The task *B* can access the same resource using OSSemPend ( ) if it is waiting for that semaphore. The task *B* posts the semaphore using OSSemPost ( ) after completing the access to that resource.

Figure 7.5(a) shows the use of a semaphore between *A* and *B*. It shows the five sequential actions at five different times, T0, T1, T2, T3 and T4. Figure 7.5(b) shows the timing diagram of the tasks in the running states as a function of time. It marks the five sequential actions at five different times, T0, T1, T2, T3 and T4.

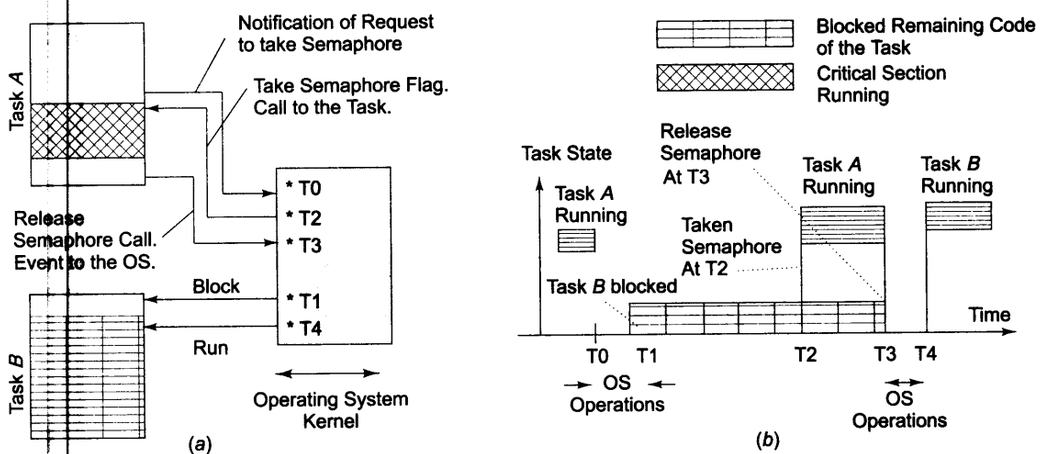


Fig. 7.5 (*a*) Use of a semaphore between tasks, *A* and *B*. It shows the five sequential actions at five different times, T0, T1, T2, T3 and T4 (*b*) Timing diagram of the tasks in the running states as a function of time. It marks the five sequential actions at five different times, T0, T1, T2, T3 and T4, and shows the use of a semaphore between tasks *A* and *B* by the operating system functions

An ISR can't be used to wait for the resource key since an ISR can just release the key. A task on the other hand can release the key as well accept the key or wait for taking the key. (row 7 Table 7.1)

Example 7.8

Consider an (Section 1.5.5) *Update_Time task*. When the task for updating time t on a system clock tick interrupt I_S starts, it has to notify that it is writing the t in a *time device* and that the t is changing. After the *Update_Time task* updates t information at the time device on I_S, it has to notify to the *Read_Time* task to run a waiting section of the code to read t from the *time device*. After the *Read_Time* reads t, it has to notify to *Update_Time task* to make note of it.

Assume OSSemPost ( ) is an IPC function at OS for posting a semaphore and assume OSSemPend ( ) is another IPC function for waiting the semaphore. Let *supdateT* be the binary semaphore pending and posted at *Update_Time* and pending and posted at *Read_Time* section for reading t. Let *supdateT* initial value = 1. The following will be the codes.

```
static void Task_ Update_Time (void *taskPointer) {
.
```

```
while (1) {
.
OSSemPend (supdateT) /* Wait the semaphore supdateT. This means that when wait over, an OS function
    decrements supdateT in corresponding event control block and supdateT becomes 0 at T2. */
/* Codes for writing date and time into the time device. */
.
OSSemPost (supdateT) /* Post the semaphore supdateT. This means that OS function increments supdateT
    in corresponding event control block and supdateT becomes 1 at T3. */
.
    };
static void Task_ Read_Time (void *taskPointer) {
.
while (1) {
.
OSSemPend (supdateT) /* Wait for the semaphore supdateT. This means that task waits till supdateT is posted
    and becomes 1. When supdateT becomes 1 and the OS function then decrements supdateT in corresponding
    event control block and supdateT becomes 0 at T4. Task then runs further the following code*/
/* Code for reading the date and time at time device */
.
.
OSSemPost (supdateT) /* Post the semaphore supdateT. This means that OS function
increments supdateT in corresponding event control block. supdateT becomes 1. */
};
```

Mutex When a binary semaphore is used to at beginning and end of critical sections in two or more tasks such that at any instance only one section code can run, then the semaphore is called *mutex*. (mutex word is derived from mutually exclusive). Example 7.8 showed that Task_ *Update_Time* and Task_ *Read_Time* uses the semaphore *supdateT* as mutex and at any instance either code section in Task_ *Update_Time* or code section in Task_ *Read_Time* runs and has exclusive access to the time device date and time variables. Mutex helps in getting access to a file or network or printer by multiple tasks at distinct instances because at an instance only codes for one of the tasks will be run to send the data to the file or network or printer. Section 7.8.3 will discuss its help in sharing data between the tasks.

In certain OS, a semaphore as a resource key is called mutex when the semaphore takes care of priority inversion problem also. Section 7.8.5 explains the inversion problem. In certain OS, a semaphore as a resource key is called mutex even when the semaphore does not take care of the priority inversion problem. In certain OS, a semaphore as a resource key is called mutex and the option is provided to programmer for using priority inversion safe or without inversion safe mutex.

1. Semaphore provides a mechanism to let a section of the task code or a task wait till another task section finishes another set of codes such that these sections use a common resource, device or file or variable.
2. When a semaphore is waiting (for taking or accepting) by a task code, then that task has the access to the necessary resources when semaphore is 'given' (sent or posted), the resources unlock.
3. Semaphore can be used as a resource key. Resource key is one that permits the use of resources like CPU, memory or other functions or critical section codes.
4. Binary semaphore can be used as a mutex as well as event notifying flag.

7.7.3 Use of Multiple Semaphores for Synchronizing the Tasks

OS semaphore functions are provided for multitasking operations. Figure 7.6 shows an example of the use of two semaphores for synchronizing the tasks *I, J* and *M* and the tasks *J* and *L*. Example 7.9 gives another example in which *I, J, K* and *L* are synchronized to run sequentially.

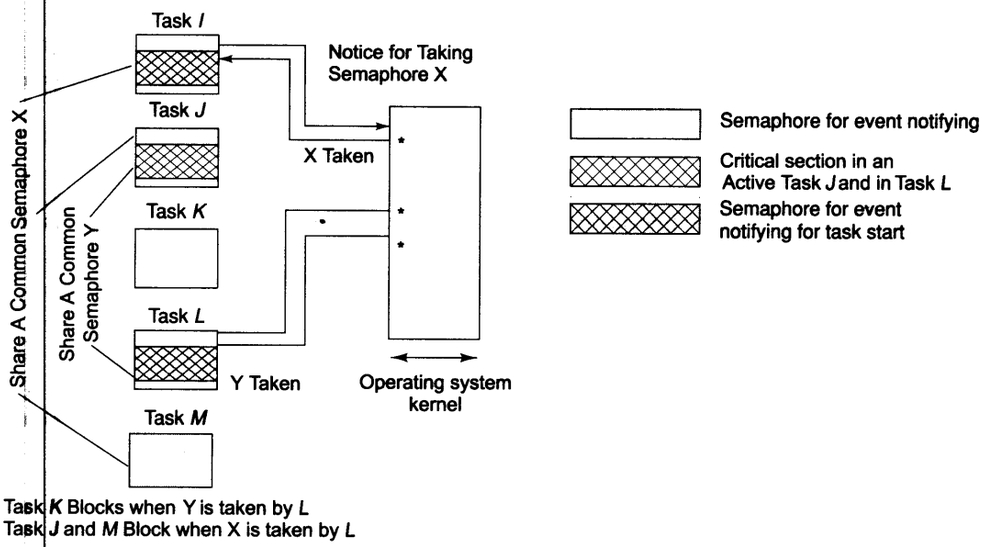


Fig. 7.6 An example of the use of two semaphores —one for synchronizing the tasks *I, J* and *M* and other for tasks *J* and *L*

Example 7.9

A semaphore can let one task run among many in the initiated into a list with *state* = '*ready*' or '*running*' run at an instance while others are waiting. Let the following be the codes.

Assume OSSemPost ( ) is an OS IPC function for posting a semaphore and assume OSSemPend ( ) is another OS IPC function for waiting for the semaphore. Let *sTask* be the binary semaphore pending and posted at each task to let another run. Let *sTask1* initially be 1 and *sTask2 sTask3* and *sTask4* initially be 0.

The following will be the codes and the first task *I* will run, then *J*, then *K*, then *L*, then *I* when at that instance *sTask1* = 1 and *sTask2 sTask3, sTask4* = 0.

```
static void Task_ I (void *taskPointer) {
.
while (1) {
OSSemPend (sTask1) /* wait for semaphore sTask1 and when wait over then an OS function decrements
    sTask1 in corresponding event control block and sTask1 becomes 0 */
/* Codes for Task_I */
.
```

```
OSSemPost (sTask2) /* Post the semaphore sTask2. This means that OS function increments
    sTask2 in corresponding event control block. sTask2 becomes 1 */
};
static void Task_ J (void *taskPointer) {
.
while (1) {
OSSemPend (sTask2) /* Wait for the semaphore sTask2. This means that task waits till sTask2is posted
    and becomes 1. When sTask2 becomes 1 and the OS function is to decrements sTask2 in corresponding
    event control block. sTask2 becomes 0. Task then runs further the following code*/
/* Code for Task J */
.
.
OSSemPost (sTask3) /* Post the semaphore sTask3. This means that OS function increments sTask3 in
    corresponding event control block. sTask3 becomes 1. */
};
static void Task_ K (void *taskPointer) {
.
while (1) {
OSSemPend (sTask3) /* Wait for the semaphore sTask3. This means that task waits till sTask3 is posted
    and becomes 1. When sTask3 becomes 1 and the OS function is to decrements sTask3 in corresponding
    event control block. sTask3 becomes 0. Task then runs further the following code*/
/* Code for Task K */
.
.
OSSemPost (sTask4) /* Post the semaphore sTask4. This means that OS function increments sTask4 in
    corresponding event control block. sTask4 becomes 1. */
};
static void Task_ L (void *taskPointer) {
.
while (1) {
OSSemPend (sTask4) /* Wait for the semaphore sTask4. This means that task waits till sTask4 is posted
    and becomes 1. When sTask4 becomes 1 and the OS function is to decrements sTask3 in
    corresponding event control block. sTask4 becomes 0. Task then runs further the following code*/
/* Code for Task J */
.
.
OSSemPost (sTask1) /* Post the semaphore sTask1. This means that OS function increments
    sTask1 in corresponding event control block. sTask1 becomes 1. */
};
```

For example, when a task *K* is to start running, it takes the semaphore sTask3. The OS blocks the tasks *I, J* and *L*. A task *L* waits for the release of the semaphore by *K*.

Number of tasks Waiting for the Same Semaphore An RTOS has the answer to the following: when a number of tasks has the same semaphore waiting then which of them takes the semaphore? In certain OS, a semaphore is given to the task of highest priority among the waiting tasks. In certain OS, a semaphore

is given to the longest waiting task in the FIFO mode. In certain OS, a semaphore is given to select an option and the option is provided for priority or FIFO mode. The task having priority if started, takes a semaphore first in case the priority option is selected. The task pending since a longer period takes a semaphore first in case the FIFO option is selected.

1. Multiple semaphores are used and different set of semaphores can share among different set of tasks.
2. Semaphore provides a mechanism to synchronize the task codes. Multiple semaphores can be used in multitasking system.

7.7.4 Counting Semaphores

An OS may provide for the counting semaphores. A counting semaphore can be an unsigned 8- or 16- or 32-bit integer. A value of counting semaphore controls the blocking or running of the codes of a task. The counting semaphore decrements each time it is taken. It increments when released by a task.

The value of counting semaphore at an instance reflects the initialized value minus the number of times it is taken plus the number of times released. The counting semaphore can be considered as the number of tokens present and the waiting task will not wait and run further if at least one token is present. The use of a semaphore is such that one of the task thus waits to execute the codes or waits for a resource till at least one token is found.

Assume that a task can send the stacks on a network into eight sequentially transmitting buffers. Each time the task runs it takes the semaphore and sends the stack into a buffer, which is next to the earlier one. Assume that a counting semaphore *scnt* is initialized = 8. After sending the data of the stack, the task takes the *scnt* and *scnt* decrements. When a task tries to take the *scnt* when it is 0, then the task blocks and cannot send stack into any buffer.

Example 7.10

Consider an ACVM (Section 1.10.2). Consider *Chocolate delivery task*. It cannot deliver more than the total number of chocolates, *total* loaded into the machine. Assume that a semCnt is initialized equal to the *total*. Each time, the new chocolates are loaded in the machine, semCnt increments by the number of new chocolates added. The *Chocolate delivery task* can be coded as follows.

```
static void Task_Deliver (void *taskPointer) {
.
while (1) { /* Start an infinite while-loop. */
/* Wait for an event indicated by an IPC from Task Read-Amount */
.
OSSemPend (semCnt) /* If chocolate available is true then the task takes the semaphore if
semCnt is 1 or > 1 (which means is not 0) and decrement the semCnt and continue remaining
operations */
.
};
```

Counting semaphore can be consider as an unsigned integer semaphore that can be 'taken' till its value = 0 and is initialized to a high value. It can also be 'given' a number of times.

7.7.5 P and V SEMAPHORES

An OS may provide for an efficient synchronization mechanism, called P and V semaphores in a standard, called POSIX 1003.1.b, an IEEE standard (POSIX stands for portable OS interfaces in Unix). OS semaphore functions P and V represent the semaphore by integer variables. A semaphore variable, apart from initialization, is accessed only through two standard atomic-operations: P and V. [P (for *wait* operation) is derived from a Dutch word 'Proberen', which means 'to test'. V (for *signal* notifying operation) is derived from the word 'Verhogen' which means 'to increment'.] (Atomic-operation is one, which can not be in parts.)

