Chapter 7

Embedded/Real-Time Operating System Concepts

CHAPTER OBJECTIVES

After reading this chapter, you will be able to:

- Understand the architecture of the kernel of an operating system
- Learn the details of task scheduling algorithms
- Gain knowledge of inter-task communication
- Understand the details of kernel objects such as tasks, task scheduler, Interrupt Service Routines, semaphores, mutexes, mailboxes, message queues, event registers, pipes, signals and timers.

This chapter gives the important concepts of embedded/real-time operating systems. We will discuss the details of operating system kernel. The various kernel objects and the operations on these objects are presented. The task scheduling algorithms are explained. Though the implementation details vary from operating system to operating system, in this chapter we will focus on the concepts. A thorough conceptual understanding will enable you to work on any operating system with ease.

7.1 Architecture of the Kernel

In Chapter 2, the software architecture of an embedded system is presented. The embedded software consists of the operating system and the application software. The services provided by the operating system are accessed through the Application Programming Interface (API) to develop application software. The API is a set of function calls using which you can access the various kernel objects and the services provided by the kernel.

Let us now dissect the kernel and look into its details.

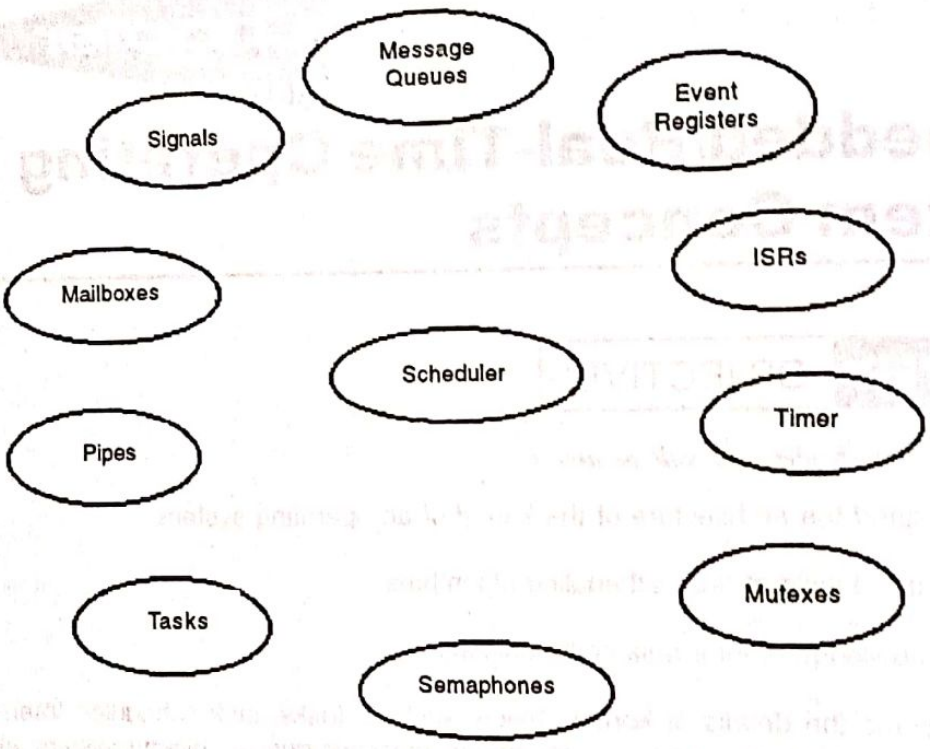


Fig. 7.1: Kernel Objects

The various kernel objects, as shown in Fig. 7.1, are:

- Tasks
- Task scheduler
- Interrupt Service Routines
- Semaphores
- Mutexes
- Mailboxes
- Message queues
- Pipes
- Event Registers
- Signals
- Timers

In Brief...

The various kernel objects are: tasks, task scheduler, Interrupt Service Routines, semaphores, mutexes, mailboxes, message queues, event registers, pipes, signals and timers.

We will discuss the details of these objects in the following sections. The API of the operating system gives the function calls to manage these objects. While discussing the kernel objects, the function calls for managing the objects are also given. The exact syntax of each function call may differ from operating system to operating system. The programming aspects using the API are discussed in Chapters 11 and 12.

Notes...

The kernel provides various services through operations on the kernel objects. These services are: memory management, device management, interrupt handling and time management.

7.2 Tasks and Task Scheduler

The embedded software consists of a number of tasks. These tasks include the operating system tasks as well as the application-specific tasks. Each task in an embedded system is implemented as an infinite loop. The task object consists of its name, a unique ID, a priority, a stack and a Task Control Block that contains all the information related to the task.

In addition to the tasks required for the application software, the kernel has its own system tasks with priorities. These tasks are:

- Startup task, which is executed when the operating system starts
- Exception handling task to handle the exceptions
- Logging task to log the various system messages
- Idle task, which will have the lowest priority and will run when there is no other task to run. This task ensures that the CPU is not idle.

Since only one CPU has to handle multiple tasks, the tasks have to share the CPU time in a disciplined manner so that one task does not get lot of time while others are waiting forever. Therefore, each task has to be assigned a priority; and a mechanism for deciding which task will get CPU time next has to be worked out. This is known as task scheduling. The object that does task scheduling is the task scheduler. In addition to the CPU time, the tasks have to share the system resources such as CPU registers, external memory and input/output devices. Another important requirement is that one task should not corrupt the data of another task. While scheduling the tasks, a number of issues, as described below, need to be kept in mind:

- Many tasks may make calls to a function. A function that can be used by more than one task without data corruption is called reentrant function. If data is corrupted when more than one task calls the function, such a function is called non-reentrant function. If you use global variables, then the function is non-reentrant. Consider the following code segment:

```
int x;
void findsq( int x)
{
return (x * x);
}
```

This function is non-reentrant because x is a global variable. It can be made reentrant by declaring the variable as local.

When you are running a task, some portion of the code should not be interrupted. Such code is called critical section of the code. Interrupts are disabled before the start of the execution of the code's critical section and enabled after the execution is completed. In the code snippet given above, the

As shown in Fig. 7.2, a task can be in one of the following states:

- Running
- Waiting
- Ready-to-Run

Running state: A task is said to be in running state if it is being executed by the CPU.

Waiting state: A task is said to be in waiting state if it is waiting for another event to occur. For instance, a task may be waiting to get some data from serial port. Even if the CPU is free, the task which is in waiting state cannot be executed till the external event occurs.

Ready-to-Run state: A task is said to be in Ready-to-Run state if it is waiting in a queue for the CPU time.

The arrows in Fig. 7.2 depict how a task can move from one state to another. A task which is in waiting state can move to the Ready-to-Run state after the external event occurs. Note that it cannot move directly to the Running state. A task which is in Ready-to-Run state can move to the Running state. A task in the Running state can move to the Waiting state if it has to wait for an external event to occur; or it can move to the Ready-to-run state if its job is not completed, but the CPU has to run another task.

A task which is presently running may be interrupted to run an Interrupt Service Routine (ISR) for a short time. After the ISR is executed, the CPU continues to execute the task which was interrupted or the highest priority task which is Ready-to-Run. The ISR can do the job of changing the state of a task. For instance, a high priority task which was in waiting state has to move to Ready-to-Run state. This operation can be handled by the ISR.

In Brief...

A task can be in one of the three states: (a) Running; (b) Ready-to-Run; and (c) Waiting.

Task stack: Every task will have a stack that stores the local variables, function parameters, return addresses and CPU registers during an interrupt. At the time of creating a task, the size of the stack has to be specified or a default stack size has to be used.

7.2.2 Context Switching

The CPU has to execute one task for some time and then execute another task which is in the queue. Consider a situation when the CPU is executing a low priority task, during which time the CPU registers contain the data corresponding to this task. The CPU now has to take out the low priority task (in other words, "preempt the low priority task") and execute the high priority task. In this case, the contents of the CPU registers have to be saved before executing the high priority task. The state of the CPU registers when a task is to be pre-empted is called the context. Saving the contents of the CPU registers and then loading the new task is called context switching.

In Brief...

The state of the CPU registers when a task has to be preempted is called the context. Saving the contents of the CPU registers and loading the new task parameters is called context switching.

7.2.3 Scheduling Algorithms

How does the kernel decide which task has to run? Various scheduling algorithms have been developed to tackle this problem. Depending on the requirement of the embedded system, the scheduling algorithm needs to be chosen. In this section, we will review the following scheduling algorithms:

- First In First Out
- Round-robin algorithm
- Round-robin with priority
- Shortest job first
- Non-preemptive multitasking
- Preemptive multitasking

First In First Out (FIFO)

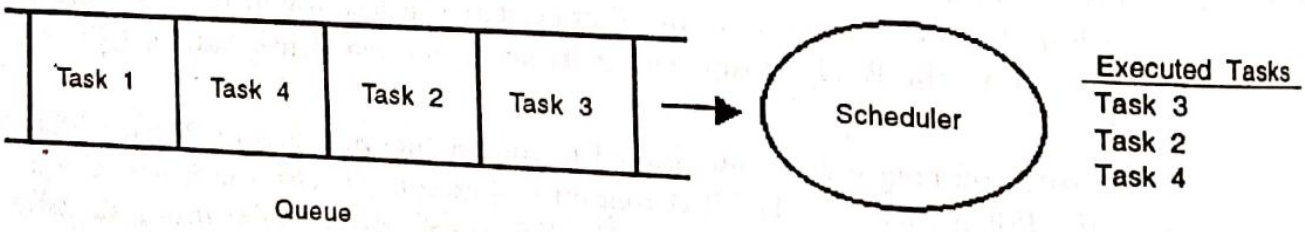


Fig. 7.3 Scheduling Algorithm: FIFO

In First In First Out scheduling algorithm, the tasks which are Ready-to-Run are kept in a queue and the CPU serves the tasks on first-come-first served basis. This scheduling algorithm, shown in Fig. 7.3, is very simple to implement, but not well suited for most applications because it is difficult to estimate the amount of time a task has to wait for being executed. However, this is a good algorithm for an embedded system has to perform few small tasks all with small execution times. If there is no time criticality and the number of tasks is small, this algorithm can be implemented.

Round-robin Algorithm

In the round-robin algorithm, the kernel allocates a certain amount of time for each task waiting in the queue. The time slice allocated to each task is called quantum. As shown in Fig. 7.4, if three tasks 1, 2 and 3 are waiting in the queue, the CPU first executes task1 then task2 then task3 and then again task1.

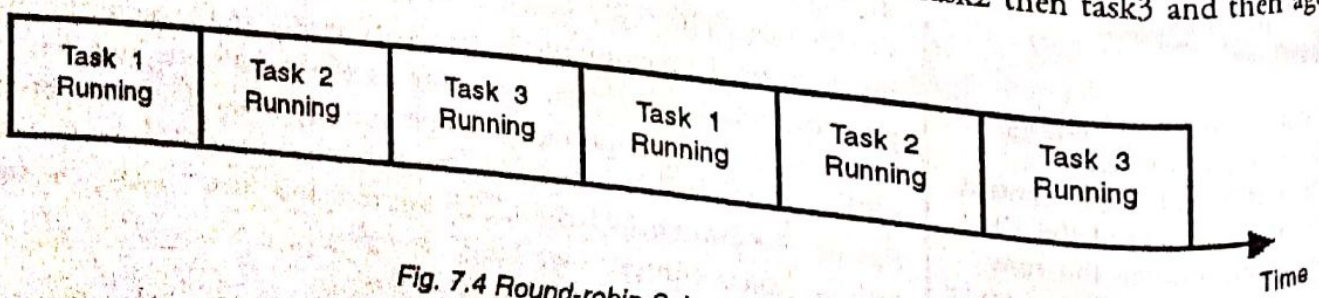


Fig. 7.4 Round-robin Scheduling Algorithm

In Brief...

