Given the priority-based scheduling in the kernel, probably a more efficient version of SEND may be constructed so that the priorities of the two processes are

compared and the scheduling decision is made within SEND. In this way the scheduler need not be invoked, whereas saving the context of the running process and restoring the context of the message recipient are performed only when the priority of the running process is comparatively lower, so that invocation of SEND results in its pre-emption.

The functional specification of SEND is based on the philosophy that each particular operating system function should be entrusted to a separate routine that should have the sole authority to perform the given function. Consequently, SEND is not allowed to make any scheduling decisions, even when the algorithm is simple and only two processes are involved. By localising the focus of each activity into a dedicated operating system routine, the system is likely to be easier to comprehend, to develop, and to maintain, thus increasing the efficiency of programming and yielding a potentially more reliable product.

The kernel provides a buffered, indirect message-exchange mechanism via mailboxes. Messages are of variable length, and they are not copied from a sender's to a receiver's space. Each mailbox is visible to the kernel by means of a dedicated message descriptor. A message descriptor must be capable of heading a queue of processes awaiting messages and a queue of messages awaiting delivery to processes. Both queues are serviced in FIFO order. Entries are placed at the rear of the queue and removed from its front. Depending on the speed of senders and receivers on a given mailbox, either the process or the message queue may contain entries, but not both. All mailbox descriptors in the system are contained in the Mailbox list.

The SEND procedure opens by turning interrupts off in order to ensure mutually exclusive access to the global pointers it manipulates. It is performed by a procedure which clears the master interrupts-on hardware flag to disable interrupts and leaves all other interrupt masks and control registers unchanged. Operating-system kernels commonly resort to the disabling of interrupts for providing mutual exclusion and indivisibility of certain sensitive operations, such as message delivery. The SEND procedure then examines the head of the process queue at the corresponding mailbox descriptor in order to determine whether some processes are waiting for a message. If the process queue is empty, the message must be placed to await delivery. To ensure FIFO delivery of waiting messages, the new message must be placed at the tail of the related queue. If the message queue is found empty, the head pointer is set to the new message. Otherwise, the new message is appended in a standard way by setting the link field of the last message, designated by the tail pointer to point to the new entry. The new entry is the last message in the updated queue, so its link field is set to nil and the queue-tail pointer is set to the appended message. After turning interrupts back on, SEND returns control to its caller.

When one or more processes are found waiting at a mailbox, the corresponding SEND delivers the message to the first of them. This process, the message recipient, is first removed from the front of the queue of waiting processes. After updating the head pointer and the tail pointer in case only the removed process was found in the queue, SEND places the address of the message to be delivered, into the delay field of the recipient process. The message remains there until recipient eventually becomes the running process and the message is actually delivered to it. Message delivery is completed by the dispatch routine. If invoked by a user process, SEND saves its context using a procedure. Its purpose is to save the stack and pointers of the procedure that the caller of SEND. This procedure prepares a return from the SEND call. This is necessary whenever a user process invokes a kernel service that causes its caller to be suspended. The SEND procedure calls this procedure to save part of the suspended process's state in such a way that dispatch routine can return control to the user process when it is scheduled to run again. Following invocation of this procedure, the procedure SEND makes the message recipient ready by inserting it into the Ready list. The next active process is then scheduled and dispatched. Given priority ordering, scheduling in the kernel is reduced to a trivial operation of "selecting" the process at the front of the Ready list. The global variable running points to this process, thus making it possible to combine scheduling and

dispatching into an invocation of dispatch routine to prepare the environment for and to surrender control to the running process.

RECEIVE-Message Operation In the kernel RECEIVE system call accepts messages from senders. It returns a message to the calling process. When invoked by the running process, RECEIVE may find one, more, or no messages and perhaps some processes waiting at the designated mailbox. Upon finding at least one message, RECEIVE removes the message from the mailbox and delivers it to the calling process-in this case the running process, which is not pre-empted. Alternatively, if no message can be delivered, the running process is suspended at the mailbox descriptor and ,a new running process is scheduled and dispatched.