1. P semaphore function signals that the task requires a resource and if not available waits for it.
2. V semaphore function signals from the task to the OS that the resource is now free for the other users.

Consider P semaphore. It is a function, P (*&sem_1*) which, when called in a process, does the following operations using semaphore, *sem_1*.

1. /* Decrease the semaphore variable*/
sem_1 = *sem_1* –1;
2. /* If sem_1 is less than 0, send a message to OS by calling a function waitCallToOS. Control of the process transfers to OS, because less than 0 means that some other process has already executed P function on sem_1. Whenever there is return to the OS, it will be to step 1. */
if (*sem_1* < 0){waitCallToOS (*sem_1*);}.

Consider V semaphore. It is a function, V(*&sem_2*) which, when called in a process, does the following operations using semaphore, *sem_2*.

3. /* Increase the semaphore variable*/
sem_2 = *sem_2* + 1;
4. /* If sem_2 is less or equal to 0, send a message to OS by calling a function signalCallToOS. Control of the process transfers to OS, because < or = 0 means that some other process has already executed P function on sem_2. Whenever there is return to the OS, it will be to step 3. */
if (*sem_2* < = 0){signalCallToOS (*sem_2*);}

Use of P and V Semaphore Functions with a Signal or Notification Property P and V functions can represent a signalling or notifying variable, sem_s when used as shown in Example 7.11.

Example 7.11

Let sem_s be a semaphore variable. Let it function as a signal or notifying an event using variable sem_s. P and V semaphore functions are used in two processes, task 1 and task 2 as follows.

```
Process 1 (Task 1)
while (true) {
/* Codes */
.
.
.
.
V (&sem_s);
```

```
Process 2 (Task 2)
while (true) {
/* Codes */
.
P (&sem_s);
/* The following codes will execute only when
sem_s is not less than 0. */
.
```

```
/* Continue Process 1 if sem_s is not equal to 0
or not less than 0. It means that no process is
executing at present. */
.
};
```

```
.
.
};
```

Use of P and V Semaphore Functions with a Mutex Property P and V functions can represent a mutex semaphore variable, sem_m when used as explained in Example 7.12.

Example 7.12

Let sem_1 and sem_2 be the same variable, *sem_m*. The latter functions as a mutex, as follows, when P and V semaphore functions are used in two processes, task 1 and task 2.

```
Process 1 (Task 1)
while (true) {
/* Codes before a critical region*/
.
.
.
/* Enter Process 1 Critical region codes*/
P (&sem_m);
/* The following codes will execute only when
sem_m is not less than 0. */
.
.
.
/* Exit Process 1 critical region codes */
V (&sem_m);
/* Continue Process 1 if sem_m is not equal to 0
or not less than 0. It means that no process is
waiting and has executed P function using
sem_m at present. */
.
.
.
};
```

```
Process 2 (Task 2)
while (true) {
/* Codes before a critical region*/
.
.
.
/* Enter Process 2 Critical region codes*/
P (&sem_m);
/* The following codes will execute only when
sem_m is not less than 0. */
.
.
.
/* Exit Process 2 critical region codes */
V (&sem_m);
/* Continue Process 2 if sem_m is not equal to 0
or not less than 0. It means that no process is waiting
and executed P function using sem_m at present.
*/
.
.
.
};
```

The same variable, sem_m, is shared between process 1 and process 2. Its use is in making both processes gain mutually exclusive access to the resource (CPU). Either process 1 runs after executing P or process 2 runs after executing P. Also, either process 1 runs after executing V or process 2 runs after executing V.

Figure 7.7(a) shows the use of P and V semaphores and the task *I*, task *J* and the scheduler. When a task takes a semaphore P, if sem_m = 'true' (=1) earlier then it becomes 'false' (=0) and task run continues as sem_m is not less than 0. When a task executes V, if sem_m was 'false' earlier then it sets to 'true' and the task continues running, else the task blocks and waits for the execution of another task. Figure 7.7(b) shows the PC assignments to a process or function when using P and V semaphores.

Use of P–V Semaphore Functions with a Counting Semaphore Property Let there be a process (task *c*). The P function decrements the count and the function increments the counts. The P function operates on a counting semaphore, sem_c1 as shown in Example 7.13.

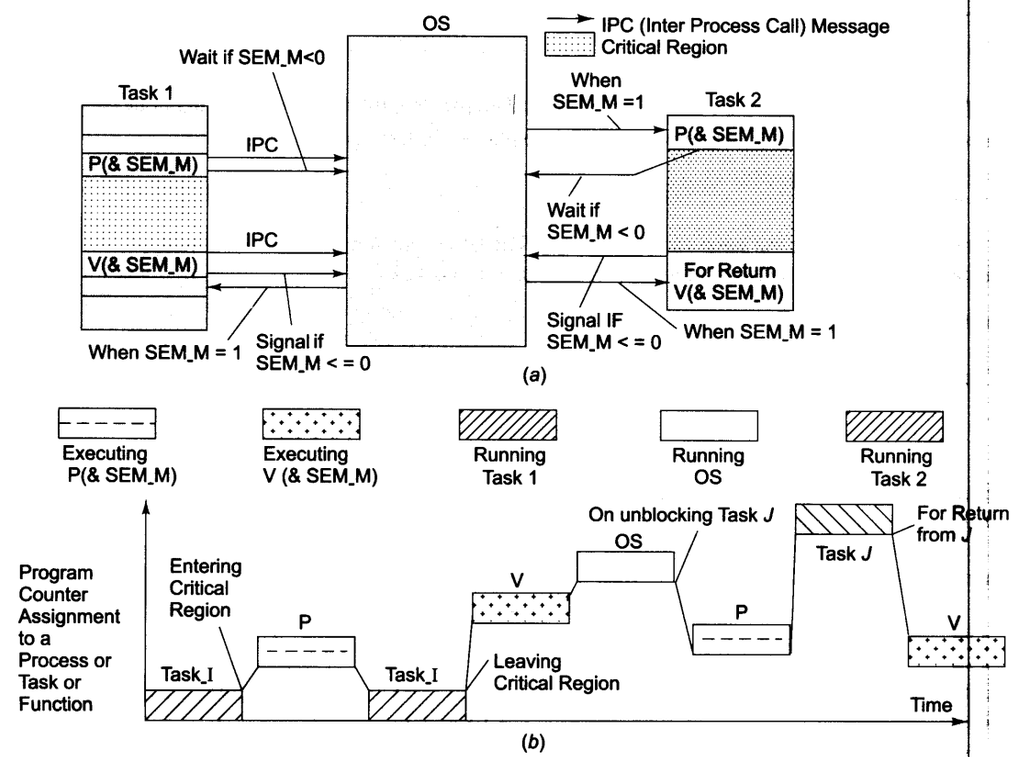


Fig. 7.7 (*a*) Use of P and V semaphores at a task 1, and at another task 2 and at a scheduler (*b*) The program counter assignments to a process or function when using P and V semaphores

Example 7.13

Assume a processes using P semaphore functions in task, task_c. Let *sem_c1* be a counting semaphore variable and represent the number of empty places created by the process c. P functions operate on these and reduces the number of empty places as follows:

```
Process c (Task_c)
    while (true) {
    /* Codes on entering a producer region*/
    .
    .
```

```
/* After exiting the producer Process 3 region codes */
P (&sem_c1);
/* Continue Process c if sem_c1 is not less than 0. */
.
.
};
```

Use of P and V Semaphore Functions with a Counting Semaphore Property for Bounded Buffer Problem Solution Let a task generate the outputs for use by another task. A task can have a separate counting semaphore.

Consider three examples:

(i) a task transmits bytes to an I/O stream for filling the available places at the stream;

(ii) a process '*writes*' an I/O stream to a printer buffer and,

(iii) a task of producing chocolates is being performed.

In example (i) another task reads the I/O stream bytes from the filled places and creates empty places.

In example (ii), from the print buffer an I/O stream prints after a buffer-read and after the printing, more empty places are created.

In example (iii) again, as consumer consumes the chocolates produced more empty places (to stock the produced chocolates) are created.

A task blockage operational problem is commonly called producer–consumer problem. A task cannot transmit to the I/O stream if there are no empty places in the stream. The task cannot write from the memory at the print buffer if there are no empty places at the print buffer. The *producer cannot produce if there are no empty places at the consumer stock.*

A classic program for synchronization is called the *producer–consumer problem program.* It is also called *bounded buffer problem program.* Here, one or more producers (task or thread processes) create data outputs that are then processed by one or more consumers (tasks or processes). The data outputs from the producers are passed for processing by the consumers using some type of IPC (Section 7.8) that uses a shared memory and counting semaphores (or message queues or mailboxes).

Let there be two processes (tasks 3 and 4). The P and V functions operate on two shared counting semaphores, sem_c1 and sem_c2, as in Example 7.14.

Example 7.14

Assume two processes using P and V semaphore functions and two tasks, tasks 3 and 4. Let *sem_c1* and *sem_c2* be two counting semaphore variables and represent the number of filled places created by the process 3 and a number of empty places created by process 4, respectively. P and V functions operate on these as follows.

```
Process 3 (Task 3)
while (true) {
```

```
Process 4 (Task 4)
while (true) {
```

```
/* Codes before a producer region*/
.
.
.
/* Enter Process 3 Producing region codes*/
P (&sem_c2);
/* The following codes will execute only when
sem_c2 (number of empty places)is not less
than 0. */
.
.
.
/* Exit Process 3 region codes */
V (&sem_c1);
/* Continue Process 3 if sem_c1 is not equal to 0
or not less than 0. */
.
.
.
};
```

```
/* Codes before a consumer region*/
.
.
.
/* Enter Process 4 Consuming region codes*/
P (&sem_c1);
/* The following codes will execute only when
sem_c1 (number of filled places) is not less
than 0. */
.
.
.
/* Exit Process 4 region codes */
V (&sem_c2);
/* Continue Process 4 if sem_c2 is not equal to 0
or not less than 0. It means that filled places are
still available at present. */
.
.
.
};
```

Two semaphores, sem_c1 and sem_c2 are shared between processes 3 and 4. When process 3 executes, it first reduces the number of empty places at process 4. When process 3 completes production, it increases the number of filled places at process 3. When process 4 consumes, it first reduces the number of filled places at process 3. When process 4 completes consumption, it increases the number of empty places at process 4. Either process 3 produces output after executing P, or process 4 consumes (uses) inputs after executing P. Also either process 3 proceeds after executing V or process 4 proceeds after executing V.

P and V semaphore functions are in POSIX 100.3b, an IEEE-accepted standard for the IPCs. They can be used as an event signalling, as a mutex, as a counting semaphore and the semaphores for the bounded buffer problem solution.

7.8 SHARED DATA

7.8.1 Problem of Sharing Data by Multiple Tasks and Routines

The shared data problem can be explained as follows: Assume that several functions (or ISRs or tasks) share a variable. Let us assume that at an instant the value of that variable operates and during its operations, only a part of the operation is completed and a part remains incomplete. At that moment, let us assume that there is an interrupt. Now, assume that there is another function. It also shares the same variable. The value of the variable *may* differ from the one expected if the earlier operation had been completed. The incomplete operation can occur as follows.

Suppose a variable is of 128 bits and the processor is of 32 bits. The operations on the variable will be using four 32-bit ALU operations in order to use the 32-bit ALU of this processor. Atomic operation is one, which

cannot be subdivided into the suboperations or none of the suboperations can be left incomplete and no other operation can start before all suboperations are complete. Now, assume that the 128-bit operation on the variable is non-atomic. This means that the operation can be interrupted before all the four operations are completed. An interrupt can occur at the end of each 32-bit ALU operation, not necessarily at the end of the 128-bit operation. Therefore, the called ISR or another function can use the incompletely operated variable or can change that variable when it shares with another function. On return, new values of that variable will be found and the incomplete operations will be performed on that new 128-bit variable in place of the old one.

Example 7.15

(*a*) Consider x, a 128-bit variable, b_{127}.....b_0. Assume that the operation OP_{sl} is shift left by 2 bits to multiply it by 4 and find $y = 4 \times x$. Let OP_{sl} done non-atomically in four suboperations, OPA_{sl}, OPB_{sl}, OPC_{sl} and OPD_{sl} for b_{31}.....b_0 , b_{63}.....b_{32} , b_{95}.....b_{64} and b_{127}.....b_{96}, respectively. Assuming at an instance that suboperations OPA_{sl}, OPB_{sl} and OPC_{sl} are completed and OPD_{sl} remained incomplete. Now interrupt *I* occurs at that instance. The *I* calls some function which uses x if x is the global variable. It modifies x = b'_{127}.....b'_0. On return from interrupt, as OPD_{sl} is not complete, OPD_{sl} operates on b'_{127}.....b'_{96} not on b_{127} ... b_{96}.

(*b*) Consider date d and time t. Let d and t be taken in the program as global variables. Assume that a thread *Update_Time_Date* is updating t and d information on system clock tick interrupt I_S. The thread *Display_Time_Date* in Example 7.2 displays that t and d information.

1. Assume that when *Update_Time_Date* ran t = 23:59:59 and date d = 17 July 2007.
2. The *Display_Time_Date* gets interrupted and assumes that display d and operation t are non-atomic. Display of d was completed but display of t was incomplete when interrupt I_S occurred.
3. After a while, the t changes to t = 00:00:00 and d = 18 July 2007 when the thread *Update_Time_Date* runs. The display will show t = 00:00:00 and date d = 17 July 2007 on the return from interrupt and re-start of blocked thread *Display_Time_Date*.

The use of shared variables d and t in two threads *Update_Time_Date* and *Display_Time_Date* causes the display error.

7.8.2 Shared Data Problem Solutions

One solution is the use of atomic operation for solving the shared data problem. We need atomic operations because of the following.