In round-robin algorithm, each task waiting in the queue is given a fixed time slice. The kernel gives control to the next task if the current task has completed its work within the time slice or if the current task has completed its allocated time.

The kernel gives control to the next task if

(a) The current task has completed its work within the time slice

(b) The current task has no work to do

(c) The current task has completed its allocated time slice

This algorithm is very simple to implement, but note that there are no priorities for any task. All tasks are considered of equal importance. If time-critical operations are not involved then this algorithm will be sufficient. Digital multimeters and microwave ovens use this scheduling algorithm.

Round-robin with Priority

The round-robin algorithm can be slightly modified by assigning priority levels to some or all the tasks. A high priority task can interrupt the CPU so that it can be executed. This scheduling algorithm can meet the desired response time for a high priority task. For example, in a bar code scanner, high priority is assigned to the scanning operation. The CPU can execute this task by suspending the task that displays the item/price value. Soft real-time systems can use this algorithm.

Shortest-Job First

In Brief...

In shortest-job first task scheduling algorithm, the task that will take minimum time to be executed will be given priority. This approach satisfies the maximum number of tasks, but some tasks may have to wait forever.

If the time for each task can be estimated beforehand, the CPU can execute the task which takes the least amount of time. Effectively, this is like a priority assignment, the priority being decided by the amount of time—the higher the execution time, the lesser the priority. The advantage of this scheduling algorithm is that a high number of tasks will be executed, but the task with the highest amount of time will have to wait, perhaps forever!

The task scheduler is the heart of the operating system. It is the task scheduling algorithm which decides whether the necessary time constraints can be met or not. If the embedded system is a hard-real time system, none of the above scheduling algorithms can be used at all.

The kernels used in embedded systems can implement priority-based multi-tasking scheduling algorithms of two types:

- Non-preemptive multi-tasking
- Preemptive multitasking

Non-preemptive Multi-tasking

Assume that you are making a telephone call at a public call office. You need to make many calls, but you see another person waiting. You may make one call, ask the other person to finish his call, and then you can make your next call. This is non-preemptive multitasking, also known as cooperative multitasking—you are cooperating with the others in the queue.

In non-preemptive multitasking, the tasks cooperate with each other to get their share of the CPU time. Hence, each task has to release the CPU and give control to another task on its own. Each task

7.2.3 Scheduling Algorithms

How does the kernel decide which task has to run? Various scheduling algorithms have been developed to tackle this problem. Depending on the requirement of the embedded system, the scheduling algorithm needs to be chosen. In this section, we will review the following scheduling algorithms:

- First In First Out
- Round-robin algorithm
- Round-robin with priority
- Shortest job first
- Non-preemptive multitasking
- Preemptive multitasking

First In First Out (FIFO)

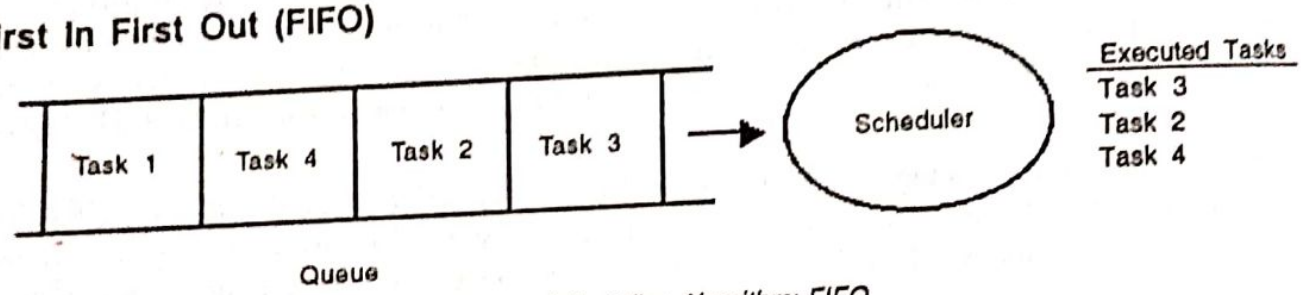


Fig. 7.3 Scheduling Algorithm: FIFO

In First In First Out scheduling algorithm, the tasks which are Ready-to-Run are kept in a queue and the CPU serves the tasks on first-come-first served basis. This scheduling algorithm, shown in Fig. 7.3, is very simple to implement, but not well suited for most applications because it is difficult to estimate the amount of time a task has to wait for being executed. However, this is a good algorithm for an embedded system has to perform few small tasks all with small execution times. If there is no time criticality and the number of tasks is small, this algorithm can be implemented.

Round-robin Algorithm

In the round-robin algorithm, the kernel allocates a certain amount of time for each task waiting in the queue. The time slice allocated to each task is called quantum. As shown in Fig. 7.4, if three tasks 1, 2 and 3 are waiting in the queue, the CPU first executes task1 then task2 then task3 and then again task1.

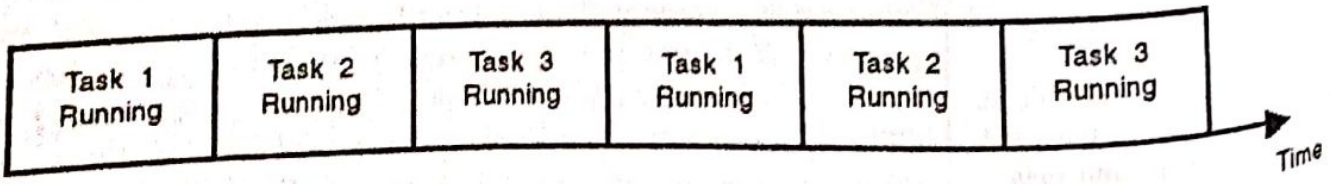


Fig. 7.4 Round-robin Scheduling Algorithm

Consider two tasks—one low priority task and one high priority task. To start with, the low priority task is running as the high priority task is in waiting state, waiting for an external event to occur. After some time, the external event occurrs and the high priority task now move to Ready-to-Run state. Through an interrupt, the ISR is executed to move the high priority task from Waiting state to Ready-to-Run state. Then another the ISR is executed to put the high priority task in the Running state. Again after some time, the high priority task may release the CPU and then the low priority task is executed.

In Brief...

In preemptive multitasking, the highest priority task is always executed by the CPU, by preempting the lower priority task. All real-time operating systems implement this scheduling algorithm.

If there are a number of tasks in the Ready-to-Run state, the task with the highest priority is always given the first chance by the task scheduler.

The main attraction of this scheme is that the execution time of the highest priority task can be calculated and hence the kernel is said to be deterministic. Most of the commercial embedded operating systems use preemptive multitasking to meet real time requirements.

Notes...

The module within the scheduler that performs context switching is called dispatcher. If ISR makes a system call, the dispatcher is bypassed.

7.2.4 Rate Monotonic Analysis

Priority assignment to a task can be static or dynamic. In static priority assignment, a task will be assigned a priority at the time of creating the task and it remains the same. In dynamic priority assignment, the priority of the task can be changed during execution time. While designing your application, it is not very easy to assign priorities to tasks. A good starting point is the Rate Monotonic Analysis used for assigning priorities.

While designing an embedded system, you need to make a list of all the tasks. And, then you need to assign priority to each task. Subjectively, we know that some tasks should have high priority, but nothing beyond! Rate Monotonic Analysis (RMA) provides an answer. It may not be accurate for any application, but it is a good starting point.

RMA makes the following assumptions:

(a) Highest priority task will run first, i.e. the priority-based preemptive multitasking is the scheduling algorithm.

(b) All the tasks run at regular intervals i.e. the tasks are periodic.

(c) Tasks do not synchronize with each other, i.e. they do not share resources or share data (which is not a valid assumption).

In RMA, the priority is proportional to the frequency of execution. If (i)th task has an execution period of Ti, and Ei is its execution time, Ei/Ti gives the percentage of the CPU time required for execution of (i)th task. In RMA, the "schedulability test" indicates how much CPU time is actually utilized by the tasks. If n is the total number of tasks, the equation for schedulabiity test is given by

$$\sum (Ei/Ti) \leq U(n) = n(2^{1/n} - 1)$$

U(n) is called the utilization factor. The value of number of tasks (n) and the utilization factor for different values of n is given in Table 7.1. If the number of tasks is infinity, then about 70% of the CPU time is utilized.

n	U(n)
1	1.000
2	0.828
3	0.779
4	0.756
Infinity	0.693

Table 7.1: Rate Monotonic Scheduling: Percentage of CPU time utilized

In Brief...

Rate Monotonic Analysis is used to calculate the percentage of CPU time utilized by the tasks and to assign priorities to tasks. Priorities are proportional to the frequency of execution. Note that the assumptions made in RMA are not always valid.

When you are designing an embedded system, it will be a good idea to start with RMA. List all the tasks, estimate the execution period and execution time, and then calculate the value of U(n). As a thumb rule, ensure that the CPU utilization is not above 70%. Assign priorities based on execution frequency.

7.2.5 Task Management Function Calls

The various function calls provided by the operating system API for task management are:

- Create a task
- Delete a task
- Suspend a task
- Resume a task
- Change priority of a task
- Query a task

7.3 Interrupt Service Routines

Interrupt is a hardware signal that informs the CPU that an important event has occurred. When interrupt occurs, CPU saves its context (contents of the registers) and jumps to the ISR. After ISR processes the event, the CPU returns to the interrupted task in a non-preemptive kernel. In the case of preemptive kernel, highest priority task gets executed.

In real-time operating systems, the interrupt latency, interrupt response time and the interrupt recovery time are very important.

Interrupt Latency: The maximum time for which interrupts are disabled + time to start the execution of the first instruction in the ISR is called interrupt latency.

Interrupt Response Time: Time between receipt of interrupt signal and starting the code that handles the interrupt is called interrupt response time.

In a preemptive kernel, response time = interrupt latency + time to save CPU registers context.

Notes...

System's worst-case interrupt response time has to be considered while evaluating the performance of an operating system/embedded software.

Interrupt Recovery Time: Time required for CPU to return to the interrupted code/highest priority task is called interrupt recovery time.

In non-preemptive kernel, interrupt recovery time = time to restore the CPU context + time to execute the return instruction from the interrupted instruction.

In preemptive kernel, interrupt recovery time = time to check whether a high priority task is ready + time to restore CPU context of the highest priority task + time to execute the return instruction from the interrupt instruction.

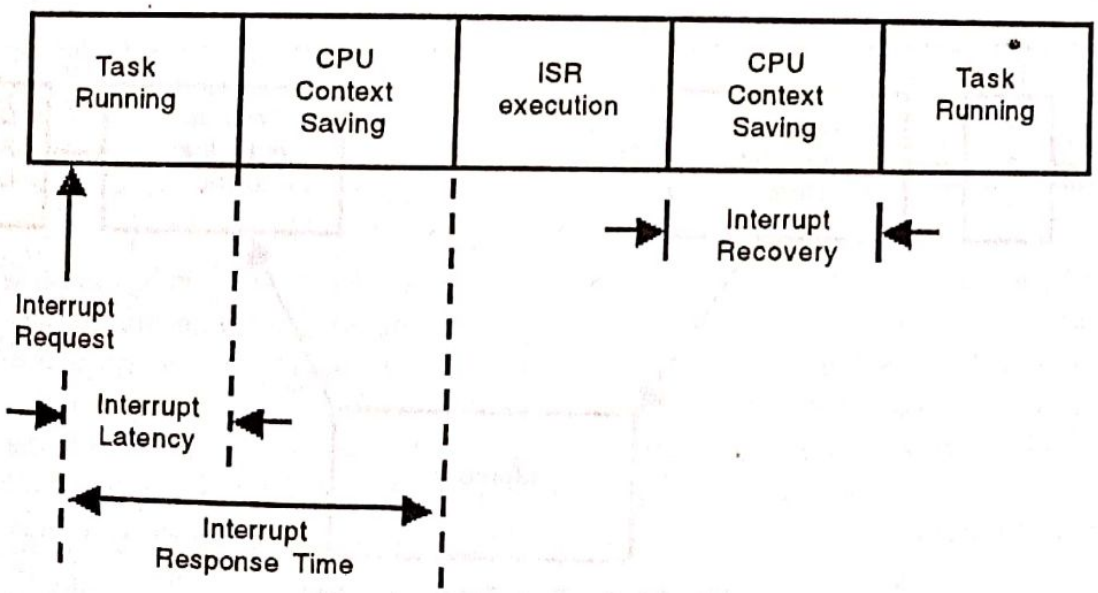


Fig. 7.7 Interrupt Latency, Interrupt Response Time and Interrupt Recovery Time

The interrupt latency, interrupt response time and interrupt recovery time are shown in Fig. 7.7.

