REAL-TIME KERNEL

FUNDAMENTALS
- **Types of Kernel**
- **Kernel structures**
- **Kernel Implementation**

BASIC SERVICES of the KERNEL

FUNCTIONS of the KERNEL
- **Initialisation or System Booting**
- **Process Management**
- **Interprocess Communication and Synchronisation**
- **Interrupt Management**
- **Process dispatching**
- **Invoking the Kernel**

FUNDAMENTALS

- a real-time operating system has a kernel inside of it, where the kernel usually denotes the lowest layer of an operating system which is directly above the hardware

- a typical kernel may provide basic and additional services to other software layers of an operating system and to system users

basic services:

1 booting and start-up of the initial system configuration.

2. process management

3. interrupt management

4. interprocess communication and synchronisation

additional services:

1. memory management,

2. timing,

3. I/O management,

4. 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,

- in a complex system the kernel is typically enclosed by additional layers of an operating system

Types of Kernel

- there are three basic ways to organise a kernel:

1. monolithic unit in which each kernel primitive executes as an atomic action;

2. a collection of specialised processes that interact to implement kernel primitives; e.g. one may handle file I/O, and another, memory management

3. a concurrent program in which more than one user process may be executing a kernel primitive at the same time

single processor kernel

- any kernel contains data structures that represent processes and three basic kinds of routines:

- interrupt handlers,

- kernel primitives, and

- dispatcher,

- the kernel starts executing when an interrupt occurs next the processor automatically saves state information so that the interrupted process can be resumed

- then the processor enters an interrupt handler

- interrupts can be divided into two major categories:

1. external interrupts from I/O devices, and

2. internal interrupts (also called traps) triggered by the executing process

- to invoke a kernel primitive, a process causes an internal interrupt by executing a machine instruction for example a supervisor call (SVC) or trap,

- the process passes an argument with this instruction to indicate what primitive is to be executed and passes other arguments in registers

- the 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 dispatcher to select a process for execution and then to load its state

multiprocessor kernel

- a multiprocessor kernel provides an object-oriented environment for applications,

- this environment consists of objects such as processes, semaphores and services such as signal and test provided by the kernel,

- the multiprocessing kernel allows the real-time system software to actually run as multiple processes.

- this approach has several advantages:

- **small processes are easier to code, debug than is monolithic software,**
- **the 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,**

- it is relatively straightforward to extend the single processor kernel to a multiprocessor kernel,

- the main changes we need to make are

1. to store kernel procedures and data structures in shared memory,

2. to access the data structures with mutual exclusion when that is needed to avoid interference, and

3. to change scheduler and dispatcher to exploit the multiple processors,

- 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,

- multiprocessors require the following new aspects of operating system design:

- **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,**

distributed kernel

- in distributed systems the kernel is replicated by placing one copy on each machine and to have the different kernels communicate with each other using channels and network communication primitives,

- a process executing on a machine accesses the channel in the shared-memory kernel,

- the kernel on each machine contains descriptors for the channels stored on that machine and for the processes that execute on that machine,

- each kernel has local interrupt handlers for internal traps, timers and input-output devices where 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.

send operation:

- when a kernel primitive 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,**

send operation:

1. net_write acquires a buffer, formats the message, and stores it in the buffer,

2. if the receiver half of the network controller is free, an actual write is initiated,

3. otherwise the buffer is inserted on a queue of write requests,

4. when a write interrupt occurs, the associated interrupt handler frees the buffer containing the message that was just written,

5. if its write queue is not empty, the interrupt handler then initiates another network write,

receive operation:

1. when a message arrives at a kernel, the network_read interrupt handler is entered,

2. it first saves the state of the executing process,

3. then it allocates a new buffer for the next network input message,

4. finally the read handler unpacks the first field of the input message

to determine how to handle the message and then calls the appropriate kernel primitive,

Kernel structures

monolithic kernel

- 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

layered kernel

- 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,

- certain modules fall naturally into a hierarchical relationship, and this is exploited in system design,

microkernel

- those functions which are needed universally, by every component of a system, form a minimal kernel, sometimes called a lightweight kernel or microkernel,

- the basic notion of a microkernel consists of devolving as much functionality as possible into processes rather than the kernel itself;

for example:

- the Mach kernel leaves device drivers in the kernel, and place higher-level services (such as file systems) outside;

- the QNX kernel moves device drivers outside of the kernel,

- a small kernel has the advantage of being easier to engineer and maintain,

- 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.