1. An interrupt can occur at the end of an instruction cycle, not at the end of a high-level instruction.
2. A DMA operation can occur at the end of a machine cycle itself and a compiler or program may not taking these atomic-level details into account (DMA operation means direct memory access, an IO device loads into the memory using the system address and data buses when CPU is not performing any bus-operation).
3. A context switch operation can occur at the end of an instruction for calling a new function, cycle itself and a compiler or program may not be taking into account these atomic-level details.

The following are the steps that, if used together, almost eliminate a likely bug in the program because of the shared data problem:

1. *Use modifier volatile* with a declaration for a variable that returns from the interrupt. This declaration warns the compiler that certain variables can modify because the ISR does not consider the fact that the variable is also shared with a calling function.

2. *Use reentrant functions* with atomic instructions in that part of a function that needs its complete execution before it can be interrupted. This part is called the critical section. For example, the suboperations of shifting of x in Example 7.15 are in critical section.
3. *Put a shared variable in a circular queue.* A function that requires the value of this variable always deletes (takes) it from the queue *front*, and another function, which inserts (writes) the value of this variable, always does so at the queue *back*. Now a problem can occur in case there are a large number of functions that post the values into and take the values but the maximum required queue size is not provided.

Example 7.16

Example shows how shared data problem get solved by using queue. Consider the Example 7.15(b). Assume that variables t for time and d for date are shared variables and there is a queue Q_{TD} into which a thread inserts the shared variables and another thread deletes these variables. [*Note:* Insertion into a queue means writing a value at the queue tail and then changing the pointer to the queue tail for the next insertion. Deletion from a queue means reading the value
from the queue head and then changing the pointer to the queue head for the next read.]

1. Assume that when *Update_Time_Date* runs the t = 23:59:59 and date d = 17 July 2007 and inserts these in a queue Q_{TD}.
2. The *Display_Time_Date* reads t and d from Q_{TD} and displays. When the thread gets interrupted after reading d, display of d, the Q_{TD} is still holding t when interrupt I_S occurs. Display of t will complete on return from the interrupt.
3. After a while, the t changes to t = 00:00:00 and date d = 18 July 2007 and the thread *Update_Time_Date* runs and inserts the new values of t and d at the back of the earlier values in Q_{TD}. The display will show d = 17 July 2007 and t = 23:59:59 and on return from the interrupt and in the next cycle run the thread will show d = 18 July 2007 and t = 00:00:00.

The use of queue for the shared variables d and t in two threads when *Update_Time_Date* inserts these into the queue and *Display_Time_Date* deletes these from the Q_{TD} causes no display error.

4. *Disable the interrupts before a critical section starts executing* and *enable* the interrupts on its completion. It is a powerful but drastic option. An interrupt, even if of higher priority than the present critical function, gets disabled. Advantage of it is that the semaphore functions have greater computational overhead than disabling of the interrupt. The difficulty with this option is that it increases in the interrupt latency period for all the tasks. The latency increases by the time taken in executing the codes of the section. A deadline may be missed for an interrupt service by that task which does not share the critical section or the resource.

As an alternative to disabling interrupts, Section 7.7.2 described using of semaphores for the shared data problem. A software designer must not use the drastic option of disabling interrupts in all the critical sections. [*Note:* In the OS for automobile applications, the disabling of interrupts is used before entering any critical section to avert any unintended action because of improper use of semaphores.] Another alternative is use of lock or spin-lock functions in a scheduler. (Section 7.11)

The use of disabling the switching of task from one to another and other steps and use of semaphores (Section 7.7) must eliminate the shared data problem completely from a multitasking, multi-ISRs and multiple shared variable cases. Each of the step has its own inherent benefits in solving the problem. A programmer must utilize the various steps optimally suited to solve the problem.

Shared data problem can arise in a system when another higher priority task finishes an operation and modifies the data or a variable. Using reentrant functions, disabling interrupt mechanism, using semaphores and IPCs such as mailbox and queue are the solutions which are used for taking care of shared data problem.

7.8.3 Applications of Semaphores and Shared Data Problem

Use of mutex facilitates mutually exclusive access by two or more processes to the resource (CPU). The same variable, *sem_m*, is shared between the various processes. Let process 1 and process 2 share *sem_m* and its initial value = 1.

1. Process 1 proceeds after *sem_m* decreases and equals 0 and gets the exclusive access to the CPU.
2. Process 1 ends after *sem_m* increases and equals 1; process 2 now gets exclusive access to the CPU.
3. Process 2 proceeds after *sem_m* decreases and equals 0 and gets exclusive access to CPU.
4. Process 2 ends after *sem_m* increases and equals 1; process 1 now gets the exclusive access to the CPU.

The *sem_m* is like a resource key and shared data within the processes 1 and 2 is the resource. Whosoever first decreases it to 0 at the start gets the access to and prevents the other to run with whom this key shares.

Mutex is a semaphore that provides at an instance two tasks mutually exclusive access to resources and is used in solving shared data problem.

7.8.4 Elimination of Shared Data Problem

The use of semaphores does not eliminate the shared data problem completely. Software designers may not take the drastic option of disabling interrupts in all the critical sections by using semaphores. When using semaphores, the OS does not disable the interrupts. Alternatively, task-switching flags can be used (Section 8.10.3). The following problems that can arise when using semaphores.

1. Sharing of two semaphores creates a deadlock problem (Refer to Section 7.8.5).
2. Suppose the semaphore taken is never released? There should therefore be some time-out mechanism after which the error message is generated or an appropriate action taken. There is some degree of similarity with the watchdog timer action on a time-out. A watchdog timer on timeout resets the processor. Here, after the time out, the OS reports an error and runs an error-handling function. Without a time out, an ISR worst-case latency may exceed the deadline.
3. A semaphore not taken, and another task uses a shared variable.
4. What happens when a train takes a signal for a wrong track? When using the multiple semaphores, if an unintended task takes the semaphore, it creates a problem.
5. There may be priority inversion problem (Refer to Section 7.8.5).

7.8.5 Priority Inversion Problem and Deadlock Situations

Let the priorities of tasks be in an order such that task *I* is of the highest priority, task *J* is of a lower and task *K* of the lowest. Assume that only tasks *I* and *K* share the data and *J* does not share data with *K*. Also let tasks *I* and *K* alone share a semaphore and not *J*. Why do only a few tasks share a semaphore? Can't all share a semaphore? The reason is that the worst-case latency becomes too high and may exceed the deadline if all tasks are blocked when one task takes a semaphore. The worst-case latency will be small only if the time taken by the tasks that share the resources is relevant. Now consider the following situation.

At an instant t_0, suppose task *K* takes a semaphore, the OS does not block task *J* and blocks task *I*. This happens because only tasks *I* and *K* share the data and *J* does not. Consider the problem that now arises on selective sharing between *K* and *I*. At the next instant t_1, let task *K* become ready to run first on an interrupt. Now, assume that at the next instant t_2, task *I* becomes ready on an interrupt. At this instant, *K* is in the critical section. Therefore, task *I* cannot start at this instant due to *K* being in the critical region. Now, if at next instant t_3, some action (event) causes the unblocked higher than the *K* priority task *J* to run. After instant t_3, running task *J* does not allow the highest priority task *I* to run. This is because even though *K* is not running and thus

unable to release the semaphore that it shares with *I*. Further, the code of task *J* may be such that even when the semaphore is released by task *K*, it may not let *I* run (*J* runs the codes as if it is in critical section all the time). The *J* action is now as if *J* has higher priority than *I*. This is because *K*, after entering the critical section and taking the semaphore when the OS is letting *J* run, but did not share the priority information about *I*—that task *I* is of higher priority than *J*. The priority information of another higher-priority task *I* should have also been inherited by *K* temporarily, if *K* waits for *I* but *J* does not and *J* runs when *K* has still not finished the critical section codes. This did not happen because the given OS design was such that it did not provide for temporary priority inheritance in such situations.

This situation is also called *priority inversion problem.* An OS must provide for a solution for the priority inversion problem. Some OSes provide for priority inheritance in these situations and thus priority inheritance problem does not occur when using them. Refer to Section 7.7.2 for use of a mutex for resources sharing. A mutex should be a mutually exclusive Boolean function, by using which the critical section is protected from interruption in such a way that the problem of priority inversion does not arise. Mutex is automatically provided in certain RTOS so that the priority inversion problem does not arise. Mutex use may also be just analogous to a semaphore defined in Section 7.7.2 in another RTOS and which does not solve the priority inversion problem.

Consider another problem. Assume the following situation.

1. Let the priorities of tasks be such that task *H* is of highest priority. Then task *I* has a lower priority and task *J* has the lowest.
2. There are two semaphores, *SemTok1* and *SemTok2*. This is because the tasks *I* and *H* have a shared resource through *SemTok1* only. Tasks *I* and *J* have two shared resources through two semaphores, *SemTok1* and *SemTok2*.
3. Let *J* interrupt at an instant t_0 and first take both the semaphores *SemTok1* and *SemTok2* and run.

Assume that at a next instant t_1, being now of a higher priority, the task *H* interrupts the tasks *I* and *J* after it takes the semaphore *SemTok1*, and thus blocks both *I* and *J*. In-between the time interval t_0 and t_1, the *SemTok1* was released but *SemTok2* was not released during the run of task *J*. But the latter did not matter as the tasks *I* and *J* do not share *SemTok2*. At an instant t_2, if *H* now releases the *SemTok1*, allows the task *I* to take it. Even then it cannot run because it is also waiting for task *J* to release the *SemTok2*. The task *J* is waiting at a next instant t_3, for either *H* or *I* to release the *SemTok1 because it needs this to again enter a critical section.* Neither task *I* can run after instant t_3 nor task *J*. There is a circular dependency established between *I* and *J*.

This situation is also called a *deadlock* situation. On the interrupt by *H*, the task *J*, before exiting from the running state, should have been put in queue-front so that later on, it should first take *SemTok1*, and the task *I* put in queue next for the same token, the deadlock would not have occurred (refer to Section 7.12 for queuing of messages).

The use of mutex solves the deadlock problem in certain OSes. Its use may be just analogous to a semaphore defined in Section 7.2.1. In other OSes, the mutex use may not solve the deadlock situation.

Priority becomes inverted and deadlock (circular dependency) develops in certain situations when using semaphores. Certain OSes provide the solution to this problem of semaphore use by ensuring that these situations do not arise during the concurrent processing of multitasking operations.

7.9 INTERPROCESS COMMUNICATION

Is it possible to send through the kernel an output data (a message of a known size with or without a header) for processing by another task? One way is the use of global variables. Use of these now creates two problems.

One is the shared data problem (Section 7.8). The other problem is that the global variables do not prevent (encapsulate) a message from being accessed by other tasks.

IPCs in a multiprocessor system are used to generate information about certain sets of computations finishing on one processor and to let the other processors waiting for finishing those computations take note of the information.

IPC means that a process (scheduler, task or ISR) generates some information by signal or value or generates an output so that it allows another process to take note or use it through the kernel functions for the IPCs. IPCs in a multitasking system are used to set or reset a signal or token or flag or generate message from the certain sets of computations finishing on one task and to let the other tasks take note of the signal or get the message.

OSes provide the software programmer the following IPC functions, which can be used.

1. Signals
2. Semaphores *as token or mutex* or counting semaphores for the intertask communication between tasks sharing a common buffer or operations
3. Queues and mailboxes
4. Pipes and sockets
5. Remote procedure calls (RPCs) for distributed processes.

Section 7.7 described two IPC functions, OSSemPend ( ) and OSSemPost ( ) and use of semaphores for the IPCs. The following shows an application for the printing from buffer by a task.

Example 7.17

Consider using a mutex semaphore in the tasks, which needs to use the *print* for the buffer data from one task at an instance. A task runs a *print* function, which reads from a print buffer and prints. The kernel should let the other tasks share this task. The *print* task can be shared among the multiple tasks, which use the mutex semaphore IPCs in their critical sections.

When the printer buffer becomes available for new data, an IPC from the *print* task is generated and the kernel takes note of it. Other tasks then take note of it. A task takes note of it by the OSSemPend ( ) function of the kernel used at the beginning of the critical section and the task gets mutually exclusive access to the section to send messages into the print buffer by using the OSSemPost ( ) function of the kernel at the end of the section (Sections 7.7.2, 7.8.3 and 7.11.1).

Consider Example 7.18, which shows use of the functions for semaphore and mailbox IPCs (semaphore and mailbox IPC functions will be described in detail in Sections 7.11 and 7.13).

Example 7.18

Consider a mobile phone device (Section 1.10.5) *Update_Time task* (Example 7.8). Assume that there is a task, *Task_Display* for a multiline display of outputs which displays current time on the last line. When the multiline display task finishes the display of the last but one line, an IPC semaphore *supdateTD* from the *display* task is generated and the kernel takes note of it. The task—continuously updating time—then can take the *supdateTD* and generate an IPC as a mailbox output for the current time and date.

When the task for updating time t on a signal posted on system clock-tick interrupt I_S starts, on taking the *supdateTD* posted by the Task_Display, it can write the td into a *mailbox* using an IPC for posting mailbox message, *timeDate*.

Assume OSSemPost ( ) is an OS IPC function for posting a semaphore and assume OSSemPend ( ) is another OS IPC function for waiting for the semaphore. Let *supdateTD* be the binary semaphore posted at *Task_Display* and pending at *Task_Display* section for displaying t and d. Let *supdateTD* initial value = 1.

Let the IPC function be OSMboxPost ( ) for posting the mailbox IPC message from *Update_Time task* and OSMboxPend ( ) for waiting for the mailbox IPC at *Task_Display* section. Let timeDate initial value = null.