7.4 Semaphores

When multiple tasks are running, two or more tasks may need to share the same resource. As shown in Fig. 7.8(a), consider a case where two tasks want to write to a display. Assume that task1 wants to display the message "temperature is 50" and task2 has to display the message "humidity is 40%". The display is a shared resource and if there is no synchronization between the tasks, then a garbled message is displayed such as "temphumieratiditurey is is 50 40%". To access a shared resource, there should be a mechanism so that there is discipline. This is known as resource synchronization.

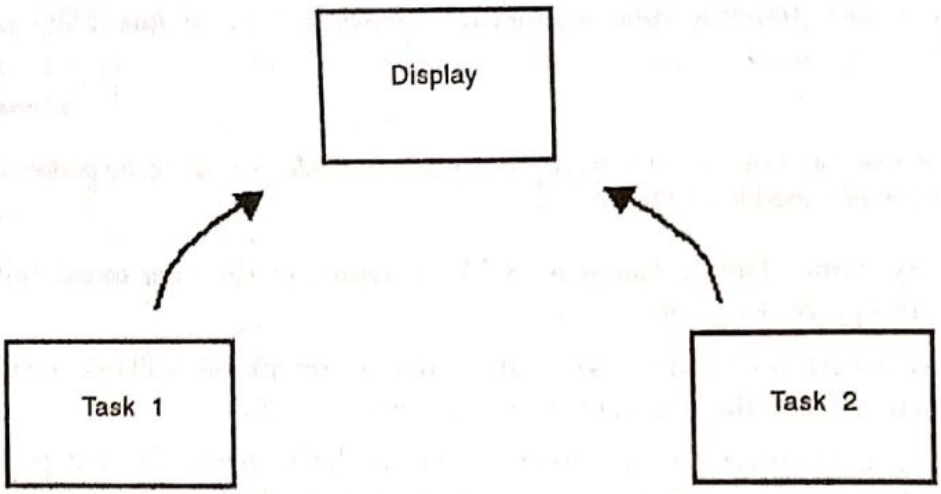


Fig. 7.8 (a): Resource Synchronization

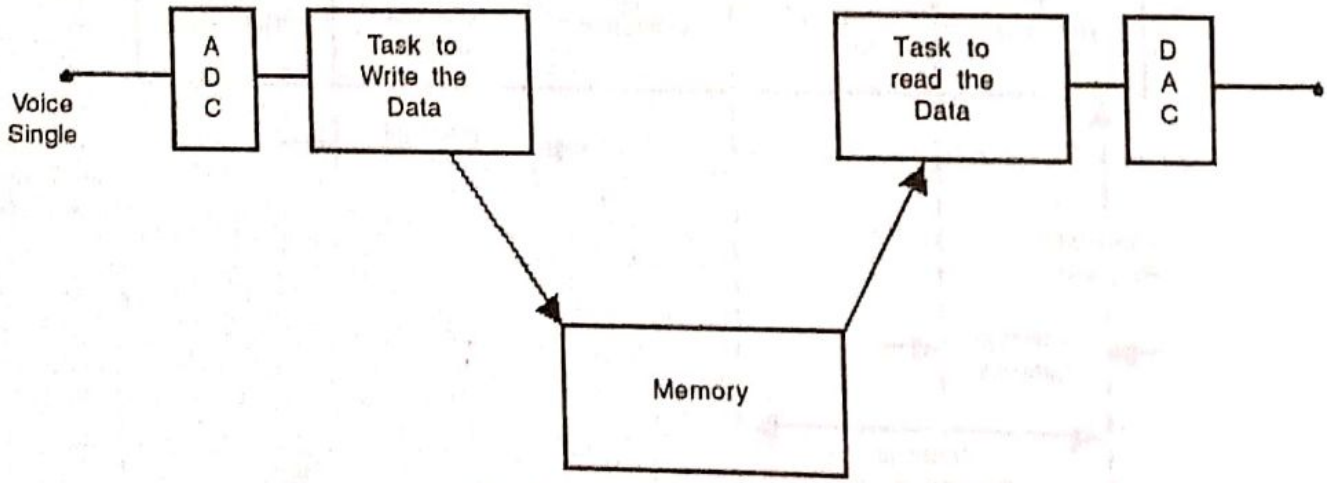


Fig. 7.8 (b): Task Synchronization

Now consider another situation shown in Fig. 7.8(b). In this case, one task reads the data from an ADC and writes it to memory. Another task reads that data and sends it to a DAC. The read operation takes place only after write operation and it has to be done very fast with minimal time delay. In this case, there should be a mechanism for task1 to inform task2 that it has done its job. This has to be done through a well-defined procedure. This is known as task synchronization.

Semaphore is a kernel object that is used for both resource synchronization and task synchronization.

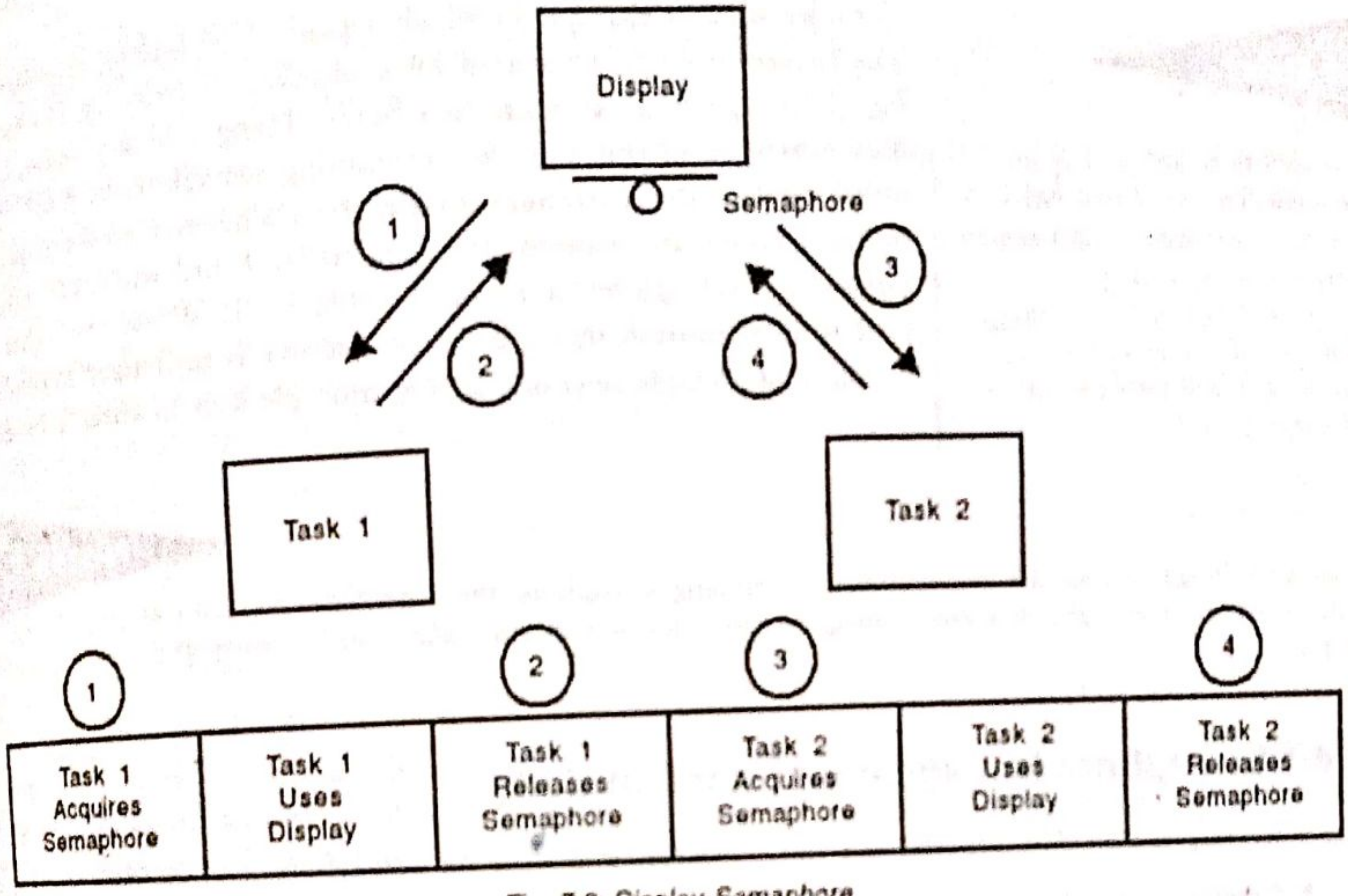


Fig. 7.9: Display Semaphore

Consider a situation shown in Fig. 7.9. Two tasks want to access a display. Display is a shared resource. To control the access, a semaphore is created. The semaphore is like a key to enter a house, and hence the semaphore is represented as a 'key'. If task1 wants to access the printer, it acquires the semaphore, uses the printer and then releases the semaphore. If both the tasks want to access a resource simultaneously, the kernel has to give the semaphore only to one of the tasks. This allocation may be based on the priority of the task or on first-come-first-served basis. If a number of tasks have to access the same resource then the tasks are kept in a queue and each task can acquire the semaphore one by one.

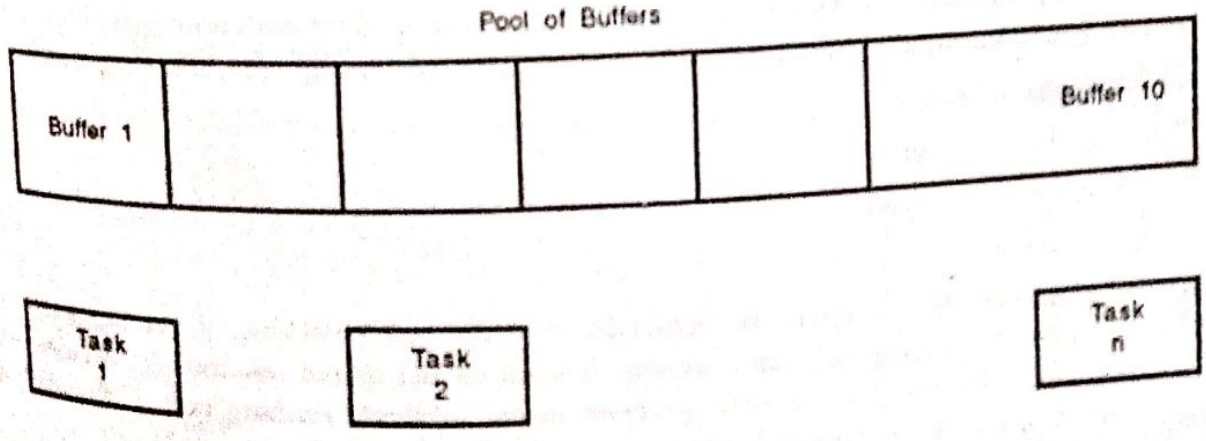


Fig. 7.10: Counting Semaphore

In Brief...