RECEIVE takes two parameters: pointer to the mailbox where the message is to be received and a time limit. When the running process wishing to receive a message invokes the RECEIVE function, either messages or waiting processes may be found at the designated mailbox. In the former case, the calling process receives the first message from the queue. Removal of a message in RECEIVE is a standard deletion from the front of a queue. The address of the. removed message is placed in the message variable. The link field of the removed message is set to itself. This is done in order to allow for differentiation between messages that are in a queue and those which are not simply by examination of their link field. This is used by interrupt service routine.

After re-enabling interrupts, the pointer to the message is returned to the calling process, which resumes its execution. Similar to SEND, RECEIVE runs with all interrupts off and, upon exit, re-enables only the interrupt levels that were enabled prior to its invocation. This is accomplished a procedure, which sets the master interrupts-on hardware flag and leaves all other interrupt masks and control registers in tact.

When RECEIVE finds no messages, the calling process is suspended by being removed from the Ready list. The removed process is then placed at the end of the queue of .processes waiting for messages at the designated mailbox. Context of the process is saved on its stack, and the new running process is scheduled and dispatched.

5.2.3.4. Interrupt-Management

Interrupt servicing in the kernel relies heavily on the message-exchange mechanism. Basically, each hardware interrupt is intercepted by the kernel and converted into a corresponding message, which is then sent to the mailbox dedicated to the interrupt level in question. The operating system should take responsibility for the dynamic disabling of lower interrupt levels and the enabling of higher ones, i.e. it can change the priority level of the running process. The kernel is implemented on system configurations that include an interrupt controller or equivalent hardware support for all three levels of interrupt masking :

- the system-wide level (interrupts on/off),
- the individual priority level (selective masking), and
- the individual device level (I/O port control register).

In order to support priority-based scheduling, the kernel must keep track of the priority of each process. To achieve this a system programmer assigns priority to each process and to communicate this parameter to the operating system, so that the priority can be entered into the process descriptor at process-creation time. In order to keep the number of system services low and to reduce the overall system complexity, the kernel should use static priorities.

An operating system should differentiate between hardware and software priority levels. The hardware priority levels are assigned to hardware devices and serviced by the interrupt-service routines. Hardware priorities are commonly used to drive the interrupt-enable/disable logic and they are placed above software priorities to achieve low latency in response to interrupts. The software priority levels are assigned to all other processes. They are scheduled according to their software priorities when no interrupt-related activities are ready to use the processor. Due to the same

handling of interprocess communication and of interrupt signalling via messages, There is no strict differentiation between interrupt-service routines and other processes and consequently between hardware and software priority levels. However, some means are required for instructing the kernel which particular interrupt levels should be enabled and which disabled when an interrupt-service routines is running. A simple scheme for accomplishing this may be to partition the kernel priority levels into interrupt-related and software levels.

In many cases users are allowed to explicitly enable and disable individual levels of interrupt. At system-initialisation time; user-serviced interrupt levels are typically disabled and mapped to the operating-system routine for detection and handling of false interrupt requests which could otherwise hang up the system. Some process, most commonly a user-written interrupt-service routine, must then first explicitly enable the interrupt level of the device that it services. One way to implement user-level interrupt enabling is to maintain an internal system-interrupt mask which is initially set to all interrupt levels off. The system service to enable a specific level may then simply turn on the corresponding bit or bits in the system mask. Finally, the operating system, when passing control to a process, may enable only those interrupt levels which are both higher than that of the running process and enabled by an earlier system call.

Preliminary processing of all interrupts in kernel-base systems is performed within the kernel. Consequently, all interrupts are vectored to the operating system, to the interrupt handler routine. When an interrupt request is asserted and accepted, control is transferred to this handler. It identifies the particular source of interrupt, saves the pre-empted process's state, and sends a message to the dedicated mailbox signalling the interrupt.

Interrupt Mailboxes and Priorities. In the kernel a separate mailbox is assigned to each hardware level of interrupt for the exchange of interrupt-signalling messages between the kernel and the interrupt-service routines. A process wishing to service interrupts from a given source can do so by invoking the RECEIVE function on the corresponding interrupt mailbox. For this reason, interrupt mailboxes in kernel systems are declared as global variables. The kernel is capable only of distinguishing between different hardware levels of interrupt. In systems where interrupt-request lines of several devices are connected to a single level, a software routine must be employed to "demultiplex" requests for service by identifying the specific originating device. The logic of such a routine can be structured along the lines of a polling sequence.