The following will be the codes:

```
static void Task_Display (void *taskPointer) {
.
while (1) {
.
/* IPC for waiting time and date in the mailbox */
TimeDateMsg = OSMboxPend (timeDate) /* Wait for the mailbox message timeDate. The timeDate becomes
    null after the mailbox is posted time and date by Task_ Update_Time and TimeDateMsg equals the
    updated time and date */
/* Code for display TimeDateMsg Time: hr:mm Date: month:date */
.
/* IPC for requesting TimeDate */
OSSemPost (supdateTD) /* Post for the semaphore supdateTD. supdateTD becomes 1
};
static void Task_ Update_Time (void *taskPointer) {
.
while (1) { /* wait for system clock inter interrupt signal or semaphore notification from ISR of I_s */
OSSemPend (supdateTD) /* Wait the semaphore supdateTD. This means that OS function decrements
    supdateTD in corresponding event control block. supdateT becomes 0 */
/* Codes for updating time and date as per the number of clock interrupts received so far */
/* Codes for writing into the mailbox */
OSMboxPost (timeDate) /* Post for the mailbox message and timeDate, which equaled null
    now equals newupdated time and date*/
.
    };
```

The need for IPC and thus intertask communications also arises in a client-server network.

IPC means that a process (scheduler or task or ISR) generates some information by setting or resetting a token or value, or generates an output so that it lets another process take note or use it under the control of OS.

7.10 SIGNAL FUNCTION

One way for messaging is to use an OS function *signal* ( ). It is provided in Unix, Linux and several RTOSes. Unix and Linux OSes use *signals* profusely and have 31 different types of *signals* for the various events. Section 9.3 will describe *signal* in VxWorks RTOS. Just a hardware mechanism sends the interrupt to the OS, task (or process) or the OS itself sends *signal*. The task or process sending the signal uses a function signal ( ) having an integer number n in the argument. A signal is function, which executes a software interrupt instruction INT n or SWI n.

A 'signal' provides the shortest communication. The *signal* ( ) sends a output n for a process, which enables the OS to unmask a signal mask of a process or task as per the n. The task is called signal handler and has coding similar to the ones in an ISR. The handler runs in a way similar to a highest priority ISR. An ISR runs on an hardware interrupt provided that the interrupt is not masked. The handler runs on the signal provided that the signal is not masked.

The signal ( ) forces the OS to first run a signalled process or task called signal handler. When there is return from the signalled or forced task or process, the process, which sent the signal, runs the codes as happens on a return from an ISR. A signal mask is the software equivalent of the flag at a register that sets on masking a hardware interrupt. Unless masked by a *signal* mask, the *signal* allows the execution of the signal-handling function and allows the handler to run just as a hardware interrupt allows the execution of an ISR.

An integer number (for example n) represents each signal and that number associates a function (or process or task) signal handler, an equivalent of the ISR. The signal handler has a function called whenever a process communicates that number.

A signal handler is not called directly by a code. When the signal is sent from a process, OS interrupts the process execution and calls the function for signal handling. On return from the signal handler, the process continues as before.

For example, signal (5). The signal mask of signal handler 5 is reset. The signal handler and connect function associate the number 5. The function represented by number 5 is forced run by the signal handler.

An advantage of using it is that unlike semaphores it takes the shortest possible CPU time to force a handler to run. The *signals* are the interrupts that can be used as the IPC functions of synchronizing.

A *signal* is unlike the semaphore. The semaphore has use as a token or resource key to let another *task process* block, or which locks a resource to a particular *task process* for a given section of the codes. A signal is just an interrupt that is shared and used by another *interrupt-servicing process.* A *signal* raised by one process forces another process (signal handler) to interrupt and catch that *signal* in case the *signal* is not masked (use of that signal handler is not disabled). Signals are to be handled only for forcing the run of very high priority processes as it may disrupt the usual schedule and priority inheritance mechanism. It may also cause reentrancy problems.

An important application of the *signals* is to handle *exceptions*. (An exception is a process that is executed on a specific reported run-time condition.) A *signal* reports an error (called 'exception') during the running of a task and then lets the scheduler initiate an error-handling process or function or task. An error-handling signal handler handles the different error login of other task. The device driver functions also use the signals to call the handlers (ISRs).

The following are the signal related IPC functions, which are generally not provided in the RTOS such as µCOS-II and provided in RTOS such as VxWorks or OS such as Unix and Linux.

1. *SigHandler* ( ) to create a signal handler corresponding to a signal identified by the signal number and define a pointer to the signal context. The signal context saves the registers on signal.
2. *Connect* an interrupt vector to a signal number, with signalled handler function and signal-handler arguments. The interrupt vector provides the PC value for the signal-handler function address.
3. A function *signal ( )* to send a signal identified by a number in the argument to a task.
4. *Mask* the signal.
5. *Unmask* the signal.
6. *Ignore* the signal.

1. The simplest IPC for messaging from processes that forces a handler function to run (provided unmasked) is the use of 'signal'.
2. A 'signal' provides the shortest communication. Signals are used for initiating exceptions and error-handling processes.
3. IPC function for a signal is signal (n) for signalling signal-handler associated with n to run if not masked.

7.11 SEMAPHORE FUNCTIONS

The OS provides for semaphore as notice or token for an event occurrence. Semaphore facilitates IPC for notifying (through a scheduler) a waiting task section change to the running state upon event at presently running code section at an ISR or task. A semaphore as binary semaphore is a token or resource key (Sections 7.7.1 and 7.7.2). The OS also provides for mutex access to a set of codes in a task (or thread or process) (Sections 7.7.2 and 7.8.3). The use of mutex is such that the priority inversion problem is not solved in some OSes while it is solved in other OSes. The OS also provides for counting semaphores. The OS may provide for POSIX standard P and V semaphores which can be used for notifying event occurrence or as mutex or for counting. The timeout can be defined in the argument with wait function for the semaphore IPC. The error pointer can also be defined in the arguments for semaphore IPC functions.

The following are the functions, which are generally provided in an OS, for example, μCOS-II for the semaphores.

1. *OSSemCreate*, a semaphore function to create the semaphore in an event control block (ECB). Initialize it with an initial value.
2. *OSSemPost*, a function which sends the semaphore notification to ECB and its value increments on event occurrence (used in ISRs as well as tasks).
3. *OSSemPend*, a function, which waits the semaphore from an event, and its value decrements on taking note of that event occurrence (used in tasks not in ISRs).
4. *OSSemAccept*, a function, which reads and returns the present semaphore value and if it shows occurrence of an event (by non-zero value) then it takes note of that and decrements that value (no wait; used in ISRs as well as tasks).
5. *OSSemQuery*, a function, which queries the semaphore for an event occurrence or non-occurrence by reading its value and returns the present semaphore value, and returns pointer to the data structure OSSemData. The semaphore value does not decrease. The OSSemData points to the present value and table of the tasks waiting for the semaphore (used in tasks).

An OS provides the IPC functions create, post, pend, accept and query for using semaphores. The time-out can be provided with 'pend' function arguments. A pointer to error-handling function can also be specified in the arguments.

7.11.1 Mutex, Lock and Spin Lock

An OS, using a mutex blocks a critical section in a task on taking the mutex by another task's critical section. Other task unlocks on releasing the mutex. The mutex wait by blocked task can be for a specified timeout.

There is a function in kernel called lock ( ). It locks a process to the resources till that process executes unlock ( ). A wait loop creates and when wait is over the other processes waiting for the lock starts. Use of lock ( ) and unlock ( ) involves little overhead compared to uses of OSSemPend ( ) and OSSemPost ( ) when using a mutex. Overhead means number of operations needed for blocking one process and starting another. However, *a resource of high priority should not lock the other processes by blocking an already running task in the following situation.* Suppose a task is running and a little time is left for its completion. The running time left for it is less compared with the time that would be taken in blocking it and context switching. There is an innovative concept of spin locking in certain OS schedulers. A *spin lock* is a powerful tool in the situation described before. [Refer to 'Multithreaded Programming with Java' by

Bil Lewis and Daniel J. Berg, Sun Microsystems Inc., 2000.] The scheduler locking process for a task *I* waits in a loop to cause the blocking of the running task first for a time interval t, then for (t - δt), then (t - 2δt) and so on. When this time interval spin-downs to 0, the task that requested the lock of the processor now unlocks the running task *I* and blocks it from further running. The request is now granted to task *J* to unlock and start running provided that task is of higher priority. *A spin lock does not let a running task to be blocked instantly, but first successively tries with or without decreasing the trial periods before finally blocking a task.* A spin-lock obviates need of context-switching by pre-emption and use of mutex function-calls to OS.

An OS provides the IPC functions for creating and accessing the resource using mutex for a process and to prevent the resource for the other processes. An OS may also provides for lock or spin-locks at the scheduler.

7.12 MESSAGE QUEUE FUNCTIONS

Some OSes do not distinguish, or make little distinction, between the use of queues, pipes and mailboxes during the message communication among processes, while other OSes regard the use of queues as different.

A message *queue* is an IPC with the following features.

1. An OS provides for inserting and deleting the message pointers or messages.
2. Each queue for the message or message-pointers needs initialization (creation) before using functions in kernel for the queue.
3. Each created queue has an ID.
4. Each queue has a user-definable size (upper limits for number of bytes).
5. When an OS call is to insert a message into the queue, the bytes are as per the pointed number of bytes. For example, for an integer or float variable as a pointer, there will be 4 bytes inserted per call. If the pointer is for an array of eight integers, then 32 bytes will be inserted into the queue. When a message-pointer is inserted into queue, the 4 bytes inserts, assuming 32-bit addresses.
6. When a queue becomes full, there is error handling function to handle that.

Figure 7.8(a) shows functions for the queues in the OS. Figure 7.8(b) shows a queue-message block with the messages or message-pointers. Two pointers, *QHEAD and *QTAIL are for queue head and tail memory locations.

The OS functions for a queue, for example, in μCOS-II, can be as follows:

1. OSQCreate, a function that creates a queue and initializes the queue.
2. OSQPost, a function that sends a message into the queue as per the queue tail pointer, it can be used by tasks as well as ISRs.
3. The OSQPend waits for a queue message at the queue and reads and deletes that when received (wait, used by tasks only, not used by ISRs).
4. OSQAccept deletes the present message at queue head after checking its presence yes or no and after the deletion the queue head pointer increments. (no wait; used by ISRs as well as tasks)
5. OSQFlush deletes the messages from queue head to tail. After the *flush* the queue head and tail points to QTop, which is the pointer at start of the queuing. (used by ISRs and tasks)
6. OSQQuery queries the queue message block but the message is not deleted. The function returns pointer to the message queue *Qhead if there are the messages in the queue or else returns NULL. It returns a pointer to the structure of the queue data structure for *QHEAD, number of queued messages, size and table of tasks waiting for the messages from the queue. (query is used in tasks)
7. OSQPostFront sends a message to front pointer, *QHEAD. Use of this function is made in the following situations. A message is urgent or is of higher priority than all the previously posted message into the queue (used in ISRs and tasks).

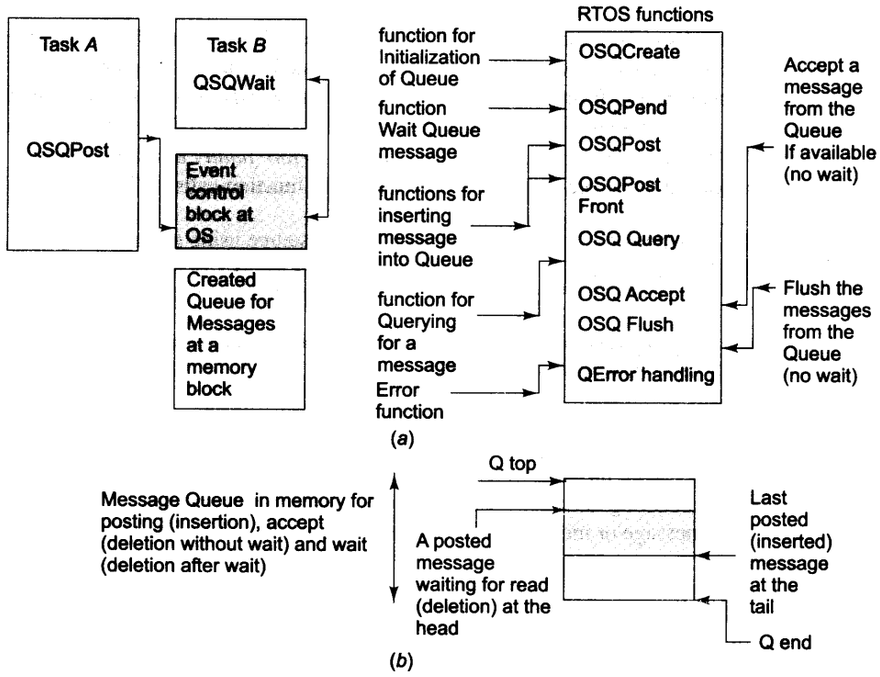


Fig. 7.8 (*a*) OS functions for queue and use of post and wait functions by Tasks *A* and *B* (*b*) Queue message block in memory

Example 7.19

Consider Orchestra Playing Robots (Example 1.10.7). Task_*Director_Output* puts the musical notes into the queue at conducting and directing robot. OSQEntries equals the number of queue entries and OSQSize equals the maximum number of notes that can be put into the queue.

```
static void Task_Director_Output (void *taskPointer) {
.
while (1) {
.
/* Codes for inserting musical notes into the queue */
for (OSQEntries = 0; OSQEntries < OSQSize; OSQEntries ++)
{OSQPost (QDirector, note)} /* Post for the Queue QDirector messages upto the OSQSize /*
.
};
static void Task_Player_Input (void *taskPointer) {
.
while (1) {
.
```

```
/* Codes for deleting notes from the queue */
for (OSQEntries = OSQSize; OSQEntries >0 ; OSQEntries --)
note (i) = OSQPend (QDirector, 0, err) /* wait for the message */
.
};
```

In certain RTOS, a queue is given select option and the option is provided for priority or FIFO. The task having priority if started deletes a queue message first in case the priority option is selected. The task pending since longer period deletes a queue message first in case the FIFO option is selected.