Semaphore is just an integer. Semaphores are of two types: counting semaphore and binary semaphore. Counting semaphore will have an integer value greater than 1. Binary semaphore will take the values of either 0 or 1.

Consider another example in which a pool of 10 buffers is available. The buffers need to be shared by a number of tasks, as shown in Fig. 7.10. Any task can write to a buffer. Using a binary semaphore does not work in this case. So, a counting semaphore is used. The initial value of the semaphore is set to 10. Whenever a task acquires the semaphore, the value is decremented by 1 and whenever a task releases the semaphore, it is incremented by 1. When the value is 0, it is an indication that the shared resource is no longer available. A counting semaphore is like having multiple keys to enter a house.

Notes...

If an 8-bit integer is used for implementing a counting semaphore, the semaphore can take a value between 0 and 255. If a 16-bit integer is used, the semaphore value can be between 0 and 65,535.

7.4.1 Semaphore Management Function Calls

The operating system API provides the following function calls for semaphore management:

- Create a semaphore
- Delete a semaphore
- Acquire a semaphore
- Release a semaphore
- Query a semaphore

7.5 Mutex

Mutex stands for mutual exclusion. Mutex is the general mechanism used for both resource synchronization as well as task synchronization. Mutual exclusion can be achieved through the following mechanisms:

- Disabling the scheduler
- Disabling the interrupts
- By test-and-set operations
- Using semaphore

Disabling the scheduler: As you know, the scheduler does the task switching, if the scheduler itself is disabled, the currently running task can complete its work on the shared resource. An excellent idea, but a very dangerous idea too! Due to some problem in the presently running task, if the scheduler cannot be enabled again, the system will crash.

Disabling the interrupts: A task in the Waiting state will move to the Ready-to-Run state through an ISR. If the interrupts are disabled, this movement will not happen. So, a solution for mutual exclusion is to disable the interrupts, execute the critical section of the code for using the shared resource and

then enable the interrupts. This is certainly a good solution, but the interrupt latency will be more. Certainly, this method can be used, but with caution.

Test-and-set operations: When two tasks have to share a resource, the functions in each of the tasks can check the value of a global variable to obtain the status of the shared resource. For instance, a global variable 'x' can be set to 0 if the resource can be used and it can be set to '1' if it cannot be used. The task has to first check the value of 'x'. If it is 0, it is set to 1—the resource is used, and then it is set back to 0. Meanwhile, if the second task wants to use the resource, it will find that the value is 1 and so it waits for some more time and checks again. Note that while the test-and-set operation is in progress, the interrupts have to be disabled. So, the procedure is

- Disable the interrupts
- Set the variable
- Access the shared resource
- Set the variable
- Enable the interrupts

Note that mutex is a special binary semaphore. A mutex can be either in locked state or unlocked state. A task acquires (locks) a mutex and after using resource, releases (unlocks) it. It is much more powerful than semaphore because of its special features listed below:

- It will have an owner. Initially, the mutex is in unlocked state. The task which acquires it is the owner. Only the task that is the owner can release the mutex, not any other task. Binary semaphore can be released by any task that did not originally acquire it.
- Owner can acquire a mutex multiple times in the locked state. If the owner locks it 'n' times, the owner has to release it 'n' times.
- A task owning a mutex, cannot be deleted.
- The mutex supports priority inheritance protocol to avoid priority inversion problem, which will be discussed later.

7.5.1 Mutex Management Function Calls

The operating system function calls provided for mutex management are:

- Create a mutex
- Delete a mutex
- Acquire a mutex
- Release a mutex
- Query a mutex
- Wait on a mutex

In Brief...

Mutex stands for mutual exclusion. Mutual exclusion can be achieved by (a) disabling the scheduler; (b) disabling the interrupts; (c) by test-and-set operations; or (d) semaphores.

Notes...

Mutex and semaphore can be used for a number of activities such as: (a) to control access to a shared resource, (b) to indicate (signal) the occurrence of an event; and, (c) to synchronize the activities of tasks.

In Brief...

Deadlock occurs when two or more tasks wait for a resource being held by another task. If the resource is not released for a long time due to some problem in that task then the system may be reset by the watchdog timer.

Deadlock

Deadlock occurs when two or more tasks wait for resource being held by another task. To avoid deadlock, a time limit can be set. When a task is waiting for a semaphore or mutex, it can wait for a fixed time; if even after this wait, the resource is not available, the queue can be emptied.

7.6 Mailboxes

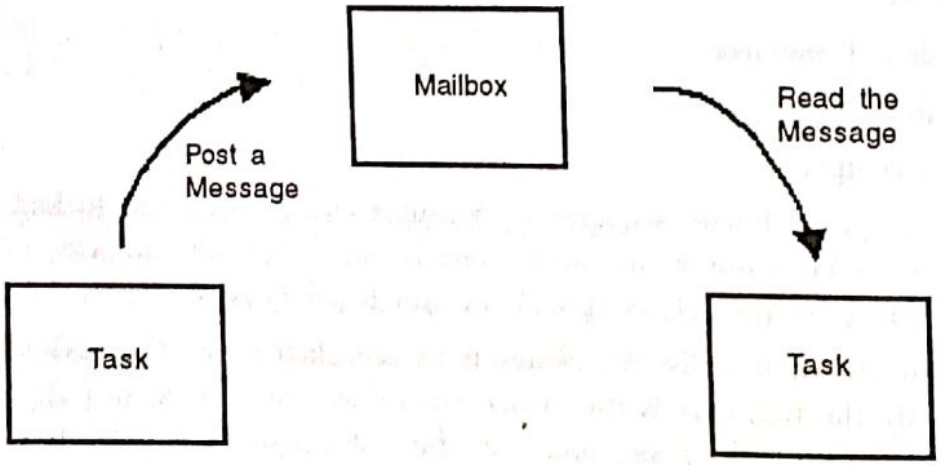


Fig. 7.11 Inter-task Synchronization through Mailbox

A mailbox object is just like your postal mailbox. Someone posts a message in your mailbox and you take out the message. A task can have a mailbox into which others can post a mail. A task or ISR sends the message to the mailbox.

7.6.1 Mailbox Management Function Calls

To manage the mailbox object, the following function calls are provided in the operating system API:

- Create a mailbox
- Delete a mailbox
- Query a mailbox
- Post a message in a mailbox
- Read a message from a mailbox

In Brief...

Mailbox is a kernel object for inter-task communication. A task posts a message in the mailbox and another task will read the message

7.7 Message Queues

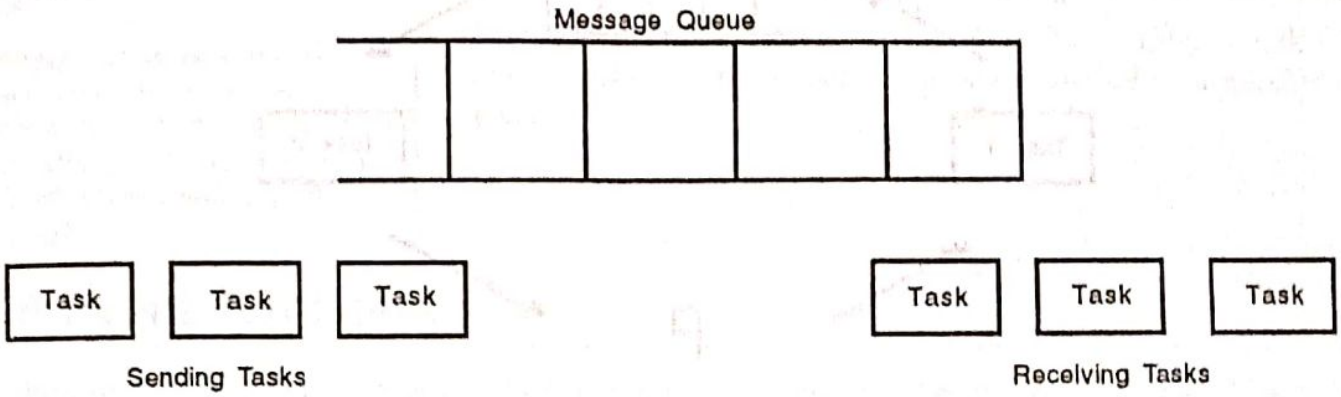


Fig. 7.12: Message Queue

Message queue can be considered as an array of mailboxes. Some of the applications of message queue are:

- Taking the input from a keyboard
- To display output
- Reading voltages from sensors or transducers
- Data packet transmission in a network

In each of these applications, a task or an ISR deposits the message in the message queue. Other tasks can take the messages. Based on your application, the highest priority task or the first task waiting in the queue can take the message.

At the time of creating a queue, the queue is given a name or ID, queue length, sending task waiting list and receiving task waiting list.

7.7.1 Message Queue Management Function Calls

The following function calls are provided to manage message queues:

- Create a queue
- Delete a queue
- Flush a queue
- Post a message in queue
- Post a message in front of queue
- Read message from queue
- Broadcast a message
- Show queue information
- Show queue waiting list

Message queues have a number of applications as shown in Fig. 7.13.

7.7 Message Queues

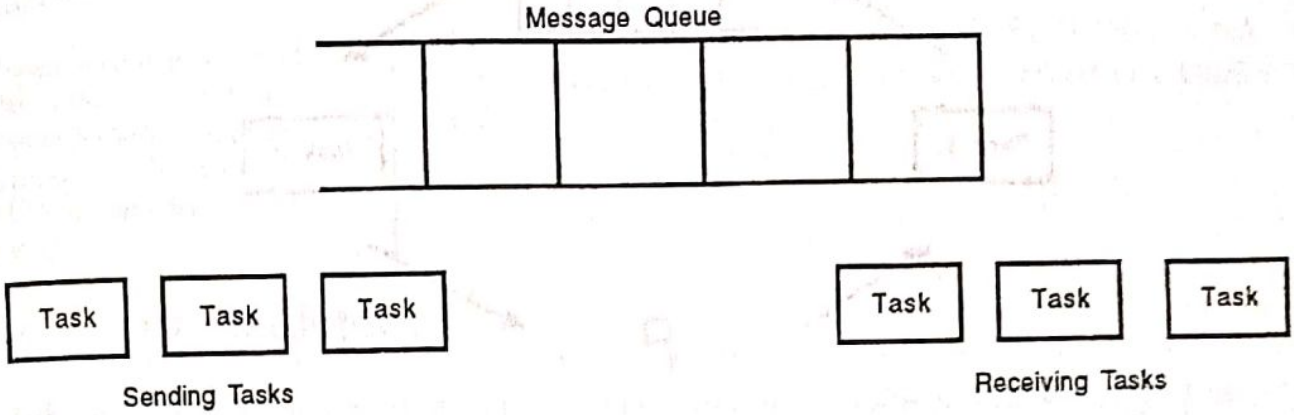


Fig. 7.12: Message Queue

Message queue can be considered as an array of mailboxes. Some of the applications of message queue are:

- Taking the input from a keyboard
- To display output
- Reading voltages from sensors or transducers
- Data packet transmission in a network

In each of these applications, a task or an ISR deposits the message in the message queue. Other tasks can take the messages. Based on your application, the highest priority task or the first task waiting in the queue can take the message.