Since a special system message is needed for signalling interrupts occurring at a particular hardware level; the kernel creates one such message and appends to each descriptor of an interrupt mailbox. Being used for signalling purposes only, the message contains only the header field and no data. There are several hardware levels of interrupt, with level 0 having the highest priority. In many cases interrupt service routines and ordinary processes in the kernel are not different the software system. Thus, some means must be provided to the operating system for establishing correspondence between the software priority and the hardware priority of a given process. This is essential, for instance to the dispatch routine, which must enable selected priority levels for hardware interrupts when creating an environment for a user process. For this reason, software priority levels of this particular implementation are divided in two groups. The first group is dedicated to interrupt-related processes. It contains distinct hardware levels of priority, a straightforward allocation of software levels to each hardware level . Processes wishing to service interrupts occurring at a given hardware level of priority must be assigned a software priority level in the corresponding range. In this way, a dispatched process is assured that while it is running, all lower and equal hardware levels of priority are disabled by the operating system. The second group includes software that are not associated with hardware priorities.

It is treated as being software and below all hardware levels of priority. This means that when a process belonging to that range of software priorities is running, all hardware levels of interrupt are enabled and an interrupt from any source may primped the running process.

Interrupt servicing in the kernel. The interrupt handler is entered whenever an interrupt request, that is, one explicitly enabled and of higher priority than that of the running process, is recognised by the hardware. In order to distinguish between hardware sources connected to different levels of interrupt, each level is vectored to its own entry point. After recording the identity of the particular hardware interrupt level, a branch to common processing of all interrupt requests is made. After saving on the stack the state of the process pre-empted by the current interrupt, the updated value of the hardware stack pointer is stored in the process descriptor of that process.

In the kernel, the hardware stack pointer register points to the private stack area of the running process. Since the hardware interrupt-processing sequence and interrupt handler both use the stack to save their respective portions of the pre-empted process's state, thus at the beginning of interrupt processing it is necessary to save the complete state of the affected processes on its own stack. Subsequent storing of the updated value of the program counter into the process descriptor of the pre-empted process completes the state-save operation. When the process in question is eventually resumed and its stack pointer restored, the state-restore sequence may be reduced to simply popping-off the saved registers in the reverse order of storing.

After saving the context of the pre-empted process the interrupt handler sets its status to pre-empted in order to inform dispatcher routine that a full context restore will be needed when the process in question is to be resumed. The interrupt handler then acknowledges the interrupt being processed to the programmable interrupt controller. This action normally results in a deasserting of the corresponding interrupt-request line to the CPU. But this does not acknowledge the interrupt to the originating I/O port, whose processing is left to the related interrupt-service routine. As a result, the kernel handles the processor part of interrupt logic and interrupt handlers handle the device specific portions of interrupt processing. This approach relieves the users from handling the processor and programmable interrupt controller interrupt flags and, in so doing, has the potential of increasing the system reliability by virtue of protecting the common interrupt logic of the system from users. At the same time, the kernel does not become involved in the semantics of the specific sources of interrupt, which otherwise could severely restrict its portability.

The kernel uses messages to signal arrivals of interrupts to the related interrupt handlers. In this case the asynchronous communication between interrupt occurrences and the related servicing processes is possible. In particular, a service process may be in some state other than Waiting-for-message when an interrupt message is sent to it. Due to buffering of messages in the kernel, the interrupt message in that case is deposited into the corresponding mailbox pending execution of the matching RECEIVE call by the interrupt-servicing process. When consecutive interrupts from the same source are close in time, a timing error may cause the servicing process to miss one or more interrupts. The kernel should at least alert the user process when interrupts are lost. This poses the problem of detecting missed interrupts in the kernel without getting involved in the details of their processing. In order to solve this problem, recall that the link field of each message is set to point to itself before the message is delivered to a requesting process. The primary reason for doing so is to assist in the detection of missed interrupts. If the link field of a given message is set to itself, it may be concluded that this message is not placed at any mailbox. When applied to interrupt messages in the kernel, by examining the link field of one, it can be determined whether it is placed at an interrupt mailbox awaiting delivery or not. Since the same message is sent by the system .for each arrival of an interrupt at a given hardware level, the only way for an interrupt message to have its link field set to itself is as a result of having been delivered to a requesting process. Conversely, if its link field has any other value, the interrupt message is awaiting delivery. Due to the relationship between interrupt mailbox descriptors and interrupt messages in the kernel, locating an interrupt message for inspection is not a problem.