An OS provides the IPC functions create, post, postfront, pend, accept, flush and query for using message queues. The timeout can be provided with 'pend' function argument. The error-pointer can also be provided in the argument.

7.13 MAILBOX FUNCTIONS

A message-mailbox is for an IPC message that can be used only by a single destined task. The mailbox message is a message-pointer or can be a message. (μCOS-II provides for sending message-pointer into the box). The source (mail sender) is the task that sends the message pointer to a created (initialized) mailbox (the box initially has the NULL pointer before the message posts into the box). The destination is the place where the OSMBoxPend function waits for the mailbox message and reads it when received.

A mobile phone LCD display task is an example that uses the message mailboxes as an IPC. In the mailbox, when the time and date message from a clock-process arrives, the time is displayed at side corner on top line. When the message from anther task to display a phone number, it is displayed the number at middle at a line. When the message from anther task to display the signal strength at antenna, it is displayed at the vertical bar on the left.

Another example of using a mailbox is the mailbox for an error-handling task, which handles the different error logins from other tasks.

Figure 7.9(a) shows three mailbox-types at the different RTOSes. Figure 7.9(b) shows the initialization and other functions for a mailbox at an OS. The following may be the provisions at an OS for IPC functions when using the mailbox.

1. A task may put into the mailbox only a pointer to the message-block or number of message bytes as per the OS provisioning.
2. There are three types of the mailbox provisions.

A queue (Section 7.12) may be assumed a special case of a mailbox with provision for multiple messages or message pointers. An OS can provide for *queue* from which a read (deletion) can be on a FIFO basis or alternatively an OS can provide for the multiple mailbox messages with each message having a priority parameter. The read (deletion) can then only be on priority basis in case mailbox message has multiple messages with priority assigned to high priority ones. Even if the messages are inserted in a different priority, the deletion is as per the assigned priority parameter.

An OS may provision for *mailbox* and *queue* separately. A mailbox will permit one message pointer per box and the queue will permit multiple messages or message pointers. μCOS-II RTOS is an example of such an OS.

The RTOS functions for mailbox for the use by the tasks can be the following:

1. OSMBoxCreate creates a box and initializes the mailbox contents with a NULL pointer.
2. OSMBoxPost sends (writes) a message to the box.

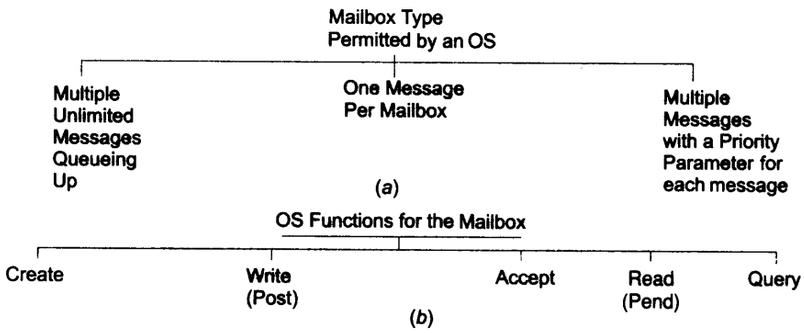


Fig. 7.9 (*a*) Mailbox types at the different operating systems (OSes) (*b*) Initialization and other functions for a mailbox at an OS

3. OSMBoxWait (pend) waits for a mailbox message, which is read when received.
4. OSMBoxAccept reads the current message pointer after checking the presence yes or no (no wait). Deletes the mailbox when read.
5. OSMBoxQuery queries the mailbox when *read* and not needed later.

An ISR can post (but not wait) into the mailbox of a task.

Example 7.20

(*a*) Consider an AVCM (Section 1.10.2). Assume that a message pointer IPC posts into the mailbox the *amount* collected by *Task Read_Amount* and the *Chocolate_delivery_task* section waits for taking message into the mailbox to make the amount equal to NULL after delivering the chocolate. Assume *mboxAmt* is a pointer to mailbox and fullAmount is a string *full amount* which should be made NULL after delivering the chocolate. OSMboxPost (mboxAmt, full Amount) is an OS IPC function for posting a message pointer into the mailbox and assume OSSemPend ( ) is another OS IPC function for waiting for the message pointer, *fullAmount*.

(*b*) Also assume that the OSMboxPost ( ) is also used by keypad for posting the mailbox IPC message for *userInput* into mailbox *mboxUser* from *Task User Keypad Input* and OSMboxPend ( ) for waiting for *userInput* for display. A mailbox IPC 'pend' is for mboxUser message in *Task_Display*.

The following will be the codes for (*a*) and (*b*).

(*a*) *static void Task Read_Amount (void *taskPointer)* {

.

while (1) {

.

/* Codes for reading the coins inserted into the machine */

/* Codes for writing into the mailbox full amount message if cost of chocolate is received*/

OSMboxPost (*mboxAmt, fullAmount*) /* Post for the mailbox message and *fullAmount,* which equaled NULL now equals *fullAmount* */

.

};

*static void Chocolate_delivery_task (void *taskPointer)* {

.

while (1) {

```
.
/* IPC for requesting full amount message */
fullAmountMsg = OSMboxPend (mboxAmt, 20, *err) /* Wait for the mailbox mboxAmt message for
    20 clock-ticks and error if message not found. mboxAmt becomes NULL after message is read.
.
};
(b) static void Task_User_Keypad_Input (void *taskPointer) {
.
while (1) {
.
/* Codes for reading keys pressed by the user before the enter key */
/* Codes for writing into the mailbox */
OSMboxPost (mboxUser, userInput) /* Post for the mailbox message and userInput, which equaled NULL
    now equals userInput */
.
};
static void Task_Display (void *taskPointer) {
.
while (1) {
.
/* IPC for waiting for User input message */
UserInputMsg = OSMboxPend (mboxUser, 20, *err) /* Wait for the mailbox mboxUser
    message for 20 clock ticks and error if message not found. mboxUser becomes null after
    message is read.
.
/* Code for display of user Input */
TimeDateMsg = OSMboxPend (timeDate, 20, err) /* Wait for the mailbox message timeDate.
/* Code for display TimeDateMsg Time: hr:mm Date: month:date */
.
};
```

An OS provides the IPC functions *create*, *post*, *pend*, *accept* and *query* for using the mailbox. The time-out and error-pointer can be provided with the 'pend' function arguments.

7.14 PIPE FUNCTIONS

The OS pipe functions are unlike message queue functions. The difference is that pipe functions are similar to the ones used for devices such as file.

A message-pipe is a device for inserting (writing) and deleting (reading) from that between two given interconnected tasks or two sets of tasks. Writing and reading from a pipe is like using a C command *fwrite with a file name* to write into a named file, and *fread with a file name* to read from a named file. Pipes are also like Java *PipeInputOutputStreams*.

1. One task using the function *fwrite* in a set of tasks, can insert (write) *to* a pipe at the back pointer address, *pBACK.
2. Another task using the function *fread* in a set of tasks can delete (read) from a pipe at the front pointer address, *pFRONT.

3. In a pipe there may be no fixed number of bytes per message but there is end-pointer. A pipe can therefore be inserted limited number of bytes and have a variable number of bytes per message between the initial and final pointers.
4. Pipe is unidirectional. One thread or task inserts into it and the other one deletes from it.

An example of the need for messaging and thus for IPC using a pipe is a network stack.

The OS functions for pipe are the following:

1. *pipeDevCreate* for creating a device, which functions as pipe.
2. *open* ( ) for opening the device to enable its use from beginning of its allocated buffer, its use is with options and restrictions (or permissions) defined at the time of opening.
3. *connect* ( ) for connecting a thread or task inserting bytes to the thread or task deleting bytes from the pipe.
4. *write* ( ) function for inserting (writing) from the bottom of the empty memory space in the buffer allotted to it.
5. *read* ( ) function for deleting (reading) from the pipe from the bottom of the unread memory spaces in the buffer filled after writing into the pipe.
6. *close* ( ) for closing the device to enable its use from beginning of its allocated buffer only after opening it again.

Figure 7.10(a) shows functions at an OS. A function is for initialization and creating a pipe. It defines pipe ID, length, maximum length (not defined in some OSes) and initial values of two pointers. These are *pFRONT and *pBACK for pipe message destination (head) and pipe message source (tail) memory locations, respectively. A function is for pipe connecting and thus defining source ID and destination ID. A function is for the error handling. Figure 7.10(b) shows pipe messages in a message buffer.

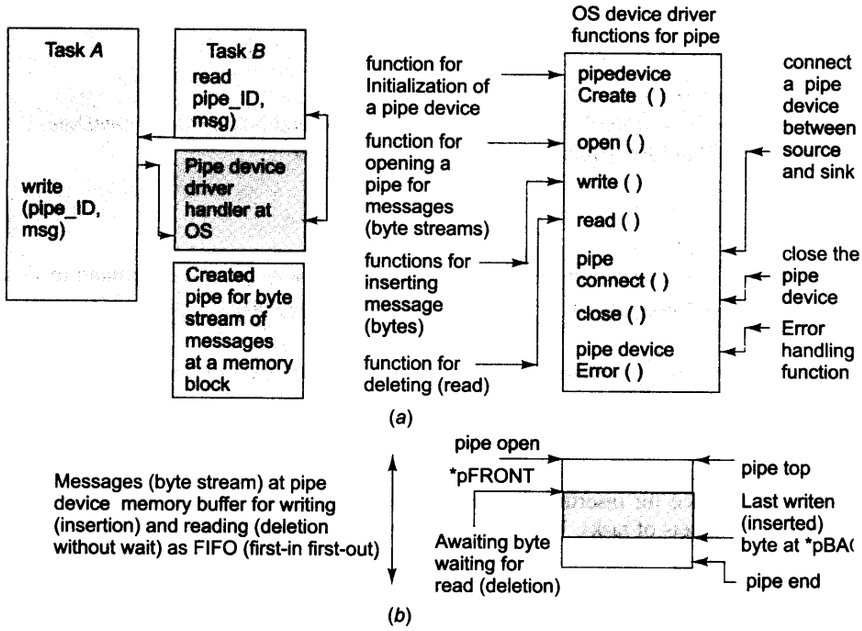


Fig. 7.10 (*a*) Functions at operating system (*initialization, connect, read, write* and *error-handling* functions) and the use of write and read functions by tasks *A* and *B* (*b*) Pipe messages in a message buffer

Example 7.21

Consider a smart card. [Section 10.3 and Example 7.4] When it is inserted into a card reader host machine, it gets the radiation and charges up.

1. Assume that the main program runs an OS function pipeDevCreate ( ) to create a task,
2. Assume that a pipe is used by the task, Task_Send_Card_Info for writing card information to the host using the pipe.

The codes at the card can be as follows.

```
.
pipeDevCreate ("/pipe/pipeCardInfo", 4, 32) /* Create a pipe pipeCardInfo, which can save four messages
    each of 32 bytes maximum */
fd = open (("/pipe/pipeCardInfo",O_WR, 0) /* Open a write only device. First argument is pipe ID /pipe/
    pipeCardInfo, second argument is option O_WR for specifying write only and third argument is 0 for
    unrestricted permission.*/
.
static void Task_Send_Card_Info (void *taskPointer) {
.
while (1) {
.
.
cardTransactionNum = 0; /* At start of the transactions with the machine*/
write (fd, cardTransactionNum, 1) /* Write 1 byte for transaction number after card insertion */
write (fd, cardFabricationkey, 16) /* Write 16 bytes for fabrication key */
write (fd, cardPersonalisationkey, 16) /* Write 16 bytes for personalisation key */
write (fd, cardPIN, 16) /* Write 16 bytes for PIN, personal identification number
    granted by the authorising bank */
.
};
```

An OS provides the IPC functions pipeDevcreate, open, connect, write, read and close. The pipe ID, limit of the total number of messages and the maximum size per message are also provided when creating a pipe.

7.15 SOCKET FUNCTIONS

Example 7.21 showed that a pipe could be used for inserting the byte steam by a process and deleting the bytes from the stream by another process. However, the use of pipe between a process at the card and a process at the host will have the following problems.

1. We need the card information to be transferred from a process *A* as bytes stream to the host machine process *B* and the *B* sends messages as bytes stream to *A*. There is need for bi-directional communication between *A* and *B*.
2. We need the *A* and *B*'s ID or port as well as address information when communicating. These must be specified either for the destination alone or for both source and destination (It is similar to sending the messages in a letter along with address specification).

A protocol provides for communication along with the byte stream information of the address or port of the destination alone or addresses or ports of both source and destination. A protocol may provide for the addresses as well as ports of both source and destination in case of the remote processes (for example, in IP protocol). Also, there are two types of the protocols.

1. There may be the need of using a *connectionless protocol* when sending and receiving message streams. An example of such a protocol is UDP (user datagram protocol). UDP protocol requires a UDP header, which contains source port (optional) and destination port numbers, length of the datagram and checksum for the header-bytes. Port means a process or task for specific application. The number specifies the process. Connectionless means there is no connection establishment between source and destination before actual transfer of data stream can take place. Datagram means a set of data, which is independent and need not be in sequence with the previously sent data. Checksum is sum of the bytes to enable the checking of the erroneous data transfer. For remote communication, the address, for example, IP address is also required in the header.
2. There may be the need of using a *connection-oriented protocol*, for example, TCP. Connection-oriented protocol means a protocol, which provides that there must first a connection establishment between the source and destination, and then the actual transfer of data stream can take place. At end, there must be connection termination.

Socket provides a device-like mechanism for bi-direction communication. It provides for using a protocol between the source and destination processes for transferring the bytes; it provides for establishing and closing a connection between the source and destination processes using a protocol for transferring the bytes; it may provide for listening from multiple sources or multicasting to multiple destinations. Two tasks at two distinct places are locally interconnect through the sockets. Multiple tasks at multiple distinct places interconnect through the sockets to a socket at a server process. The client and server sockets can run on the same CPU or at distant CPUs on the Internet.

Sockets can be using a different domain. For example, a socket domain can be TCP, another socket domain may be UDP, the card and host example socket domains are different.