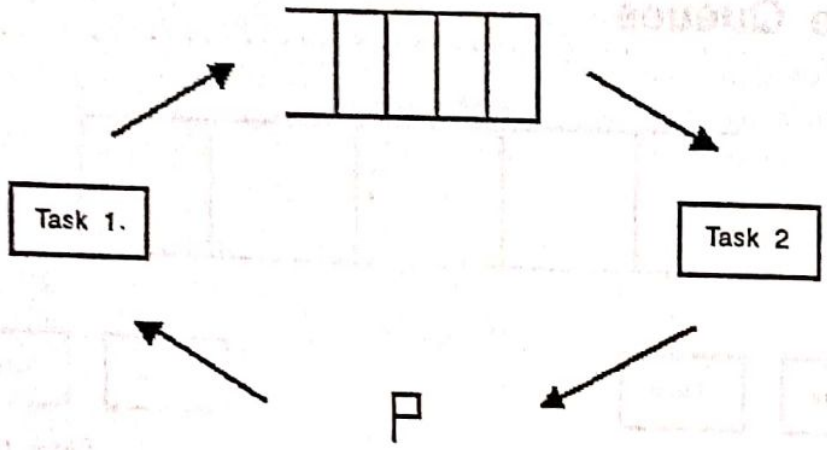
At the time of creating a queue, the queue is given a name or ID, queue length, sending task waiting list and receiving task waiting list.

7.7.1 Message Queue Management Function Calls

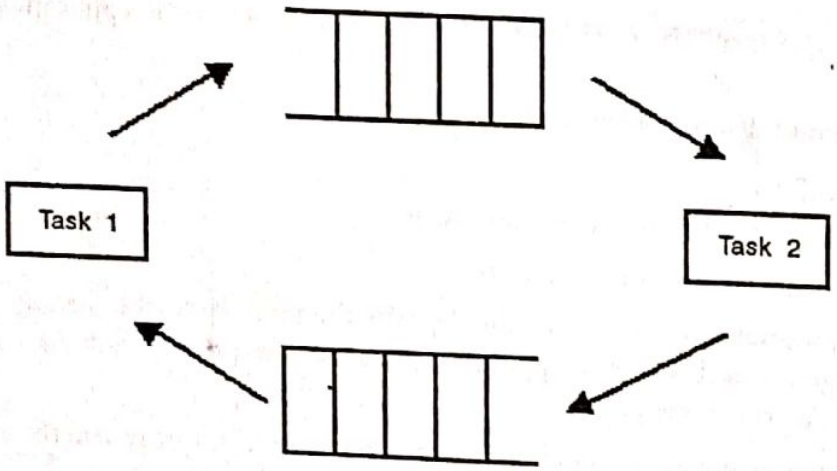
The following function calls are provided to manage message queues:

- Create a queue
- Delete a queue
- Flush a queue
- Post a message in queue
- Post a message in front of queue
- Read message from queue
- Broadcast a message
- Show queue information
- Show queue waiting list

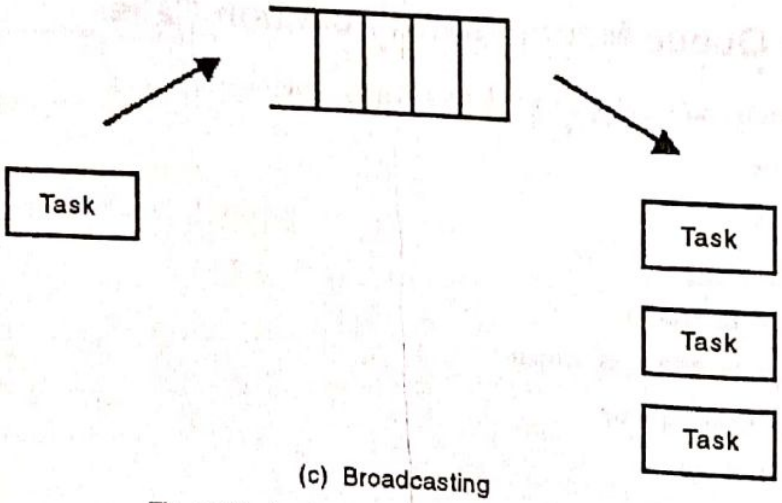
Message queues have a number of applications as shown in Fig. 7.13.



(a) One-way Communication with ACK



(b) Two-way Communication



(c) Broadcasting

Fig. 7.13: Applications of Message Queues

One-way data communication: As shown in Fig. 7.13(a), a task can send messages to the queue which are read by the other task. For each message received, the second task uses a semaphore to indicate an acknowledgement.

In Brief...

Message queue is a kernel object used for inter-task communication. A task posts the messages in the message queue and other tasks will read the messages.

Two-way data communication: As shown in Fig. 7.13(b), two tasks can have two queues to send messages in both directions.

Broadcast communication: As shown in Fig. 7.13(c), a task can send messages to the queue which are broadcast to a number of tasks.

7.8 Event Registers

A task can have an event register in which the bits correspond to different events. In Fig. 7.14, a 16-bit event register is shown. These 16 bits are divided into four portions corresponding to the events of three tasks and one ISR. For example, task2 can set the first four bits to 1 or 0. Using a predetermined protocol, the meaning for each of these bits is decided. Task1 comes to know the status of events from these bits. Each of the bits in the event register is an event flag.

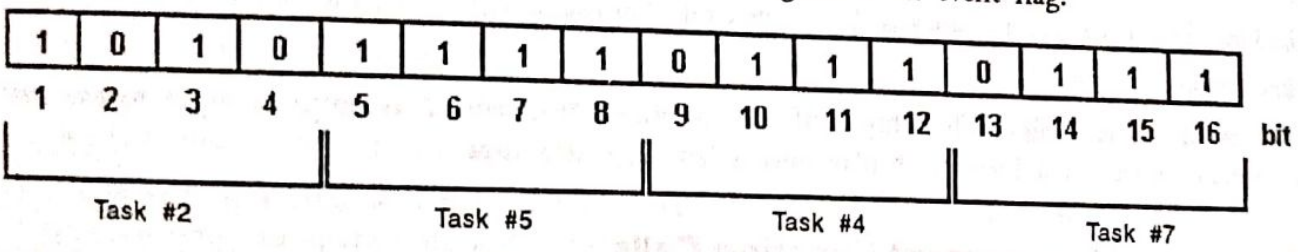


Fig. 7.14: 16-bit Event Register

7.8.1 Event Register Management Function Calls

In Brief...

For managing the event registers, the following function calls are provided:

- Create an event register
- Delete an event register
- Query an event register
- Set an event flag
- Clear an event flag

Each bit in an event register can be used to obtain the status of an event. A task can have an event register and other tasks can set/clear the bits in the event register to inform the status of an event. The meaning of 1 or 0 for a particular bit has to be decided beforehand.

7.9 Pipes

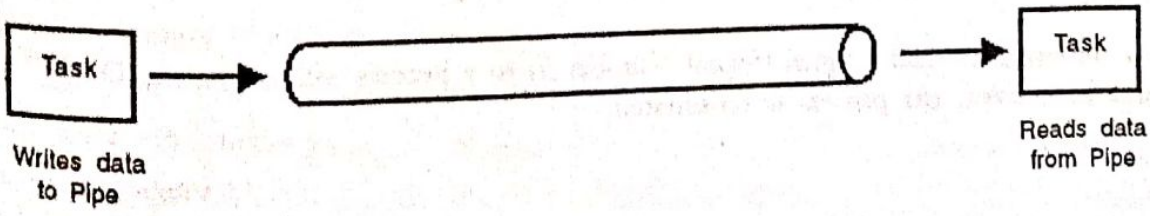


Fig. 7.15: Pipe

As shown in Fig. 7.15, a task can write into a pipe and the other task reads the data that comes out of the pipe. In other words, the output of one task is passed on as input to the other task. Task-to-task or ISR-to-task data transfer can take place using pipes.

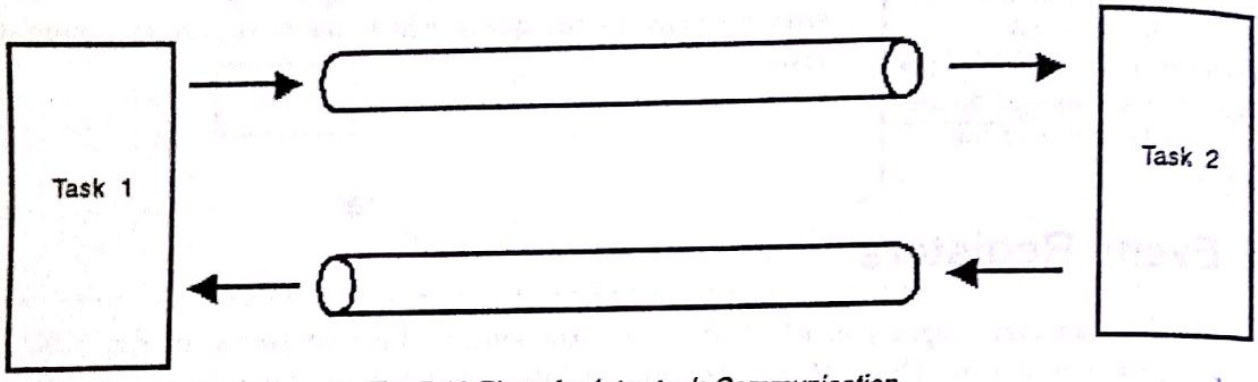


Fig. 7.16 Pipes for Inter-task Communication

Pipes can be used for inter-task communication as shown in Fig. 7.16. One task may send the data packets through one pipe and the other task may send acknowledgements through the other pipe.

In Unix/Linux, we use the pipes as shell commands. For example, consider the following shell command:

$cat hello.c | more

The symbol | is a pipe. The output of the command "cat hello.c" is given as input to the 'more' command which is a filter to display only a few lines at a time.

7.9.1 Pipe Management Function Calls

The function calls in the operating system API to manage the pipes are:

- Create a pipe
- Open a pipe
- Close a pipe
- Read from the pipe
- Write to the pipe

In Brief...

Pipe is a kernel object for inter-task communication. Pipe is used to send the output of one task as input to another task for further processing.

7.10 Signals

Signals can be passed to indicate an event. However, many RTOS do not support it then and their use is discouraged. Again, in shell commands, the signals are sent to kill a process. Consider the following command:

$kill –9 879

This is a command to send a signal (Signal Number 9) to a process with a process ID of 879. When the signal is received, the process is terminated.

7.10.1 Signal Management Function Calls

The function calls to manage a signal are:

- Install a signal handler
- Remove an installed signal handler
- Send a signal to another task
- Block a signal from being delivered
- Unblock a blocked signal
- Ignore a signal

In Brief...

Signal is a kernel object to indicate the occurrence of an event to a task. The use of signals is generally discouraged.

7.11 Timers

Timers are used to measure the elapsed time of events. For instance, the kernel has to keep track of different times:

- A particular task may need to be executed periodically, say, every 10 msec. A timer is used to keep track of this periodicity.
- A task may be waiting in a queue for an event to occur. If the event does not occur for a specified time, it has to take appropriate action.
- A task may be waiting in a queue for a shared resource. If the resource is not available for a specified time, an appropriate action has to be taken.

7.11.1 Timer Management Function Calls

The following function calls are provided to manage the timer:

- Get time
- Set time
- Time Delay (in system clock ticks)
- Time delay (in seconds)
- Reset timer

7.12 Memory Management

In addition to the kernel objects discussed above, memory management is an important service provided by the kernel. The API provides the following function calls to manage memory:

- Create a memory block
- Get data from memory
- Post data in the memory
- Query a memory block
- Free the memory block

7.13 Priority Inversion Problem

When tasks share a resource, there is a possibility of getting into a problem, known as priority inversion problem. Mars Pathfinder software bug (mentioned in Chapter 4) got into this problem. Let us study this problem with just three tasks—Low Priority (LP) task, Medium Priority (MP) task and High Priority (HP) task.