The interrupt handler checks whether the interrupt message appended to the interrupt mailbox of the level being processed is awaiting delivery or not. If not, the interrupt message is sent to the corresponding mailbox by means of the SEND procedure. Otherwise, the message is still awaiting

delivery to signal some previous unprocessed interrupt, and a missed interrupt is detected. The interrupt message is not sent when missed interrupts are detected, because doing so does not serve any useful purpose and it might corrupt system .queues.

In normal processing, a service process is waiting at the interrupt mailbox, and it becomes the running process after receiving the message from SEND. However, if service is lagging behind and no interrupt-service process is suspended at the mailbox, SEND places the message and returns control to its caller, the interrupt handler. In that case, it schedules and dispatches the new running process. It is for this reason that SEND does not turn the interrupts back on when returning to the caller whose status field differs from ready.

Enabling Hardware Interrupt. Enabling of a hardware interrupt level modifies the system interrupt mask to allow recognition of a hardware interrupt level passed as a parameter. Enabling an enabled level has no effect. The system interrupt mask is typically a collection of dedicated bits, each of which indicates whether the corresponding hardware level of interrupt has been enabled by means of the enabling of a hardware interrupt call or not. The actual bit settings in system interrupt mask are machine-specific: They depend on both the number of distinct levels of interrupt in a given system and the ordering and setting of interrupt-enable/disable bits in control registers of the related hardware interrupt controller. The system interrupt mask is used by the dispatch routine while creating the interrupt-related part of the environment for the process being dispatched.

5.2.3.5. Process dispatching

Traffic-management is an important function of an operating-system kernel. Whenever a running process is pre-empted or suspended, the kernel must save the process's context for proper subsequent resumption. In general, the context of a process encompasses that part of the run-time environment which must be recreated by the operating system to ensure proper operation of the process when it is resumed after a period of inactivity. In particular, each system routine in the kernel whose execution results in a rescheduling of processes must save the context of the process which was running when the kernel was entered. When a new running process is selected by the scheduler, the kernel must restore the context of that process before actually surrendering control of the processor to it. This function of an operating system is often called process dispatching.

The process dispatcher is an internal kernel routine which transfers control to the new running process after restoring its context. In a sense, it is an exit routine from the kernel which should be invoked after each rescheduling in order to enforce the scheduling decision. The identity of a new running process must be made known to the process dispatcher, so that the related process descriptor can be found and used to restore the target context.

The dispatch directly manipulates certain machine-specific registers and uses special machine instructions, such as interrupt return. The dispatch procedure takes one parameter (process), a pointer to the process descriptor of the process to be dispatched. The first action of dispatch routine is to set the hardware stack and frame pointers to the values recorded in the process descriptor and thus restore the process's stack area. The next activity is to enable the interrupt levels that should be on when the process being dispatched is running. An interrupt level is recognisable in the kernel only when two conditions are satisfied simultaneously:

- the hardware priority of the request is higher than that of the running process, and
- the level at which the request is made has been explicitly enabled by means of the enabling interrupt call.

In order to enable the appropriate hardware levels of interrupt, dispatch routine uses the interrupt mask stored in the process descriptor. The value of this mask is computed by the CREATE procedure on the basis of the software priority of the related process. Its format is machine-specific, because the mask is supposed to be directly copiable to the appropriate register of the hardware interrupt controller. The interrupt mask is derived in such a way that only interrupt levels higher than

that of the running process may be enabled. In order to enable only a subset of those levels which is defined by preceding invocations of enabling interrupts, procedure dispatch combines the process-specific mask with system interrupt mask. The value of the system interrupt mask is applied directly to the appropriate programmable interrupt controller register. This mask, however, takes effect only when control is passed to the user's process, as interrupts: remain off throughout dispatch routine.