The use of a socket for IPC is analogous to the use of sockets for an Internet connection between the browser and webserver. *A socket provides a bi-directional transfer of messages and may also send protocol header. The transfer is between two or between the multiple clients and server-process.* Each socket may have the task source address (similar to a network or IP address) and a port (similar to a process or thread) number. The source and destination sets of tasks (addresses) may be on the same computer or on a network. Figure 7.11 shows the initialized sockets between the client set of tasks and a server set of tasks at an OS.

Example 7.22

Assume that using the OS socket functions, a socket interconnects a byte stream between the source set of processes *I* and destination targeted set of processes *J*. Let there be the four process threads: *a*, *b*, *c* and *d* in process set *I*. There are two threads, *x* and *y* in a set of processes, *J*. Let the socket be used to send a byte stream from (process set *I*, process thread *c*) to a (process set *J*, process thread *x*). Now, the socket at the source (process 1 socket) is specified as the socket at (*I*, *c*) and socket at the process 2 as the socket at (*J*, *x*). When the bytes are sent or received at the socket at (*I*, *c*) from or to the socket at (*J*, *x*), the protocol specifies (*I*, *c*) and (*J*, *x*). The protocol may also specify the length of the bytes being communicated. The protocol may also specify the checksum of the bytes being communicated so that if any bit is lost in communication to remote then retransmission request can be sent.

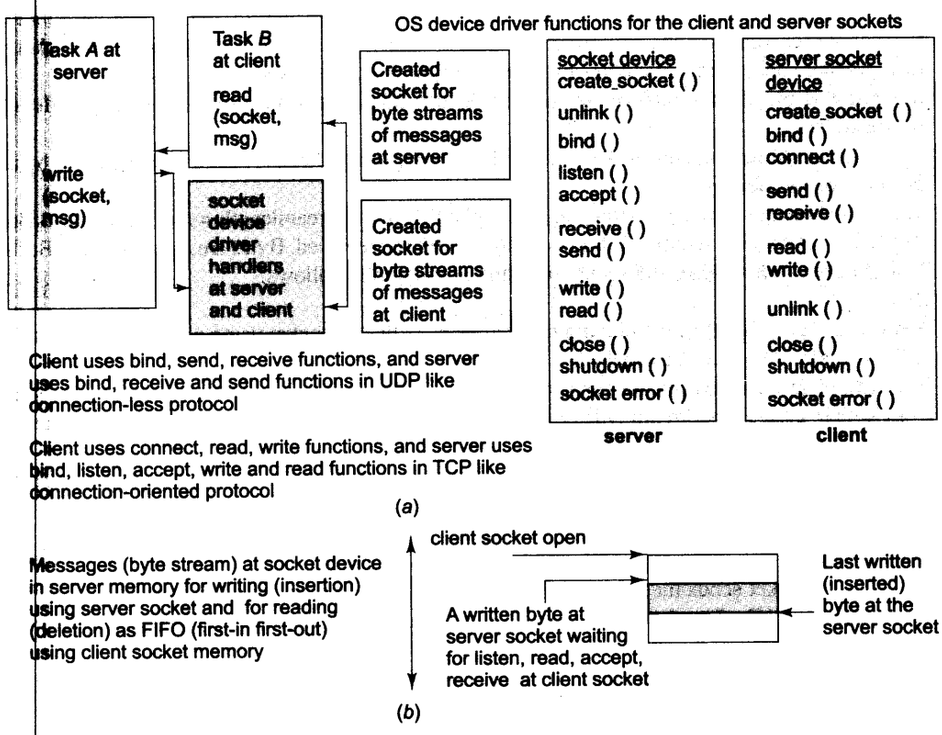


Fig. 7.11 (*a*) Initialized virtual sockets between the client set of tasks and a server set of tasks and the operating system provisions for the socket-functions (*b*) Byte stream between client and server

Example 7.23

Consider orchestra-playing robots (Example 1.5.7).

1. A process, Task_Master in the director robot creates a socket using a statement as follows:
   sfd1 = socket ("/socket/serversocket1", playStream, 0).
   The sfd1 is an unsigned integer for a socket descriptor. "/socket/serversocket1" is the path and file from which the stream, playStream will be sent or received from play robots. 0 represents unrestricted permission to use the file.
2. Task_Master socket binds the sfd1 and data structure at the socket address, sockAddr by using the function as follows:
   bind (sfd1, (struct sockaddr *)&local, lBytes).
   The lBytes is length of the bytes in the play stream;
3. Task_Master socket listens to eight orchestra-playing robots by using function as follows:
   listen (sfd1, 8)
4. Task_Master socket accepts bytes (from a playing robot process socket) from the socket with descriptor sfd2 using function as follows: sfd2 = accept (sfd1, &playRobotSockAddress, &playRlBytes)

The &playRlBytes refers to address for the maximum length of bytes from the playing robot. The playRobotSockAddres is the address of the data structure for the client (playing robot) socket.

5. Task_Master socket sends the bytes by using the function as follows:
   send (sfd2, & playBuffer, playstreamlen, 0); /* Send total playstreamlen bytes from playBuffer using the socket sfd2.
6. Task_Master socket receives the bytes from the playing robots (e.g., for acknowledgement) by using the function as follows:
   while (streamlength > 0 && streamlength < = playRlBytes) {streamlength = recv (sfd2, &ackBuffer, 200, 0)}; /* The recv () returns –1 if no more bytes are to be received. Bytes are received in ackBuffer */
7. Task_Master socket closes the socket by using a function as follows:
   close ( );
8. A process, Task_Client in the playing robot creates a client 1 socket using a statement as follows:
   sfd2 = socket ("/socket/clientsocket1", sockStream, 0).
   The sfd2 is an unsigned integer for a socket descriptor. 'socket/serversocket1' is the path and file from which stream, socStream will be sent or received from play robots. 0 represents unrestricted permission to use the file.
9. Task_Client socket connects the sfd1 and data structure at the socket address, sockClientAddr by using the function at the server process as follows:
   connect (sfd2, (struct sockClientaddr *)&remote, CllBytes).
   The CllBytes is maximum length of the bytes in the play stream.
10. Task_Client socket sends the acknowledgement bytes by using the function as follows:
    send (sfd2, & ackBuffer, playRlBytes, 0); /* Send total ackstreamlen bytes from ackBuffer, using the socket sfd.
11. Task_Client socket receives the bytes from the playing robots (e.g., for acknowledgement) by using the function as follows:
    while (streamlength > 0 && streamlength < = playstreamlen) { streamlength = recv (sfd2, &playBuffer, 200, 0)}; /* The recv () returns –1 if no more bytes are to be received. Bytes are received in playBuffer */

Application of sockets are as follows:

1. An *application* of sockets is to connect the tasks in the distributed environment of embedded systems. For example, a network interconnection or a card process connects a host-machine process.
2. A TCP/IP socket is another common *application* for the Internet. Another exemplary *application* of socket is a task receiving a byte stream of TCP/IP protocol at a mobile internet connection.
3. An exemplary *application* is when a task writes into a file at a computer or at a network using NFS protocol (network file system protocol).
4. Another *application* of the socket is the interconnection of a task or a section in a source set of tasks in an embedded system with another task at a destined set of tasks. The kernel has the socket-connecting functions with the codes specifying the source and destination sets and tasks.

The OS functions for socket in Unix are as following.

1. The socket ( ) [in place of open ( ) in case of pipe] gives a socket descriptor sfd. The socket ( ) enables its use from beginning of its allocated buffer at the socket address, its use with option and restrictions or permissions defined at the time of opening. A socket can be a stream, SOCK_STREAM or UDP datagram SOCK_DGRAM.
2. The unlink ( ) before the bind ( ).

3. The bind ( ) for binding a thread or task inserting bytes into the socket to the thread or task and deleting bytes from the socket. bind ( ) the socket descriptor to an address in the Unix domain. bind (sfd, (struct sockaddr *)&local, len); where len is string length. sockaddr is a data structure with a record of 16-bit unsigned num and a path for the file and a data structure struct sockaddr_un {unsigned short num; char path[108]; }
4. The listen (sfd, 16) function for listening 16 queued connections from the client socket.
5. The accept ( ) accepts the client connection and gives a second socket descriptor.
6. The recv ( ) function for deleting (reading) and receiving from the socket from the bottom of unread memory spaces in buffer. The buffer has messages after writing into the socket.
7. The send ( ) function for inserting (writing) and sending from the socket from the bottom of the memory spaces in the buffer filled after writing into the socket.
8. The close ( ) for closing the device to enable its use from beginning of its allocated buffer only after opening it again.

1. A Socket is an IPC for sending a byte stream or datagram from one or multiple task sockets to another task or server process socket as a bi-direction FIFO-like device using a protocol for transferring the bytes. Datagram provide protocol-header bytes along with the byte stream.
2. The sockets can be a client-server set of sockets (multiple processes and single server process) or peer-to-peer sockets IPC. A socket has a number of applications. An Internet connection socket is for virtual connection between two ports: one port at an IP address to another port at another IP address.
3. An OS provides the IPC functions for creating socket, unlinking, binding, listening, accepting, receiving, sending and closing.

7.16 RPC FUNCTIONS

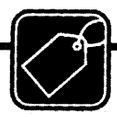
RPC is a method used for connecting two remotely placed methods by first using a protocol for connecting the processes. It is used in the cases of distributed tasks.

The OS can provide for the use of RPCs. These permit distributed environment for the embedded systems. The RPC provides the IPC when a task is at system 1 and another task is at system 2. Both systems work in the peer-to-peer communication mode. Each system in peer-to-peer can make RPCs. The OS IPC function allows a function or method to run at another address space of shared network or an other remote computer. The process 1 makes the call to the function that is local or remote and the process 2 response is either remote or local in the process 1 to process 2 method call.

Summary

- A process is a computational unit that processes on a CPU under the state-control of a kernel in OS.
- A process may consist of multiple threads that define thread as a minimum unit for a scheduler to schedule it to run the CPU and provide other system resources. Unix provides for processes and their threads as light-weight processes. Light weight mean functions not dependent on functions like memory-management functions. Java use the threads.
- A single CPU system runs one process (or one thread of a process) at a time. A scheduler is a must to schedule a multitasking or multithreading system.

- A task is a computational unit or set of codes, actions or functions that processes on a CPU under the state-control of a kernel in OS.
- A task is similar to a process or thread. Each task is an independent process that takes control of the CPU when scheduled by a scheduler at an OS. No task can call another task. Each task and its state is recognized by its TCB (memory block) that holds information of the PC, memory map, the signal (message) dispatch table, signal mask, task ID, CPU state (registers and so on) and a kernel stack (for executing system calls and so on). A task is in one of the four states: idle, ready, blocked and running, that are controlled by the scheduler.
- Often the same data is used in two different tasks (or processes) and if another task interrupts before without completing the operation on that data, then the shared data problem arises. Disabling of interrupts till the completion of the operation by the first task, and then re-enabling interrupts, is one solution. Use of *semaphores* (as *tokens, mutex or counting semaphores*) is another efficient way for solving the shared data problem and running critical section codes. Use of lock functions and spin-locks are also provided in the OSes.
- A buffer is a memory block for a queue or stream of bytes between an output source and input sink. [For example, between the tasks, files, computer and printer, physical devices and network.] It has to be bounded between two limits. It cannot be unlimited or infinite. For example, a print buffer cannot accept unlimited output from a computer. The bounded buffer problem is of synchronizing the source and sink. A *producer cannot keep on producing beyond a limit if consumers do not consume. The consumers cannot keep consuming unless the producer keeps producing.* Counting semaphores provide solution to this problem.
- There has been POSIX IEEE standardization of the OS and IPC functions, for example, the P and V semaphore POSIX functions. These function are event notifying, resource key, mutex and counting semaphores.
- *The priority inversion* problem and deadlock situation can arise in certain situations when using a semaphore. An OS should be such that it can take care of it by having the appropriate provisions to avoid these situations. Certain OSes provide mutex semaphores such that priority inversion problem does not arise.
- The OS functions handle IPCs between the multiple tasks.
- The OS provides for the following IPCs: signals, semaphores, queues, mailboxes, pipes, sockets and RPCs.
- A mailbox may either provide for only one message or multiple messages in an RTOS.
- A pipe is a queue or stream of messages that connects the two tasks and that uses the functions as used in a device.
- The sockets are used in networks or client-server-like communication between the tasks using functions as used for the devices. The RPCs are used for the case of distributed tasks.



Keywords and their Definitions

Buffer : A memory block for a queue or network stack or pipe or stream of bytes between an output source and input sink, for example, between the tasks, files, computer and printer, physical devices and network.

Counting semaphore : A semaphore in which the value of which can be initialized to an 8- or 16- or 32-bit integer and that is decremented and incremented. A task does not block if its value is found to be >0 and the task blocks if its value is found to be 0.

Critical section : A section in a task the execution of which should block execution of another such section in another task, for example, when a buffer in printer is shared between two or more tasks.

Deadlock situation : A task waiting for the release of a semaphore from a task and another a different task waiting for another semaphore release to run. None of these is able to proceed further due to circular dependency. An OS can take care of this by appropriate provisions.

Interprocess communication : A mechanism from one task (or process) sending signal or messages or event notification from one task to the system and which the OS communicates to another task. Using IPC mechanism and functions a task uses signals, exceptions, semaphores, queues, mailboxes, pipes, sockets and RPCs.

Mailbox : A message(s) from a task that is addressed for another task.

Message queue : A task sending the multiple messages into a queue for use by another task(s) using queue messages as an input.

Mutex : The special variable and mechanism used to take note of certain actions to prevent any task or process from proceeding further and at the same time let another task exclusively proceed further. Mutex helps in mutual exclusion of one task with respect to another by a scheduler in the multitasking operations.

P and V semaphores : The semaphore functions defined in a POSIX IEEE standard to be used as event notification or mutex or counting semaphore, or to solve the classical producer–consumer problem when using a bounded buffer.