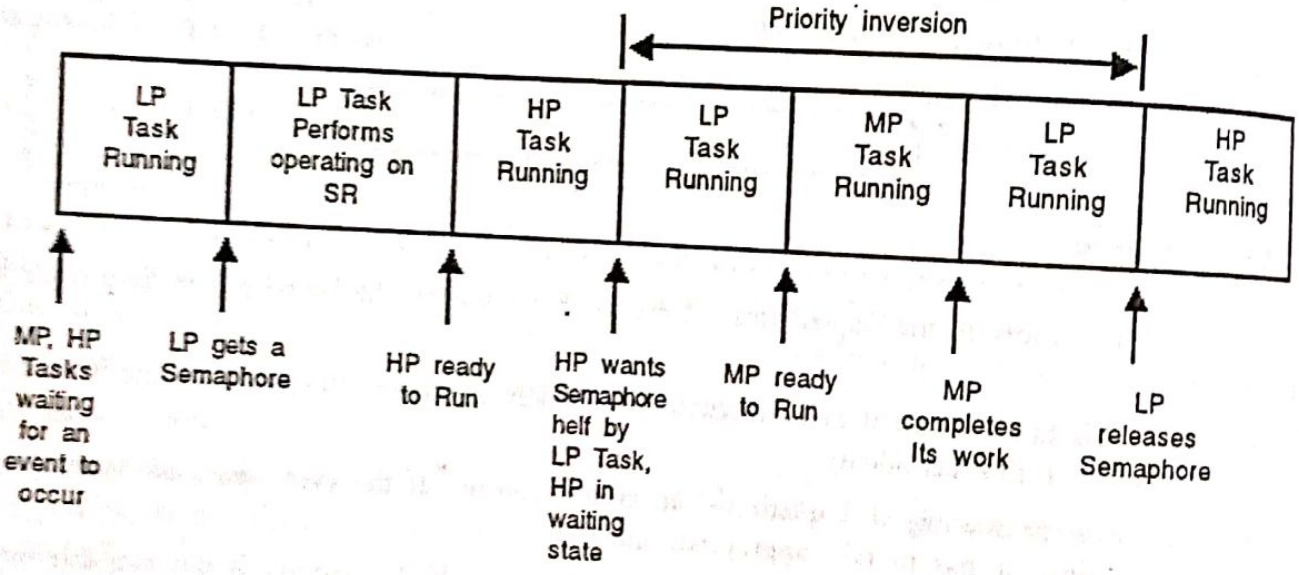


Fig. 7.17: Priority Inversion Problems

As shown in Fig. 7.17, to start with, the MP task and HP task are in the Waiting state and the LP task is Running. After some time, LP gets a mutex for a shared resource (SR) and performs some operations on the shared resource, but it still has not completed its operations. Meanwhile, the HP is Ready-to-Run and hence the HP task is executed by the CPU. Now the HP wants the mutex of the shared resource being held by LP task. So, HP task is moved to the Waiting state and the LP task is executed. After some time, the MP task is Ready-to-Run and hence the LP task is preempted and the MP task is Running. The MP task completes its job and now the LP task is Running. When LP task releases the mutex, HP task acquires it and now this task gets executed. In this process, the high priority task has to wait for a very long time, in spite of its high priority due to the semaphore held by the LP task. This problem is called the priority inversion problem because the priorities of LP task and HP task are effectively inversed!

In Brief...

Priority inversion problem arises when a high priority task has to wait while a lower priority task executes. To overcome this problem, priority inheritance protocol is used.

7.13.1 Priority Inheritance

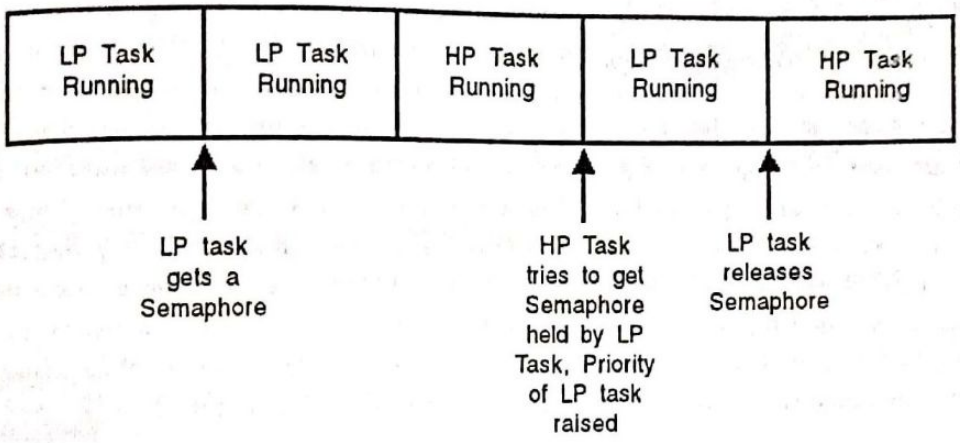


Fig. 7.18 Priority Inheritance

Priority Inheritance provides a solution to the problem of priority inversion. When the LP task acquires the mutex, its priority can be increased to a value more than the priority of the HP task. Kernel has to automatically change this priority. In such a case, the LP task will complete its work very fast and release the mutex which can be acquired by the HP task. Generally, in Priority Inheritance, the priority of the task which acquired the mutex will be increased to a value higher than the priority of the task competing for that resource.

7.13.2 Pathfinder Problem Revisited

In the Mars Pathfinder, priority-based preemptive multitasking operating system was used. Information bus management task was of high priority, communication management task was of medium priority and meteorological data collection task was of low priority. The high priority task and the low priority tasks shared a resource—shared memory. Whenever the low priority task has to access the shared medium, it would acquire a mutex, write the data to the shared memory and release the mutex. Occasionally, the medium priority task was running, while the high priority task was waiting for the low priority task to release the mutex. If the medium priority task was running for a longer time, the watchdog timer used to suspect that something was wrong, and it used to reset the processor. The problem was solved using priority inheritance. The low priority task would become high priority task while holding the mutex. To make this change, while initializing the mutex, the parameter to set the priority inheritance was to be changed from FALSE to TRUE. This is just a change in the file that declares global variables. So, this change was made in the file and the file uploaded to the Pathfinder embedded system. And, no longer the system was reset due to the priority inversion problem.

Summary

This chapter presented a conceptual overview of the embedded/real-time operating system kernel objects and services. The kernel objects are: task scheduler, tasks, semaphores, mutexes, Interrupt Service Routines, mailboxes, message queues, event registers, pipes, signals and timers. The task scheduler is the heart of the operating system. It decides which task has to run next based on a

scheduling algorithm. Preemptive multitasking scheduling algorithm is used in embedded/real-time operating systems. In this algorithm, the highest priority task which is Ready-to-Run will be executed. A task can be in one of the three states: Running, Ready-to-Run and Waiting. A task which is in the Waiting state can move to the Ready-to-Run state and inform its change of state through an ISR. In multitasking environment, task synchronization and resource synchronization are the two important issues. To share resources and also for inter-task communication a number of kernel objects are used. Semaphores are used to bring in discipline in accessing shard resources. Semaphore is of two types: binary semaphore and counting semaphore. Binary semaphore takes only two values 1 and 0. Counting semaphore can take a larger value. Mutex is similar to binary semaphore but it has the additional features of ownership and support of priority inheritance. Mailboxes, message queues, event registers, pipes and signals are used for inter-task communication. Timer provides the necessary timing services. The operating system kernel is a collection of these objects and operations on these objects to provide the necessary functionality required to manage the embedded system resources.

Questions

1. What are the objects of an operating system kernel?
2. What is task scheduling? Explain the various scheduling algorithms.
3. What is interrupt latency?
4. Explain how a semaphore can be used for inter-task synchronization.
5. What is the difference between semaphore and mutex?
6. Explain the use of message queues, mailboxes and pipes.
7. Differentiate between preemptive and non-preemptive operating systems.

Exercises

1. You need to develop an embedded system that takes analog voice signal as input, converts it into digital format using an ADC, converts the digital data into packets and sends the packets over a data network. List the various tasks in the embedded software. How do you assign priorities to the tasks? Do you need an embedded operating system? If so, of what type of operating system?
2. Is priority inheritance an important feature? Discuss.
3. List some real-time applications for which you cannot use the desktop computer.
4. Semaphores can be used for data communication. List such applications.
5. Run the following program on a Linux machine and analyze the output:

```
int main(void)
{
        printf("Hello\n");
        if(fork()) = = 0)
          printf("World\n");
}
```

Chapter 8

Overview of Embedded/Real-Time Operating Systems

CHAPTER OBJECTIVES

After reading this chapter, you will be able to:

- Understand the commonalities and differences in the operating systems available off-the-shelf
- Get a good understanding of commercial and open source operating systems used in embedded/real-time systems as well as handheld/mobile devices.

To develop embedded software, a number of commercial and open source operating systems are available. In this chapter, we will study the features that are common to all the operating systems and also the features that differ. We will also study the salient features of some of the widely used operating systems.

8.1 Off-the-Shelf Operating Systems

To reduce development time and effort, it is preferable to use an off-the-shelf operating system that suits your application needs. Depending on the application, you need to choose one of the following categories of operating systems:

- **Non-real-time embedded operating systems:** These operating systems are suitable for non-real-time applications. They use a preemptive kernel, but strict deadlines cannot be met.
- **Real-time operating systems:** These operating systems provide the necessary functionality for achieving real-time performance through very low interrupt latency. These are 'deterministic' operating systems, i.e. the worst-case response time can be predicted.
- **Handheld/mobile operating systems:** These operating systems are meant for handheld computers and mobile devices such as smart phones. Operating systems such as Palm OS, Symbian OS and Windows CE have been develo d to address this

In Brief...

The operating systems used in embedded systems can be classified as (a) embedded operating systems that do not meet real-time requirements; (b) real-time operating systems; and (c) handheld/mobile operating systems.

market. But in recent years, many other embedded/real-time operating systems are being ported onto the handheld/mobile devices.

8.1.1 Commonalities of the Operating Systems

Because of the immense competition in the embedded/real-time operating system market, the features supported by operating systems of different vendors appear almost the same. Of course, every vendor claims that his operating system is the best. The features listed below are common to most of these operating systems:

- **Integrated Development Environment (IDE):** To facilitate easy and fast development, vendors supply an IDE that includes editor, compiler, debugger and also the necessary cross-platform development tools.
- **POSIX compatibility:** POSIX1003.1-2001 standard specifies the Application Programming Interface to achieve portability of applications. Many off-the-shelf operating systems provide compliance to POSIX.
- **TCP/IP support:** TCP/IP protocol stack, including application layer protocols such as FTP, SMTP, HTTP, is integrated along with the operating system software. Function calls will be provided to access the network-related services.
- **Device drivers:** A number of device drivers are provided for commonly used devices such as serial port, parallel port, USB etc. A Device Driver Kit (DDK) is also generally included that facilitates fast development of device drivers.

In Brief...

An Integrated Development Environment, POSIX compatibility, TCP/IP support and device driver kit are some of the important features offered by all the embedded operating system vendors.

8.1.2 Portable Operating System Interface (POSIX)

POSIX is a standard developed by IEEE. Before POSIX was standardized, every OS vendor used to give his proprietary Application Programming Interface (API) for application development. This interface is a set of function calls to access the operating system objects and services. If you developed the application using the API supplied by one vendor, it was not possible to port the application to another operating system. POSIX standard addressed this problem and the API was standardized. IEEE POSIX 1003.1c-2001 standard specifies the API for portable operating system interface. IEEE POSIX 1003.13 "Standardized Application Environment Profile—POSIX Real-time Application Support" addresses the API for real-time embedded systems. This standard gives the various C language function calls and library functions that need to be implemented by the Operating System (OS) vendors. The concept of threads became popular only because of this standard in fact, threads are referred as POSIX threads. As operating systems can be used for wide range of applications from tiny systems to very large multi-processor based systems, different profiles are defined in the POSIX standard. Small System POSIX Profile is for small embedded systems that have a single process with multiple threads.