If the process being dispatched was pre-empted by an interrupt during its run, the dispatch routine restores its context by popping-off the contents of registers from the stack in the reverse order from their storing by interrupt handler. Although not different from other processes at the user level, interrupt handlers are unique in the sense of being invoked by the hardware interrupt processing sequence. The effects of this sequence must be undone by the specialised interrupt-return instruction IRET. With the user process's program counter on stack, dispatch routine passes control to a pre-empted process by means of the IRET instruction. In machines where IRET does not restore the value of the hardware interrupt flag, an enable-interrupt (EI) instruction should precede IRET.

When handling a process not pre-empted on its last run, the dispatch routine checks whether a message needs to be delivered. This is normally the case when the process being dispatched is exiting the Waiting-for-message state. If so, dispatch routine must retrieve the address of the message from the delay field of the related descriptor; where it is placed by the SEND procedure. After setting the link field of the message to point to itself, dispatch routine passes control to the user process as if it were returned by the RECEIVE routine. In general, this requires emulation of the return function in order to deliver the message destined for the process being dispatched.

With the process's program counter on stack,, a convenient way for dispatch routine to finally pass control is by means of the regular subroutine return instruction (RET). Since dispatch routine is always invoked with interrupts off, the last instruction prior to RET is customarily enabling of interrupts (EI), subject to masks in effect.

5.2.3.6. Access to hardware and to physical memory addresses

Access to hardware is often made possible by means of a special mechanism rather than by providing specific statements in the language. This generally implies some means for the vectoring of interrupts to user-written handler routines and the ability to manipulate certain machine registers, such as the processor status word, interrupt masks, and the stack pointer. Access to hardware is often provided by the use of specialised libraries whose functionality is not defined in the language but is usually informally suggested and often becomes a de facto standard.

Access to physical memory addresses usually refers to the ability to obtain the address of, or to form a pointer to, a static object in memory. Another common variety is assignment of an address to a variable. This feature is useful for accessing I/O ports from a high-level language. Access to memory words and physical addresses is often coupled with some sort of pointer arithmetic with relaxed type checking.

Controlled type-conversions refer to the ability to occasionally escape the strong type checking if it is otherwise enforced by the programming language. In other languages, a generic type-such as an unsigned integer or a word-is provided and made compatible with other types of the same length. Note that controlled type-conversion may substitute for the pointer arithmetic mentioned earlier. An uncontrolled type-conversion is often made possible by systems which support independent compilation. In such systems, one may simply call a procedure coded in another language, say, assembler, to equate two of different types. This approach is an uncontrolled one because the compiler is not alerted to potentially dangerous and often non-portable conversion of types. The portability of programs that access hardware and physical memory addresses is often restricted to a given type of processor or even to a specific hardware conflagration.

5.2.4. Invoking the Operating System

Software system must provide a way to invoke operating system services from user programs and to pass some parameters when needed by a specific operating system routine. In computer systems that provide some hardware support for protection between the operating system and user processes, an additional requirement is to change the level of privilege whenever crossing the operating system-user boundary. Three methods are used to invoke operating system services:

- procedure call
- system call
- software interrupt

5.2.4.1. Procedure call

A procedure call assumes that the operating system routines are coded as procedures. Users may invoke them and pass parameters to the operating system by means of the standard procedure call. This method is simple and usable on virtually any computer system, since all that it needs is hardware support for the procedure call/return mechanism. However, procedure calls are generally unsuitable for changing the level of privilege when transferring between a caller and a callee. An exception to this rule may be claimed by some virtual-memory systems, where the operating system is mapped into each user's virtual space. Such systems may use the procedure call mechanism for its efficiency, since they provide protection to common operating system routines through address-space isolation, but privileged access to system facilities may still be difficult to implement.