Pipe : A device for use by the task for sending the messages and another task using the device receives the messages as stream. A pipe is a unidirectional device.

Priority inversion : A problem in which a low priority task inadvertently does not release the process for a higher priority task. An operating system can take care of this by appropriate provisions.

Process : A code that has its independent PC values and an independent stack. A single CPU system runs one process (or one thread of a process) at a time. A *process* is a concept (abstraction). It defines *a sequentially executing (running) program and its state. A state*, during the running of a process, is represented by its status (running, blocked or finished), its control block, called process control block (PCB) or *process structure*, its data, objects and resources.

Remote procedure call : A method used for connecting two remotely placed methods by first using a protocol for connecting the processes. It is used in the cases of distributed tasks.

Semaphore : A special variable operated by the OS functions which are used to take note of certain actions to prevent another task or process from proceeding further.

Shared data problem : If a variable is used in two different processes (tasks) and if another task interrupts before the operation on that data is completed, then the shared data problem arises.

Signal : A function to call a signal handler by interrupting the processes. It uses INT n SWI n instruction, where n defines the handler, which should run.

Socket : It provides the logical link using a protocol between the tasks in a client-server or peer-to-peer environment. It enables a bi-directional stream or datagram or network stack.

Synchronization : To let each section of codes, tasks and ISRs run and gain access to the CPU one after the other sequentially or concurrently, following a scheduling strategy, so that there is a predictable operation at any instance.

Task : A task is for the service of specific actions and may also correspond to the codes, which execute for an interrupt. A task is an independent process that takes control of the CPU when scheduled by a scheduler at an OS. Every task has a TCB.

Task control block : A memory block that holds information of the PC, memory map, the signal (message) dispatch table, signal mask, task ID, CPU state (registers and so on), and a kernel stack (for executing system calls and so on).

Task state : A state of a task that changes on scheduler directions. A task at an instance can be in one of the four states, *idle*, *ready*, *blocked* and *running* that are controlled by the scheduler.

Thread : A minimum unit for a scheduler to schedule the CPU and other system resources. A process may consist of multiple threads. A thread has an independent process control block like a TCB and a thread executes codes under the control of a scheduler. It is a light weight process.

Review Questions

1. How does a data output generated by a process transfer to another using an IPC?
2. What are the parameters at a TCB of a task? Why should each task have distinct TCB?
3. What are the states of a task? Which is the entity controlling (scheduling) the transitions from one state to another in a task?
4. Define critical section of a task. What are the ways by which the critical section run by blocking other process(es)?
5. How is data (shared variables) shielded in a critical section of a process before being operated and changed by anther higher priority process that starts execution before the process finishes?
6. How does use of a counting semaphore differ from a mutex? How is a counting semaphore used?
7. Give an example of a deadlock situation during multiprocessing (multitasking) execution.
8. What is the advantage and disadvantage of disabling interrupts during the running of a critical section of a process?
9. Explain the term multitasking OS and multitasking scheduler.
10. Each process or task has an endless (infinite) loop in a pre-emptive scheduler. How does the control of resources transfer from one task to another?
11. What is an *exception* and how is an error-handling task executed on throwing the exception?
12. How do functions differ from ISRs, tasks, threads and processes? Why is an ISR not permitted to use the IPC pend wait functions.
13. List the features of P and V semaphores and how these are used as a resource key, as a counting semaphore and as a mutex.
14. What are the situations, which lead to priority inversion problems? How does an OS solve this problem by a priority inheritence mechanism?
15. What is meant by a pipe? How does a pipe may differ from a queue?
16. What is meant by a spinning lock? Explain the situation in which the use of the spin lock mechanism would be highly useful to lock the transfer of control to an higher priority task?
17. What is a mailbox? How does a mailbox pass a message during an IPC?
18. When are the sockets used for IPCs? List four examples. When are RPCs used? List two examples.
19. What are the analogies between process, task and thread? Also list the differences between the process, task and thread.

Practice Exercises

20. Design a table to clearly distinguish the cases when there is concurrent processing of processes, when tasks and when threads by using a scheduler.
21. Make a table similar to Table 7.1 to clearly distinguish ISRs, ISTs and tasks.
22. What is the advantage of using a *signal* as an IPC? List the situations which warrant use of signals.
23. List five exemplary applications of solutions to the bounded buffer problem using P and V mutex semaphores.
24. Every tenth second a burst of 64 kB arrives at 512 kbps in an interval of 100 seconds. Is an input buffer required? If yes, then how much? If yes, then write a program to use the buffer using P and V semaphores.
25. Use Web search to understand an IEEE-accepted standard POSIX 1000.3b in detail.
26. Can different IPCs be used? Given the choice, how will you select an IPC from signal, semaphore, queue or mailbox?
27. List the tasks in the automatic chocolate-vending machine (Example 1.10.2). List the IPC functions required and their uses in the ACVM.
28. List the processes used in smart card (Example 1.10.3). How does the card communicate with the host using the sockets? List the IPC functions required and their uses in the smart card.
29. List the tasks in the digital camera (Example 1.10.4). List the IPC functions required and their uses in the camera.
30. List the processes in the smart mobile phone (Example 1.10.5). The display process has multiple threads in the phone. List the threads. List the IPC functions required and their uses in the phone.
31. List the processes in the PDA (Example 1.10.6). Assume that PDA services the events by the ISRs and signal handlers using a queue of the events. How can this be done? Show it by a diagram.
32. List the processes in the director of OPRs (Example 1.10.7). List the processes in eight playing robots in OPRs.

Real-Time Operating Systems

8

Following points have been discussed in the previous chapter.

- Application program has functions, ISRs, threads, processes (tasks) multiple physical as well as virtual device drivers and several program objects, which concurrently process on single or multiple processors.
- OS functions provide a mechanism to create multiple tasks (processes and threads), control the task-states and allocate system-resources to the tasks.
- OS functions control the context-switching in multiprocessing (multitasking and multi-threading) program.
- The OS functions provides the IPC functions to enable communication of the signals, semaphores and messages from the ISRs and tasks to other waiting service-thread or tasks.
- The OS functions also provides the IPC functions for the pipes, sockets and RPCs.
- The OS also provides for mutex, lock and spin-lock functions and for disabling of interrupts to let a critical section of code run without pre-emption by other process(es).

LEARNING OBJECTIVE

We will learn the following in this chapter.

1. *OS services.*
2. *Program structure, in-between layers of OS and interfaces between the top application software and down system-hardware layers.*
3. *OS functions for the process (task), timer, event, memory, device, file-system and I/O-subsystem management.*
4. *RTOS providing in addition to the basic OS services, a control on the context switching between the tasks such that the system satisfies the real-time requirements, time constraints and deadlines of tasks.*
5. *ISR-handling in an RTOS environment.*
6. *Basic design principles when using an RTOS.*
7. *Soft and hard real-time scheduling considerations.*
8. *RTOS task-scheduling service models and basic strategies for scheduling the multiple tasks–cooperative, cyclic, time slicing round robin and preemptive scheduling RTOS, and critical section handling in priority scheduling cases.*
9. *OS security issues.*

Chapters 9 to 12 will describe with exemplary RTOSes and case studies.

8.1 OS SERVICES

8.1.1 Goal

The OS goals are *perfection and correctness* to achieve the following:

1. *Facilitating easy sharing of resources as per schedule and allocations.* Resources mean processor(s), memory, I/Os, devices, virtual devices (e.g., pipes, sockets), system timer, keyboard, displays, printer and other such resources, which processes (tasks or threads) request from the OS. No processing task or thread uses any resource until it has been allocated by the OS at a given instance.
2. *Facilitating easy implementation* of the application-program with the given system-hardware. An application programmer for a system can use the OS functions that are provided in given OS without having to write the codes for the services (functions) that follow.
3. *Optimally scheduling* the processes on one (or more CPUs if available) and providing an appropriate context-switching mechanism.
4. *Maximizing the system performance to let different processes (tasks or threads) share the resources most efficiently with protection and without any security breach.* Examples of security breach are tasks obtaining illegal access to other task-data directly without system calls, overflow of the stacks into memory and overlaying of PCBs (Section 7.1) at the memory.

5. *Providing management functions* for the processes (tasks or threads), memory, devices and I/Os and other functions.
6. *Providing management and organization* functions *for the devices, files and virtual devices and I/Os.*
7. *Providing easy interfacing* and management functions for the network protocols and networking.
8. *Providing portability* of application on different hardware configurations.
9. *Providing interoperability* of application on different networks.
10. *Providing a common set of interfaces* that integrates various devices and applications through the standard and open systems.

The OS goals are perfection, correctness, portability, interoperability and providing a common set of interfaces for the system, and orderly access and control when managing the processes.

8.1.2 User and Supervisory Mode Structure

When using an OS, the processor in the system runs in two modes. There is clock, called system clock. At every tick of the clock, there is an interrupt. On interrupt, the system time updates, the system context switches to the supervisory mode from the user mode. After completing the supervisory (kernel-space) functions, the system context switches back to the user mode.

1. *User mode*: The user process is permitted to run and use only a subset of functions and instructions in OS. This is done in the user mode either by sending a message to a waiting process associated with the OS kernel or by initiating a system call (call by an OS function). The use of hardware resources including memory is not permitted without making the call to the OS functions. The OS calls the resources by system call. User function call is distinct from a system call, and is not permitted to read and write into the protected memory allotted to the OS functions, data, stack and heap. This protected memory space is also called kernel space. Hence execution of user functions calls is slower than the execution of the OS functions (which run on system call). This is because of the protected access to memory by the functions running in user-space.
2. *Supervisory mode*: The OS runs the privileged functions and instructions in the protected mode and the OS (more specifically, the kernel) only accesses the hardware resources and the protected area memory. [The term kernel means nucleus.] In the supervisory mode the kernel codes run in protected mode. Only a system-call is permitted to read and write into the protected memory allotted to the OS functions, data, stack and heap. The kernel space functions execute faster than the user-space functions.

Example 8.1

RTOS Windows CE and several RTOSes enables running of all the application-program threads in the supervisory mode (kernel mode). Therefore, the threads execute fast. This improves the system performance. If the threads are to execute in the user mode, as in Unix or in non-real-time OS, then the execution slows down due to checks on the code access to the protected kernel space.

8.1.3 Structure

A system can be assumed to have a structure as per Table 8.1.

Table 8.1 Layered Model of the System

Layer from Top	*Top-down Structure Layers*	*Actions*
1	Application software	Executes as per the applications, run on the given system hardware using the interfaces and the system software.
2	Application program interface (API)	Provides the interface (for inputs and outputs) between the application software and system software so that it is able to run on the processor using the given system software.
3	System software other than the one provided at the OS (operating system)	The OS may not have the functions, for example, for the specific network and certain device drivers, such as a multimedia device. This layer gives the system software services other than those provided by the OS service functions.
4	OS interface	Interface (*for inputs and outputs*) between the above and OS
5	OS	Kernel supervisory mode services (Table 8.2), file management and other functions such as the user-mode processing services.
6	Hardware–OS interface	Interfaces to let the functions be executed on the given hardware (processor, memory, ports and devices) of the system.
7	Hardware	Processor(s), memories, buses, interfacing circuits, ports, physical devices, timers and buses for devices networking.

The OS structure consists of kernel and other service functions. The OS enables an *application* run on the system-hardware.

8.1.4 Kernel

The OS is the middle layer between the application software and system hardware. An OS includes some or all of the following structural units.

1. Kernel with file management and device management as part of the kernel in the given OS.
2. Kernel without file management and device management as part of the kernel in the given OS and any other needed functions not provided for at the kernel.

The kernel is the basic structural unit of any OS in which the memory space of the functions, data and stack are protected from access by any call other than the system-call. It can be defined as a secured unit of an OS that operates in the supervisory mode while the remaining part and the application software operates in the user mode. Table 8.2 gives the functions (services) in the kernel, they are as per the OS design.

The kernel has management functions for processes, resources, ISRs, ISTs, files, device drivers and IO subsystems and network subsystems. The memory or device and file management functions may be outside the kernel in a given OS, especially in an embedded system.

Table 8.2 Kernel Services in an operating system (OS)

Function	*Actions*
Process[1] management: creation to deletion	Enables process creation, activation, running, blocking, resumption, deactivation and deletion and maintains process structure at a PCB (process control block) (Section 7.1.1)
Process management: process structure maintenance	Enables process structure maintenance and its information at PCB
Process management: processing resource requests	Processing resource requests by processes made either by making calls that are known as system calls or by sending message(s)
Processes management: scheduling	Processes *scheduling*. For example, in the cyclic scheduling or priority scheduling mode (Section 8.10)
Process management: interprocess communication (IPC) (communication between tasks, ISRs, OS functions)	Processes synchronizing by sending data as messages from one task to another. The OS effectively manages shared memory access by using the IPC signals, exception (error) handling signals, semaphores, queues, mailboxes, pipes and sockets (Section 7.9)
Services memory management allocation and de-allocation[2]	Memory allocation, de-allocation and management. It also restricts the memory access region for a task (Section 8.5)
File management[3]	File management provides management of the creation, deletion, read ( ), write ( ) to the files on the secondary memory disk (Section 8.6). A file in the embedded system (disk-free system) can be in RAM, where the operations are done in RAM memory in a way identical to the file on disk
Device management[4]	A physical device management is such that it is accessible to one task or process only at an instant. Device manager components are: (i) device drivers and device ISRs (device interrupt handlers): (ii) resource managers for the devices (Section 8.6). Besides physical devices, the management of *virtual device* like pipe or socket is also provided (Sections 7.14 and 7.15). Virtual devices emulate a hardware device and the virtual device driver send signals (Section 7.10) similar to the ISR calls by the physical device
Device drivers	Facilitate the use of number of physical devices like keyboard, display systems, disk, parallel port, network interface cards, network devices and virtual devices (Section 8.6)
I/O management	Character or block I/Os management. For example, to ensure actions such that a parallel port or serial port is accessible to only one task at a time (Section 8.6)
Interrupts control mechanism (for handling ISRs)	Facilitate running of the ISRs and ISTs (Section 8.7)

[1] When considering the processes controlled by an OS, a process also means task in multitasking OS, and thread in multithreading OS (Refer to Sections 7.1 to 7.3).