In Brief...

Portable Operating System Interface (POSIX) standard facilitates portability of applications across different operating systems.

Notes...

POSIX standards are also available for other programming languages such as ADA and FORTRAN.

3.1.3 Differences in Operating Systems

Off-the-shelf operating systems differ in the following aspects:

- **Support for processors:** The operating system code consists of (a) processor-independent code; and (b) processor-dependent code. Operations such as context switching, in which the contents of the CPU registers have to be stored in the memory, have to be done through assembly language programming. Hence, a small portion of the operating system software is processor-dependent. As a result, every operating system may not support all the processors. Commercial operating systems support the popular processors such as Intel x86/Pentium, MIPS, PowerPC, Intel StrongARM etc. For the specific processor of your choice, you need to check whether the operating system 'port' is available.
- **Footprint:** The footprint or the memory occupied by the kernel differs from OS to OS. Real-time operating system kernels require only a few Kilobytes of memory. The operating system vendor specifies the minimum amount of RAM and ROM required for the kernel. Of course, you need to calculate the total memory requirement for your embedded system keeping in view the size of the application software and the communication software.
- **Java environment:** In tune with the recent trends in using Java for embedded systems, some vendors provide a Java Virtual Machine (JVM) support on their OS. If you want to develop applications in Java, you need to check on the availability of the JVM.
- **Board Support Packages (BSPs):** Some vendors supply hardware boards built around different processors with OS ported onto the hardware. These are called BSPs. BSPs speed up software development.
- **Scheduling algorithms:** Some operating systems provide a number of scheduling algorithms such as round-robin, first-in-first-out etc. whereas some operating systems support only priority-based preemptive scheduling. If you want flexibility in scheduling algorithms, your choice may be comparatively limited.
- **Priority inheritance:** Whether priority inheritance is a good feature or not, it is a debatable issue. Some operating systems do not support this feature, as their developers dislike it. Some operating systems do support it as their developers feel that the designer should be given flexibility in application development.
- **Maximum number of tasks:** The maximum number of tasks supported differs from OS to OS. Some OSs support only 64 tasks, some OSs claim 'unlimited' number though there certainly will be a limit. For example, if a 32-bit integer is used for task ID, the number of tasks is limited to 2^{32}.
- **Assigning task priorities:** In some OSs, each task should have a unique priority and the priority is fixed. In some other OSs, the priority can be changed dynamically during execution time.

- **POSIX support:** In the section on commonalities of operating systems, we discussed POSIX support. Why is it appearing here again?! Though many vendors claim POSIX support, you need to check the details. Small System POSIX profile is for minimal real-time environment. Multi-threaded single POSIX process is supported in such environments. Support for this profile does not mean full-fledged POSIX support. If you are gullible, the vendor may take you for a ride!
- **Licensing terms:** The prices of the operating systems differ. Some are free under the GPL licence. For some operating systems, one-time payment will entitle you to the entire source code and you need not pay any royalty for each copy of runtime licence. Some vendors ask for a small amount for the development system without source code, and exorbitant amount for the source code. Check on these cost aspects and then do a hard bargain to get the best deal.

In Brief...

Embedded operating systems differ in many aspects: the processors supported, footprint, support for Java environment, Board Support Packages availability, scheduling algorithms, number of tasks and methodology for assigning priorities, support for priority inheritance and licensing terms.

In the following sections, we will briefly review the various off-the-shelf operating systems available from commercial vendors as well as under the GPL licence.

Notes...

Though operating system vendor claim POSIX support, this support, in fact, may be only for "small system POSIX profile" which is for small real-time systems. Support for this profile does not mean a full-fledged POSIX support.

8.2 Embedded Operating Systems

In Brief...

Embedded operating systems that do not meet real-time requirements use a preemptive priority based kernel, but they cannot meet strict deadlines. Stripped-down versions of desktop operating systems can be used as embedded operating systems.

Embedded operating systems use a preemptive priority based kernel, but they do not meet strict deadlines. The stripped-down versions of the desktop operating systems can be used as embedded operating systems i.e., in the operating system software, remove all the unnecessary features and make the kernel occupy a small memory and you have the embedded operating system. This strategy is used in developing the following three embedded operating systems:

- Embedded NT
- Windows XP Embedded
- Embedded Linux

8.2.1 Embedded NT

Many embedded systems use the Single Board Computer (SBC) hardware. This hardware is essentially same as the desktop computer's hardware. The embedded system, however, does a focused job; hence the application and the operating system together need to be bundled and transferred to the target

hardware. Typical applications include Internet Kiosk, ATM etc. For such applications, Microsoft's Embedded NT is an excellent choice. Embedded NT is based on Windows NT4.0. Embedded NT works on 80x86 and Pentium processors only.

The requirement of Embedded NT for minimal operating system functionality without any network support is 9 MB of RAM and 8 MB of program such as Flash. The exact memory requirement of the target hardware depends on the application. If the application is a single Win32 application with networking capability, minimum RAM requirement is 16 MB. With network components and device drivers, the requirement is about 16 MB of RAM and 16 MB of program memory.

Application development using Embedded NT is very easy. You can develop the applications in Visual Studio environment (Visual Basic or Visual C++) and the application can be ported onto the target system. IIS 3.0 is included in Embedded NT.

To develop applications using Embedded NT, the development system has to be installed on a system with the following configurations:

- Windows NT 4.0
- Service Pack 4.0 or later
- Internet Explorer 5.0 or later
- Visual Studio 6.0 development environment
- Windows NT 4.0 Service Resource Kit

Development tools called the Component Designer and the Target Designer are provided in Embedded NT. The Component Designer facilitates defining and adding components to the Target Designer. The Target Designer is used to create a bootable NT target system. It generates a target image, which consists of a number of directories. All these directories and files need to be copied onto the target machine's file system. Target system can be created in any of the following ways:

- **Create a headless system:** Embedded systems that do not have a monitor, keyboard or mouse are called headless systems. To create a headless system, the following components need to be added:
  - Null VGA
  - Null Keyboard Drive
  - Null Mouse

Debugging a headless system is a problem because of lack of input/output devices. To overcome this problem, "Console Administration Component" is provided. This component allows monitoring the embedded system through a serial interface.

- **Create a target system that boots from a CDROM:** The target image generated by the Target Designer is transferred to a CDROM and the embedded system boots from the CDROM.
- **Create a bootable partition on the hard disk:** In this case, a partition can be created on the hard disk from which the Embedded NT target image can boot.

In Brief...

Embedded NT is only for x86/Pentium processors. It is suitable for embedded systems built around single board computers for applications such as Internet kiosks and Automatic Teller Machines. 9 MB of RAM and 8 MB of ROM are the minimum requirements.

- **Create the target image on an empty hard disk:** Target image can be transferred to an empty hard disk and the hard disk can be connected to the embedded system. This approach is the best while testing the embedded software in the lab environment.

8.2.2 Windows XP Embedded

Microsoft's Windows XP Embedded (abbreviated Embedded XP in this section), is the successor to Embedded NT. Like Windows NT, it is also a preemptive multitasking operating system and uses Win32 applications and drivers. Hence, the main attraction of this operating system is that applications developed on desktop using Visual Studio can be ported onto the embedded system. However, as compared to many other embedded operating systems, it requires huge resources particularly the RAM and the Flash memory.

In Brief...

Windows XP Embedded is a good choice for single board computer-based embedded systems that do not have real-time requirements. Support for IEEE 802.11 and infrared interfaces, and IP version 6 are its attractions.

Embedded XP is now being used in a number of embedded systems such as set top boxes, point of sale terminals, Internet Kiosks, etc. Embedded XP has many attractive features such as support for IP version 6, IrDA compliance, 802.11 LAN connectivity, Universal Plug and Play feature, etc. It also has support for Telephony APIs (TAPI) and NetMeeting to provide multimedia applications.

The application software to be embedded can be created using Embedded Visual Studio which provides the following utilities to create the target image:

- Target Designer, which is used to create bootable runtime image of the application you have developed. You can transfer the runtime image to the target system, and the system will boot with Embedded XP and the application will run from this image.
- Component Designer, which is used to create components required for the embedded application.
- Deployment tools, which are used to prepare the target device for the runtime image.

The bootable image is created using the Target Designer and transferred to a CDROM or a Flash (disk on a chip). The memory requirement of the embedded system depends on the application, but minimum of 9 MB RAM and 8 MB of ROM are required.

Notes...

The attractive feature of Windows-based embedded operating systems is that applications developed using Visual Studio can be easily ported onto the embedded system.

8.2.3 Embedded Linux

Open source software revolution started with the development of Linux. Like Linux kernel, Embedded Linux kernel is covered by the GNU General Public License (GPL) and hence the complete source code is available free of cost. GPL also permits redistribution of the source code even if it is modified as per the requirements of your application. However, modified source code should be made available usually for a nominal cost which is not more than the cost of reproducing the software. You can get the details of open source and its implications at www.gnu.org/copyleft/gpl.html. It is expected that

open source software will have profound impact on the industry in the coming years and Embedded Linux is likely to be used extensively in embedded applications. It is now being used on a number of devices such as PDAs, Set Top Boxes, cellular phones, Internet appliances and also in major mission critical equipment such as telecommunication switches and routers.

Embedded Linux (www.embedded-linux.org) is available freely and openly in source code form and is gaining popularity. The vendors who offer embedded Linux solutions and support for both real-time and non real-time applications include Coventive, FSM Labs, Lineo, LynuxWorks, Mizi, MontaVista, PalmPalm, RedHat, ridgeRun, TimeSys etc.

The main attractions of embedded Linux are:

- Open source software
- Availability of a large number of software resources in source code form (device drivers, application software, networking software, protocol software, speech/image processing software, etc.)
- Support for POSIX
- No royalty
- Availability of many people with expertise in Unix/Linux programming.

In Brief...

The attractive features of embedded Linux are: open source software, POSIX support, availability of many software resources in source code form and availability of a large pool of experts for programming.

As many non-standard variations of embedded Linux are available (mainly due to the fact that the source code is available to everyone and anyone can make the changes), it may lead to non-portable applications.

Embedded Linux Consortium Platform Specifications

Embedded Linux Consortium (ELC) released the Embedded Linux Consortium Platform Specifications (ELCPS) to bring in standardization for using Linux in embedded systems. ELCPS Version 1.0 was released in December 2002. It is compatible with POSIX 1003.1-2001 as well as POSIX1003.13.

ELCPS defines three environments to cater to different embedded system requirements. These are

- Minimal system environment
- Intermediate system environment
- Full system environment

For each of these environments, the APIs to be supported are defined in ELCPS. Operating system vendors can ensure conformance to these specifications so that the applications can be made portable. If you are modifying the Linux kernel to suit your embedded system requirements, you need to ensure ELCPS conformance.

Minimal system environment: This configuration is applicable for embedded systems with one processor and associated memory. These embedded systems work in isolated mode without any user interaction and have no secondary storage or file system. Only one process with one or more Linux tasks or POSIX threads will be running. Hence, the set of APIs required for this type of embedded software development will be minimal and also the OS footprint will be very small.

Intermediate system environment: Embedded systems of this category will have one or more processors, but need not have any secondary storage. A file system can be built, say, in a Flash memory device. There will be multiple processes, asynchronous input/output operations and support for dynamic linking of objects (libraries). Support for secondary storage is also provided.

Full system environment: The embedded systems of this category will have one or more processors, secondary storage, network support, and user interfaces. This environment supports full multi-purpose Linux.

In Brief...

Embedded Linux Consortium Platform Specifications (ELCPS), finalized in December 2002, will ensure portability of applications.