- 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 aim is to implement efficient mechanisms in the kernel and as much as is reasonable at user level,

- 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,

- requirements of the applications that run above a minimal kernel can be expressed by indicating the relative priorities of these user-level processes,

- 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,**
- **services which now run above the kernel are easier to engineer, tune, update and maintain,**
- **time spent executing the kernel can be bounded,**

- a potential disadvantage of microkernels:is

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

- it should be possible to build a conventional operating system interface above a microkernel and run existing software above

for example:

- UNIX emulation can take UNIX system calls and translate them into calls on the microkernel services,

adaptable kernel

the adaptable kernel architecture is 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,

Kernel Implementation

traditional assembly language implementation:

- produces a monolithic object that must be loaded into the target system in its entirety.

modern implementation 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,

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:

- 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. 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).

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),

Initialisation

- whenever a computer system is started, the kernel 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, a kernel 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,

- a kernel should provide mechanisms and implementation of policies such as those governing resource allocation,

- some basic scheduling must be done by the kernel in order to control the execution of processes,

for example:

- 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, kernel 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 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 in the related process descriptor, for subsequent restoration when the process is scheduled to run again,

- in a multi-process 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 kernel by means of system lists of process descriptors.

- 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 kernel. Once started, the timer generates interrupts in approximately regular intervals.

Communication and synchronisation.

- the interprocess synchronisation and communication mechanisms define the basic features and properties of real-time systems,

- messages, which may serve both interprocess synchronisation and communication represent the best solution,

- 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 by dedicating a mailbox to each potential hardware source of interrupt,

- 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,

FUNCTIONS of the KERNEL

Initialisation or System Start-up

booting or bootstrapping:

- whenever a computer system is "cold-started," after being powered or a system crash, at least a portion of the kernel must be in the main memory and given control of the processor

- typically, the hardware initially transfers control to a known address where a bootstrap loader is placed in the ROM,

- the bootstrap loader brings the rest of the system gradually to main memory, for instance, from the secondary storage

- in memory-based systems the starting routine can simply transfer control to the kernel, which itself may be residing in ROM,

- if the kernel is specified as a memory-based system, the code of processes to be run under it, is in the 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,

- in the kernel-based systems control must be passed to a section of code, whenever the system is started (booted), that turns interrupts off and initialises hardware and I/O ports by their service drivers,

- with hardware prepared for normal operation, the kernel initialises its data structures and global variables before passing control to the first user process. These include 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,

- 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 the initialisation table,

- in many applications, however, there may not be a single-user process which should be present and therefore activated automatically by the kernel in every installation,

- thus, it is required to create a known process as user-definable but system-initiated, whenever the system is started, this process is always the first user process to receive control from the kernel,

- 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,

- after finishing initialisation, the interrupt system is enabled for the level of priority of the invoked process,

- finally, the dispatcher is invoked to start one of the created processes and to surrender control of the processor to it,

- after initialisation, the control of the processor is distributed between active processes and the kernel in response to and in accordance with the changes of system state and the resource-allocation policies,

Process Management

process descriptor = PD

- PD contains a process's attributes and other information that the kernel needs about a particular process,

- systems programmers may provide the initial values of process attributes by means of a special data structure, the static process descriptor (SPD),

- a SPD 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,

- SPD 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 its PD,

- SPDs 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 SPD formed, a process can be created by means of the CREATE system call,