Another, no less serious problem with invoking the operating system via procedure calls is that of binding. Namely, in most program-development systems the operating system is already in place when user programs are being developed and run. When invoking the operating system, the question is when and how to bind user specified names of operating system services to the actual addresses of the corresponding operating system routines long since loaded in memory. A straightforward solution, employed in some primitive monitors and early microcomputer systems is to use direct instead of symbolic addressing and to specify the actual address of the desired operating system routine. A more convenient approach, used in some larger systems is to allow symbolic referencing of operating services by their prescribed names. Translation of those names into the entry points of the corresponding modules is performed with the aid of a system-supplied library with public declarations of the names of user-callable operating-system services. The address of each service can be determined and furnished into the operating system library by the linker or the linking-loader at the time when the executable version of the operating-system code is prepared. User programs that call on operating-system services directly must specify the operating system library as one of the input modules during the linking process in order to have the symbolic references to operating system resolved.

Both these schemes are sensitive to changes of operating-system service entry points, which are quite customary in different releases of operating systems, and even in different hardware configurations running under the same release and version of a given operating system. As a result, object-code portability is severely restricted, and programs may have to be relinked even in order to work on a given hardware installation when the operating system is upgraded to a new release. Such a mode of operation is obviously undesirable in distributed and network environments, since it may preclude migration of executable programs even among the homogeneous nodes running the same kind of operating system on the same type of the processor.

5.2.4.2. System call

A system call is usually a specialised hardware instruction dedicated to invoking the operating system. Common implementations of system call provide at least one operand, the identity of the desired operating system service. Namely, system call typically provides a common entry point to the operating system. The indicated service may then be invoked on the basis of the supplied parameter

by means of a jump or index table. Since it always transfers control to the operating system, the system call instruction may be allowed to switch to supervisor or some other privileged operating system mode. A context switch, if necessary, can also be performed at this point and perhaps be included in the microcode of the system call instruction. When returning control to a user process, the operating system can do so by means of some sort of (privileged) switch-mode instruction.

Another advantage of the system call instruction is that it alleviates the problem of knowing the addresses of the operating system routines at compile-time or link-time. Users can simply specify a number of the desired operating system service as a parameter to the system call instruction. The key is in the indirect transfer of control by means of the jump table, whereby system call performs the required run-time binding. Namely, when the operating system code is loaded in memory, the system call jump table is normally filled with the actual address of each numbered operating system routine. As long as the service numbers are kept identical between releases, object-code portability of user programs may be accomplished when the system call instruction is used.

5.2.4.3 Software interrupt

A software interrupt is a versatile instruction. A software interrupt instruction triggers the hardware interrupt-processing sequence. A software interrupt has the same effect as if an external interrupt request were asserted. Depending on the supporting architecture, this may include the context-save and transfer of control to the related interrupt-service routine. The software interrupt instruction is useful for invocation of operating-system services and the debugging of interrupt-related code.

When used to invoke the operating system, the software interrupt instruction may provide both context switch and vectoring directly to the related operating system routine. In addition, switching to a privileged mode of operation can also be performed at this time. Similar to the supervisor call instruction, the operating system or some other program can load the address of the corresponding service into each interrupt vector dedicated to an operating system service. The users are thus relieved from knowing anything but the number or the "interrupt level" of the corresponding operating system service. Due to the resulting run-time binding, the portability of object code can easily be accomplished when the software interrupt instruction is used to invoke the operating system. By designating a single interrupt vector as an entry point to the operating system, the software interrupt instruction can be used to emulate the supervisor call described earlier. However, the bottleneck of a single entry point can be alleviated by the software interrupt instruction by dedicating a separate vector to each individual operating-system service.

The software interrupt instruction can also be used to aid in the debugging of interrupt-related code, an area where standard debuggers are of little or no use and where generation and simulation of realistic test cases are difficult. Namely, the software interrupt instruction can be used to simulate an interrupt coming from a physical device and to trigger the entire event-recognition sequence by being issued from a test program. However; permissive use of this facility can be dangerous in a system in normal operation, and some safeguards must be in place to prevent potentially hazardous uses of the software instruction, such as triggering false interrupts at the level assigned to an active device.