The vendor can obtain conformance for any of the three environments. For each environment, the various API calls to be supported such as for task management, inter-process communication, file management, thread-safe general ISO C library interface, mathematical function library calls, header file definitions for symbolic constants etc. are specified.

Notes...

Embedded Linux Consortium Platform Specifications address three types of embedded systems: (a) minimal system environment for systems with one processor and memory; (b) intermediate system environment for systems with one or more processors and secondary storage; and, (c) full system environment which requires one or more processors, secondary storage, user interface and network support.

8.3 Real-Time Operating Systems

There are nearly 100 real-time operating systems in the commercial market. So, shopping for a real-time operating system is not an easy task. We will review the following operating systems:

- QNX Neutrino
- VxWorks
- MicroC/OS-II
- RTLinux

8.3.1 QNX Neutrino

In Brief...

QNX Neutrino is a real-time operating system that supports multiple scheduling algorithms and up to 65,535 tasks. MySQL can be integrated with this OS to create embedded database applications.

QNX Neutrino is a popular real-time operating system of QNX Software Systems Limited (www.qnx.com). It supports a number of processors such as ARM, MIPS, Power PC, SH-4, StrongARM, x86 and Pentium. Board Support Packages and Device Driver Kit help in fast development of your prototype. It provides an excellent Integrated Development Environment. It has support for C, C++ and Java languages and TCP/IP protocol stack.

It has support for multiple scheduling algorithms such as round-robin, FIFO etc. and the same application can use different scheduling algorithms for different tasks. Up to 65,535 tasks are

supported and each task can have 65,535 threads. Minimum time resolution is one nanosecond. Even small embedded systems can use this OS as it requires 64K kernel ROM and 32K kernel RAM.

8.3.2 VxWorks

Wind River's VxWorks (www.windriver.com) is one of the most popular real-time operating systems. This OS has been used in the Mars Pathfinder. It supports a number of processors including PowerPC, Intel StrongARM, ARM, Hitachi SuperH, Motorola ColdFire, etc.

In Brief...

VxWorks is a real-time operating system that supports multiple scheduling algorithms and also priority inheritance.

It supports both preemptive and round-robin scheduling algorithms. 256 priority levels can be assigned to the tasks. It supports priority inheritance. Those who are against priority inheritance need not use this feature, which is an option provided to the developer.

8.3.3 MicroC/OS-II

Micro-controller Operating System version II developed by Jean J. Labrosse (www.ucos-II.com) is a preemptive real-time operating system which is popular for teaching RTOS concepts. It is also used widely in many commercial applications including mission-critical applications. It is certified for use in commercial aircraft by Federal Aviation Administration. The standard RTCA.DO-178B specifies the requirements.

In Brief...

MicroC/OS-II is a real-time operating system used extensively in academic institutions for teaching operating system concepts. It is available in source code form for non-commercial purposes. The number of tasks can be 64 out of which eight are system tasks. Round-robin scheduling is not supported.

The author of this OS summarizes its features beautifully: "source code availability, ROMable, scalable, preemptive, portable, multitasking, deterministic, reliable, support for different platforms".

This OS supports 64 tasks out of which eight are system tasks. Hence, the application can have up to 56 tasks. Each task is assigned a unique priority. Round-robin scheduling algorithm is not supported by this operating system.

8.3.4 RTLinux

FSM Labs (www.fsmlabs.com) has two editions of RTLinux—RTLinuxPro and RTLinuxFree. RTLinuxPro is a priced-edition and RTLinuxFree is the open source release. RTLinux is a hard real-time operating system with support for many processors such as x86, Pentium, PowerPC, ARM, Fujitsu, MIPS and Alpha. A footprint of 4 MB is required for RTLinux. It does not support priority inheritance.

RTLinux runs underneath the Linux operating system. The Linux OS becomes an idle task for RTLinux. RTLinux tasks are given priority as compared to Linux tasks. Interrupts from Linux are disabled to achieve real-time performance. This interrupt disabling is done using a layer of emulation software between the Linux kernel and the interrupt controller hardware.

In Brief...

RTLinux runs underneath the Linux operating system. The Linux is an idle task for RTLinux. The real-time software running under RTLinux is given priority as compared to non-real-time threads running under Linux. This OS is an excellent choice for 32-bit processor based embedded systems.

The tasks, which do not have any timing constraints, will run in the Linux kernel only. Soft real-time capability is provided by Linux system. Hard real-time tasks run in the real-time kernel. Worst case time is 15 microseconds between giving an interrupt signal and starting of the real-time handler.

MiniRTL, a tiny implementation of RTLinux, runs on 486 machines. This implementation is targeted towards PC/104 boards.

8.4 Handheld Operating Systems

Handheld computers are becoming very popular as their capabilities are increasing day by day. Handheld computers integrated with mobile phones are called smart phones. The smart phones support data, voice and video services. The important requirements for a mobile operating system are:

- To keep the cost of the handheld computer low, small footprint of the operating system is required. A footprint of 64 KB to 2MB for the OS would be attractive. Generally, low-cost handheld computers will have 2 to 4 MB of ROM and 2 to 16 MB of RAM.
- The operating system should have support for soft real-time performance. For audio and video applications, timing constraints are imposed though the deadlines are not critical.
- TCP/IP stack needs to be integrated along with the operating system.
- Communication protocol stacks for Infrared, Bluetooth, IEEE 802.11 interfaces need to be integrated.
- There should be support for data synchronization. A special utility is required using which the data, in say, the desktop computer and the handheld computer can be synchronized. For instance, when you connect your handheld computer to your desktop computer and run the data synchronization software, the address book information in both the computers should be made to contain the same addresses. Each handheld operating system vendor presently gives proprietary software for data synchronization.

The popular handheld/mobile operating systems are:

- Palm OS
- Symbian OS
- Windows CE
- Windows CE.NET

An overview of these operating systems is given below.

Notes...

Data synchronization between two devices is presently done through proprietary protocols. SyncML, a markup language, has been standardized that uses standard protocols and a standard markup for data synchronization.

8.4.1 Palm OS

Palm OS (www.palmos.com) is perhaps the most popular handheld operating system. In addition to Palm Computing Inc.'s palmtops, many other vendors such as Sony, Hndspring use this OS in their handheld computer hardware. The OS will run on even low-end processors of 16 MHz to 33 MHz clock speed with 512 KB ROM, and 32 KB RAM. Bluetooth and IrDA protocol stacks are integrated into the OS to provide wireless interfaces to the handheld computer.

Using Palm OS SDK, application software can be developed for these devices. The applications can be developed in C or C++ or Java. The data between a handheld and a desktop computer can be synchronized using a utility called HotSync. The Palm Conduit Development Kit (CDK) is used to develop conduits (plug-ins to HotSync) in VC++ or VB or Java. Palm OS also provides a virtual file system API to develop a file system. Software development can be done on a desktop and tested on an emulator. The emulator can run on Windows, Unix/Linux, and Mac operating systems.

In Brief...

Palm OS is a popular handheld operating system. Application development on this OS can be done in C, C++ or Java. HotSync is used for data synchronization. Conduit Development Kit is used to develop applications for synchronization of data.

Though nearly 37% of the market share of handheld operating systems was held by Palm OS, now there is stiff competition from Symbian OS and Windows CE.

8.4.2 Symbian OS

In Brief...

Symbian OS is a popular handheld operating system which has been licensed to a number of mobile device manufacturers. Data synchronization is done through Symbian Connect.

Symbian OS (www.symbian.com) has a good market share of handheld operating systems. Licensees of Symbian OS include major mobile phone suppliers such as Ericsson, Kenwood, Motorola, Nokia, Panasonic, Psion, Sanyo, Siemens and Sony. Ericsson's R380 smart phone and Nokia's 9210 communicator are based on Symbian OS. The OS architecture supports both voice service and packet data services. Support for Bluetooth and IrDA is provided. The software development kit enables programmers develop applications in C++ or Java. Data synchronization is achieved with Symbian Connect.

8.4.3 Windows CE

In Brief...

Windows CE is a handheld operating system for Pocket PCs. As it uses a subset of Win32 APIs, application software can be developed in VB or VC++ using embedded visual tools.

Micorosft's Pocket PC hardware runs the Windows CE operating system. This OS runs on Intel x86 family processors as well as Alpha, NEC, Toshiba, PowerPC and Hitachi's Super H. The handheld computers of Casio, HP, NEC, Philips, Sharp etc. run this operating system. It requires 2 MB ROM.

Embedded Visual Tools are used to develop applications above Windows CE. The main attraction of these tools is that a subset of Win32 API is used to develop applications. Application software

developers well versed in VB or VC++ can develop applications for Windows CE without much effort or training and this is the main attraction of using this operating system. "Pocket" versions of Microsoft Office applications (Outlook, Excel, Word etc.) are available. For end users, this is a blessing as they need not learn a new user interface for running a word processor or a spreadsheet. ActiveSync is used to synchronize the data between the desktop and the handheld computer.

8.4.4 Windows CE.NET

Microsoft's Windows CE.NET is successor to Windows CE. It is a real-time embedded operating system and is now becoming a popular operating system for smart phones, PDAs, set top boxes, IP Phones, CD players, digital cameras, DVD players, etc. In tune with the other operating systems of Microsoft, this OS facilitates development of applications either using Microsoft Visual Studio.NET or using embedded Visual C++. Hence, developing new applications or porting existing applications is very easy. With support for TCP/IP, IP version 6, Bluetooth, 802.11 and Ethernet and Infrared interfaces as well as Pocket Internet Explorer, handheld computers and smart phones will look like desktop computers in your palm. The minimal kernel functionality requires just 200 KB. Average ISR latency is 2.8 microseconds on a 166 MHz Pentium processor.

In Brief...

Windows CE.NET is a hard real-time operating system that is being used in a number of consumer devices. Applications can be developed using Microsoft Visual Studio.NET.

Windows CE.NET 4.2 Emulation Edition and Evaluation Kit are freely downloadable for non-commercial purposes and evaluation. Without the target hardware, the application software can be tested using the emulators. Applications can be developed and tested on a desktop computer running Windows XP.

Microsoft provides nearly 2 million lines of source code under the Shared Source Program that enables modification to the source code to suit your application requirements.

Summary

In this chapter, selected commercial and open source operating systems are reviewed. These operating systems have the following commonalities: an Integrated Development Environment which contains all the development tools, POSIX compatibility, support for TCP/IP protocol stack and provision of Device Driver Kits. However, these operating systems differ in a number of aspects which include: processors supported, footprint, support for Java or lack of it, availability of Board Support Packages, support for different scheduling algorithms, number of tasks supported and assignment of priorities, support for priority inheritance and the licensing fee. An overview of the salient features of selected operating systems is presented. Embedded NT, Windows XP embedded and Embedded Linux are reviewed. An overview of some real-time operating systems viz., QNX Neutrino, VxWorks, MicroC/OSII and RTLinux is also given. Palm OS, Symbian OS, Windows CE and Windows CE.NET handheld/mobile operating system features are also discussed briefly.

Questions

1. List the various commercially available embedded operating systems and explain their features.
2. List the various open source embedded operating systems and explain their features.
3. List the various mobile/handheld operating systems and explain their features.
4. What is POSIX compatibility?

Exercises

1. List a few other embedded/real-time operating systems not covered in this chapter.
2. What are the issues involved in synchronization of data between the handheld computer and the desktop computer? Explore the standardization activities for data synchronization.
3. Develop a software package that measures the typing speed. The software has to run on a handheld computer such as the Pocket PC.
4. What type of an operating system you will use for the following applications:
   a. An Internet kiosk installed in a rural area.
   b. A navigation system used in an aircraft
   c. A navigation system used in a car
   d. A DVD player
   e. A desktop computer to be used as a process control system with hard real-time requirements
   f. An electronic toy helicopter