- PD 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,

- the 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 kernel from the stack accessible to user processes,

- 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,

- 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 process's stack pointer may be stored in its PD

System lists:

the following lists are to be provided in the kernel by data structures:

- **existent list,**
- **ready list,**
- **suspended list,**
- **waiting-for-message (process descriptors, one list per mailbox)**
- **waiting messages (messages, one list per mailbox)**
- **mailboxes (mailbox descriptors).**

- the process may subsequently move between process states,

- whenever a process changes its state, the corresponding kernel routines record the resulting state of the system by moving the related PD between the corresponding system lists,

- a PD must provide link fields for the links required by various

system lists.

- to handle these lists a single multipurpose link field in a PD is used. Each PD has one link field per each type of list,

- as a result, when a PD 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 PD from one place in memory into another.

Existing list

- the existing list contains process descriptors of all active processes,

- w**hile 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,

Ready list

- it contains only the PDs 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,

- 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.**

- 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,

- 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 PD i.e. all that is needed is to update the corresponding PD's status field,

- in this case a new process can be scheduled by searching all PDs 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 PD or to insert a PD 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 PD in the Ready list always belongs to the highest-priority ready process, and scheduling is reduced to setting a system pointer to it,

Management of the ready list

- whenever a process needs to be "made ready", it is inserted by a kernel procedure into the ready list,

- this procedure inserts a PD, passed as a parameter, into the ready list such a way that preserves its priority ordering, The parameter passed to this procedure is a pointer to the PD to be processed,

- in particular, the new PD 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,

- to find a proper place in the ready list where the submitted PD is to be inserted, the procedure searches the ready list for the first PD of lower priority. If one is found, the new PD 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,

problem:

- there is the following problem in every multiprocess system is what the kernel should do when there is nothing to do,

- 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 pre-emption,

- while in a high-priority wait loop, the kernel may never be able to detect the lower-priority event needed to get the user processes back in motion,

solution:

- special „null" process that is created by the system as a part of the start-up sequence,

- 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,

- existence of the "null" process makes the execution of scheduler routine faster, since it does not have to test for emptiness of the ready list,

- the „null" process is designed to never terminate and to always be ready,

- as a result, the 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,

Delayed list

- the DELAY system call inserts entries into this list, and the check delay list routine searches the delay list and removes qualified entries,

management of the delayed list

1. unsorted delay list:

- 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.

2. sorted delay list:

- insertions into a sorted delay 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

problem:

- decrementing of remaining delays of all processes in the delay list,

solution:

- to make delays relative to each other, rather than absolute in terms of remaining time units i.e. delays are 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 delay 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 PDs of processes beyond the one at the head of the delay 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.

Process creation & termination

- creation of a process entails initial formation of the PD, typically followed by moving the process into the existing and later the ready state,

- when a process does not exist any more, its PD is normally returned to the pool of free PDs,

- the CREATE system call associates a PD with the newly created process, enters the initial values of the process attributes, such as process priority, and enters the PD into existing lists,

- the CREATE procedure takes one parameter, the address of the static PD of a process to be created, and starts by forming a PD in the area pointed to by the static PD,

- to create a stack; first, the top of the stack must be calculated and placed into the stack field of the PD. 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,

- an important task of the CREATE system call is the preparation of the newly formed PD 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. To achieve it the CREATE system call places 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 PD, onto its private stack. In addition, the stack field of the PD 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,

- the next activity of CREATE system call is to calculate the process's interrupt mask on the basis of its software priority indicated in the static PD,

- 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

- 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 PD field, the interrupt mask may be calculated by dispatch routine at run-time.

- 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,

Process delay

- an important process-management function of the kernel is delaying of a process for a specified period of time to preform it, the DELAY system call delays a process for a specified time,

- a process wishing to delay itself invokes the DELAY system call and takes two parameters:

1. the ID of the process to be delayed, and

2. the delay expressed in terms of system time units.

- 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 delay list in the presence of concurrent updates.

- 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,

- the kernel services such requests by means of two separate but cooperating activities:

1. delay subroutine

- 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 PD,

- 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 delay 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 delay list by modifying the forward link of its predecessor and the back link of its successor, if any

- after updating its status and recording its calculated relative delay, ticks, the stack pointer of the delayed process is saved in its PD 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.

2. delay list check subroutine

- whenever an interrupt from the interval timer is processed, the kernel decrements the number of remaining intervals in the corresponding PDs of all the processes in the delay list,

- the delay list check handler 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 delay list,

- once an interrupt occurs, delay list check 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,

- if the updated delay becomes 0, the top process and its potential successors with identical delay are removed from the delay list and placed into the ready list,

- in general, when accurate timing is desired, the delay list check handler process or its equivalent must have higher priority than any process which delays itself,

- the delay list check 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 check handler in the kernel-based systems, and the high priority of delay list check handler process provides for accurate tick keeping, subject to the accuracy and stability of the hardware clock,

- if more processes are left, the back pointer of the new first process is set to the list head,

- in general, when delay list check handler adds one or more entries to the ready list the scheduler should be invoked to select the next process to run.

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

- a mailbox serves as a meeting place of processes and messages, and kernels 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 MBD 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,

- in order to serve interprocess message communication, a mailbox must first be created and initialised by means of the corresponding system call in the kernel at the message creation time, and a separate MBD is usually dedicated to each active mailbox to support keeping track of the mailbox,

- typically, a sender or the system-initialisation module performs this task,

- it normally results in formation of the related MBD capable of queuing messages or processes. When a MBD is created, both queues are empty and their headers must be initialised to reflect that fact,

- when a mailbox is created, all queues are assumed to be empty and their respective pointers are set to 0. Since memory allocation is not performed by the kernel, a process wishing to create a mailbox must

provide a record of the MBD type and invoke a routine to initialise the MBD,

- procedure that creates the mailbox takes one parameter, the address of the MBD, this procedure initialises all pointers of the mailbox descriptor to 0 and places the MBD 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 MBD is in the process of being inserted into the mailboxes list , interrupts are turned off to ensure indivisibility of the operation,

- a MBD in the kernel must be structured in such a way that either messages (resulting from the execution of unmatched SENDs) or PDs of suspended processes (resulting from the execution of unmatched RECEIVEs) can be queued at it,

- FIFO and priority queues for waiting processes and messages are widely used because FIFO is simple to implement and it ensures that, with the proper supply of messages, waiting processes cannot be subjected to indefinite postponement,

- there should be a list of all mailbox descriptors in the system in order to support message management,

- 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 MBD is in active use,

- 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.

- 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.

- MBDs and the mailbox list can be used in kernels to check and declare affiliation of processes with active mailboxes by means of system calls,

- a process may detach itself from a mailbox that it no longer uses, deletion of an existing mailbox is not offered by the kernel but which may be desirable in the operating system,

- the kernel 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 kernel immediately

SEND message operation

- SEND system call needs two parameters:

1. the identity of the mailbox (i.e. the MBD) where the message is to be delivered and

2. the message itself,

- 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

1. placing data in the data field, if any,

2. setting the length field in the header

3. setting the message type

- 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,

- when SEND is invoked by the running process, either processes or messages may be found waiting at the designated mailbox descriptor,

- 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 one or more processes are waiting,

- when no processes are waiting - unmatched RECEIVE -, the message cannot be delivered immediately, and SEND places it at the mailbox returning control to the caller, the running process that 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 messages; a portion of each MBD may be used as a header of the queue of messages awaiting delivery at its related mailbox,

- the scheduler should be invoked to determine which process should receive control after the completion of SEND,

- 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 PD 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 dispatcher 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,

- when the message is delivered to a waiting process, there are only two viable candidates from which to select the next running process:

1. the caller of SEND (previously running process), and

2. 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,

- 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 MBD.,

- MBD 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,

- 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 delivery 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 dispatcher 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,

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, to the running process, which is not pre-empted,

- if no message can be delivered, the running process is suspended at the MBD, the calling process enters the waiting-for-message state and remains suspended until arrival of a message at the designated mailbox, and a new running process is scheduled and dispatched.

- 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,

- 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,

- the waiting-for-message state of processes in the kernel can be handled by a layered version of the suspended list, whereby PDs 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,

- RECEIVE takes two parameters:

1. pointer to the mailbox where the message is to be received, and

2. 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 first 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,

- 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,

Interrupt Management

- interrupt service in the kernel relies on the message-exchange mechanism,

- 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 kernel 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,

- there are 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 the priority can be entered into the process descriptor at process creation time,

- the kernel should differentiate between hardware and software priority levels.

1. 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.

2. 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.

- preliminary processing of all interrupts in kernel-base systems is performed within the kernel,

- consequently, all interrupts are vectored to the kernel, i.e. to the interrupt handler,

- 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 messages between the kernel and the interrupt handler,

- 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, software must be employed to "demultiplex" requests for service by identifying the specific originating device.

- there is a special system message 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,

- some means must be provided to the kernel for establishing correspondence between the software priority and the hardware priority of a given process,

- for this reason, software priority levels of this particular

implementation are divided in two groups,

1. 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 kernel.

2. group includes software that are not associated with hardware priorities,

- it is treated as being software and below all hardware levels of priority,

Interrupt service 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,

- 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 PD 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 kernel uses messages to signal arrivals of interrupts to the related interrupt handlers,

- a 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,

- 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,

- 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,

Process dispatching

- when a new running process is selected by the scheduler, the kernel must restore the context of the new running process before actually surrendering control of the processor to it,

- the dispatcher transfers control to the new running process after restoring its context,

- dispatcher 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 PD can be found and used to restore the target context,

- the dispatcher takes one parameter, a pointer to the PD of the process to be dispatched.

- the first action of dispatch routine is to set the hardware stack and pointers to the values recorded in the PD 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,

- 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,

- the interrupt mask is derived in such a way that only interrupt levels higher than that of the running process may be enabled,

- 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,

- when handling a process not pre-empted on its last run, the dispatcher 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, dispatcher 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, dispatcher routine passes control to the user process as if it were returned by the RECEIVE routine,

Invoking the Kernel

- software system must provide a way to invoke kernel services from user programs and to pass some parameters when needed by a specific kernel routine,

- in computer systems that provide some hardware support for protection between the kernel 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:

1. procedure call

2. system call

3. software interrupt

Procedure call

- users may invoke them and pass parameters to the kernel by means of the standard procedure call,

- this method is simple on 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,

- there is also problem with invoking the kernel via procedure calls is the binding. Namely, in most program-development systems the kernel 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,

System call

- a system call is usually a special hardware instruction dedicated to invoking the kernel,

- a system call typically provides a common entry point to the operating system,

- the requested 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 kernel, the system call instruction may be allowed to switch to supervisor or some other privileged kernel mode,

- when returning control to a user process, the kernel can do so by means of some sort of (privileged) switch-mode instruction.

- an advantage of the system call instruction is that there is no need for knowing the addresses of the kernel routines at compile-time or link-time,

- when a system call performs the required run-time binding. Namely, when the kernel code is loaded in memory, the system call jump table is normally filled with the actual address of each numbered kernel routine

Software interrupt

- 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 kernel services and the debugging of interrupt-related code,

- when used to invoke the kernel, the software interrupt instruction may provide both context switch and vectoring directly to the related

kernel routine,

- in addition, switching to a privileged mode of operation can also be performed at this time,

- similar to the supervisor call instruction, the kernel or some other program can load the address of the corresponding service into each interrupt vector dedicated to an 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 kernel, 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 kernel 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,