[2,3,4] Memory, file and device management functions form the part of kernel in a given OS. However, these functions may be outside the kernel in a given OS, especially in an embedded system using only the microkernel of an OS. This exclusion makes the kernel code small.

8.2 PROCESS MANAGEMENT

8.2.1 Process Creation

At reset of processor in a computer system, an OS is initialized first, and then a process, which can be called initial process, is created (initialization of OS means enabling the use of OS functions, which includes the function to create the processes). Then the OS is started and that runs an initial process (starting the OS means calling the OS functions, which includes the call to all the created processes before the OS start but after the OS initialization). Processes can be created hierarchically. The initial process creates subsequent processes. OS schedules the processes and provides for context switching between the processes and threads (Sections 7.1 to 7.3).

Example 8.2

(a) Recall Example 7.4. It showed an RTOS function to create a process Task_Send_Card_Info by using OS_Task_Create ( ) function in the *main*. Task_Send_Card_Info task creates two other tasks, Task_Send_Port_Output and Task_Read_Port_Input. The OS then controls the context switching between the tasks.

(b) Consider Example 7.2 for a mobile phone device (Section 1.10.5).

An OS function first creates the *Display_process*. The display process then creates the following threads:

1. Display_Time_DateThread
2. Display_BatteryThread
3. Display_SignalThread
4. Display_ProfileThread
5. Display_MessageThread
6. Display_Call StatusThread
7. Display_MenuThread.

Creation of a process means specifying the resources for the process and address spaces (memory blocks) for the created process, stack, data and heap, and placing the process initial information at a PCB. The process manager allocates a PCB (or TCB in case task represents a process) when it creates the process and later manages it. The other OS units can send the manager the queries for the process when required. PCB is a *process descriptor* used by the process manager.

A PCB or TCB (Sections 7.1 and 7.3) describes the following.

1. *Context*: Processor status word, PC, SP and other CPU registers at the instant of the last instruction run executed when the process was left and the processor switched to another process.
2. *Process stack pointer.*
3. *Current state:* Is it created, activated or spawned? Is it running? Is it blocked? (spawn means create and activate)
4. *Addresses* that are allocated and that are presently in use.
5. *Pointer for the parent process* in case there exists a hierarchy of the processes.
6. *Pointer to a list of daughter processes* (processes lower in the hierarchy).
7. *Pointer to a list* of *resources, which are usable (consumed) only once.* For example, input data, memory buffer or pipe, mailbox message, semaphore (there may be producers and consumers of these resources).
8. *Pointer to a list of resource type usable more than once*: A resource type example is a memory block. Another example is an IO port. Each resource type will have a count of these types. For example, the number of memory blocks or number of IO ports.

9. *Pointer to queue of messages.* It is considered as a special case of resources that are usable once. It is because messages from the OS also queue up to be controlled by a process.
10. *Pointer to access permissions descriptor* for sharing a set of resources globally, and with another process.
11. *ID* by which identification is made by the process manager .

8.2.2 Management of the Created Processes

Recall process, thread and task definitions in Sections 7.1 to 7.3. A process (or thread or task) is considered a unit in which sequential running is feasible only under the control of an OS, with each process having an independent control block (descriptor of the process at an instant). (Recall the PCB and TCB described in Sections 7.1 and 7.3.)

Process manager is a unit of the OS that is the entity responsible for controlling a process execution. Process management enables process *creation, activation, running, blocking, resumption, deactivation* and *deletion*. A process manager facilitates the following. Each process of a multiple process (or multitasking or multithreading) system is executed such that a process state can switch from one to another. A process does the following sequential execution of the states: 'created', 'ready or activate', 'spawn' (means create and activate), 'running', 'blocked' or 'suspended', 'resumed' and 'finished' and 'ready' after 'finish' (when there is an infinite loop in a process) and finally 'deleted'. Blocking and resuming can take place several times in a long process. The different OSes make the provisions for possible states between creation and deletion differently.

The process manager executes a process request for a resource or OS service and then grants that request to let the processes share the resources. For example, an LCD display is a shared resource. The LCD display can be used only by one task or thread at an instance. A running process requests by two methods, which are listed in Table 8.3.

The process manager (i) makes it feasible for a process to sequentially (or concurrently) execute or block when needing a resource and to resume when it becomes available; (ii) implements the logical link to the resource manager for resources management (including scheduling of processes on the CPU); (iii) allows specific resources sharing between specified processes only; (iv) allocates the resources as per the resource allocation mechanism of the system and (v) manages the processes and resources of the given system.

A process manager creates the processes, allocates to each a PCB, manages access to resources and facilitates switching from one process state to another. The PCB defines the process structure for a process-state.

8.3 TIMER FUNCTIONS

A real-time clock (hardware timer timeout) in the system interrupts the system with each tick, which occurs a number of times in 1 second. An interrupt on a tick can be called SysClkIntr (system real-time clock timer interrupt). An OS provides a number of OS timer functions. *These functions use SysClkIntr interrupts on the clock ticks.*

The periodic SysClkIntr interrupts on this tick is used by the system to switch to the supervisory mode from the user mode on every tick. The following are the steps.

1. Before servicing of SysClkIntr, the context of presently running task or thread saves on the TCB data structure.
2. SysClkIntr service routine calls the OS.
3. The OS finds the new messages or IPCs (Sections 7.9 to 7.15), which are received from the system call by the OS event control blocks for IPC functions.
4. The OS either selects the same task or selects a new task or thread [by preemption (Section 8.10.3) in case of preemptive scheduling] and switches the context to the new one.
5. After return from the interrupt, the new task runs from the code, which was blocked from running earlier.

Table 8.3 Request for a Resource or Operating System (OS) Service by a Running Process

Request Method	*Explanation*
Message(s)	A process running on user mode generates and puts (sends) a message so that the OS lets the requested resource (e.g., input from a device or from a queue) use or run an OS service function (e.g., to define a delay period after which the process needs to be run again). A message can be sent for the OS to let the LCD display be used by a task or thread for sending the output. An ISR sends a message to a waiting thread to start on return from the ISR (Section 8.7).
System call	A call to a function defined at the OS. For example, OSTaskCreate ( ) is a system call to create a task. First an SWI instruction is issued to trap the processor and switch it to supervisory mode. The OS then executes a function like a library function . On finishing the instructions of the called function, the processor switches back from the supervisory mode to the user mode and lets the calling process run further.

Each OS has a function for defining the OS ticks per second, which defines generation of the SysClkIntr interrupts and which in turn provides the timer functions of the OS. The function thus defines the SysClkIntr interrupt intervals after initiating the ticks. The functions thus also define the period after which the system calls the ISR on the SysClkIntr interrupts and switches to the OS supervisory mode functions listed before.

Example 8.3

(a) # define OS_TICK_PER_SEC 100 /*µCOS-II function to define the number of ticks per second = 100 before the beginning of main ( ) and initiating of OS by OSInit ( ) function*/.
(b) OSTickInit ( ) /*µCOS-II function to initiate the defined number of ticks per second after the beginning of the first task and creation of all the tasks to which the context will be switched by the OS on the tick. It initiates SysClkIntr interrupts every 10 ms as OS ticks/s = 100 */.

The number of ticks if made large, then the frequent running of the OS codes to be run on SysClkIntr interrupt takes place because the context switching to the supervisory mode takes place too frequently. The number of ticks, if made small, the total time spent on SysClkIntr interrupts per second reduces, but then system time accuracy for the OS timer functions becomes small.

Example 8.4

Let OS ticks period be set to 10 ms. Assume that time spent in servicing SysClkIntr interrupt is 10 µs. 10 µs × 100 ÷10 ms = 0.1% of time is spent in SysClkIntr interrupt-initiated ISRs. Let the OS ticks period be set to 100 µs. Assume that time spent in servicing SysClkIntr interrupt is 10 µs. 10 µs × 100 ÷100 µs = 10% of time is spent in SysClkIntr interrupt-initiated ISRs though time resolution of OS time functions improve to 100 µs instead of 10 ms earlier.

Table 8.4 gives the example of RTOS timer functions and the actions on calling these functions. There are RTOS timer functions for the delay, delay resume, time set, time get and for waiting-time setting for the IPC events (e.g., semaphore, mailbox, message queue message).

Table 8.4 Exemplary Timer Functions

Function	*Action on Calling the Function*
OS_TICK_PER_SEC	Defines the number of system clock ticks per second, which is also the number of SysClkIntr interrupts per second.
OSTickInit ( )	Initiate the clock ticks in the system and SysClkIntr interrupts
OSTimeDelay ( )	Delay the task executing this function
OSTimeDelayResume ( )	Resume the delayed task
OSTimeSet ( )	Set the system clock tick count value. OSTimeSet (1000) sets the *count* = 1000. OSTimeSet (0) sets the *count* = 0. After each SysClkIntr and clock tick the *count* increments by 1
OSTimeGet ( )	Get the system clock-tick's *count* value.
OSSemPend (semVal, twait, *semErr)	Wait for the semaphore release and semVal becoming 1 for the period as per twait, if semaphore released in this period, then take it and task proceeds further after decrementing the semVal to 0, else after *twait* defined period, the task code proceeds with no further wait. If *twait* = 100, then wait for 100 system clock ticks. The semErr points to the error.
OSMboxPend (semVal, twait, *mboxErr)	Wait for the mailbox message for the period as per twait, if mailbox is posted the message, then take it and the task proceeds further, else after *twait* defined period, the task code proceeds further. If *twait* = 100, then wait for 100 system clock-ticks. The mboxErr points to the error.

Example 8.5

(a) OSTimeDelay (n) by period equal to period of n clock ticks.
(b) Assume that using the timer functions, OS_TICKS_PER_SEC 100 and OSTickInit ( ); the system clock ticks every 10 ms. Assume that ACVM (Section 1.10.2), the Task_ReadCoins for reading the coin is running. Consider that the following codes in the while loop of an AVCM task, *Task Read-Amount* (Example 7.3) waits for 1 minute for the coin amount:

```
/*************************************************************/
count =0; OSTimeSet (count);
while (count < = 6000) /* While loop waits for for the coins amount upto 60000 ms = 1 minute */
{count = OSTimeGet ( ); OSTimeDelay (100);
/* Code for finding the coins amount after every 100 clock ticks, which means every 1 s*/
; }
/*************************************************************/
```

(c) Calling OSTimeGet ( ), finding the count, *count1*, running a section of codes and then calling OSTimeGet ( ), finding the new count, *count2* defines the interval, T spent by the system in-between the two function calls of OSTimeGet ( ). T = {(*count2* – *count1*) × interval between two clock ticks} is the interval between two calls to OSTimeGet.

8.4 EVENT FUNCTIONS

In case of IPC (Section 7.9), there is wait for only one semaphore post event or mailbox message-posting event (Sections 7.10 to 7.15). Provisioning of event functions in an OS offers an advantage that there can be wait for more than one event and the events can also be from the different tasks or ISRs.

The queue messages can be from the different tasks or ISRs. Queues can offer the same advantage that there can be wait for more than one messages. However, the OS functions for queue execute in more time than the event-functions. Some OS supports and some do not support event functions. The event-functions enable OS actions after a group of events. The OS event functions can be understood as follows:

There is an event register. It has 8 or 16 or 32 event-flags, which form the groups. Each bit of the group in the register corresponds to one event flag in a set of flags.

An event register creates using an OS function, OSEventCreate ( ). An event register can be divided into groups, each group assigned to different tasks. For example, a 16-bit register can be divided into four groups. Group 0 is from bit 0 to bit 3, group 1 from bit 4 to bit 7, group 2 is from bit 8 to bit 11 and group 3 is from bit 12 to bit 15. An OS function, OSEventQuery ( ) queries an event register to find the event register existence and its contents. An event register deletes using an OS function, OSEventDelete ( ).

Each event sets one of the bits at the event register using the SET (eventFlag) function. Event flag in the register can be set by an ISR or task. CLEAR (eventFlag) clears the flag in the event register. An event flag can be cleared by an ISR or task.

A task can use the WAIT_ALL function for the occurrences of setting all the event flags in a group. [Wait till AND operation between all flags in the group equals to *true*.] The task can use WAIT_ANY function for an occurrence of setting of any of the event flags in the group. [Wait till OR operation between all flags in the group equals to true.]

8.5 MEMORY MANAGEMENT

8.5.1 Memory Allocation

When a process is created, the memory manager allocates the memory addresses (blocks) to it by mapping the process address space (Section 7.1). Threads of a process share the memory space of the process (Section 7.2).

8.5.2 Memory Management after Initial Allocation

The memory manager of the OS has to be secure, robust and well protected. There must be control such that there are no memory leaks and stack overflows. Memory leaks mean attempts to write in the memory block not allocated to a process or data structure. Stack overflow means that the stack exceeds the allocated memory block(s) when there is no provision for additional stack space. Table 8.5 gives memory-management strategy.

Example 8.6

An OS provides for dynamic memory allocation and de-allocation functions. Dynamic memory allocation is used for creating memory address space for a buffer or a messages queue or some other purpose during execution of a task. Dynamic memory de-allocation is used for freeing the memory taken up for the buffer during execution of the task.

Consider fragmented physical memory allocations. Fragmented means that memory addresses in two variable-size blocks of a process are not continuous. When a block of memory address is allocated, the time is spent in first locating the next free memory address before allocating that to the process. A standard memory allocation scheme scans a linked list of indeterminate length to find a suitable free memory block. When one allotted block of memory is de-allocated, the time is spent in first locating the next allocated memory block before de-allocating that to the process. The time for allocation and de-allocation of the memory and blocks are variable (not deterministic) when the block sizes are variable and when the memory is fragmented. In RTOS, this leads to unpredictable task-performance (run-time).

RTOS μCOS-II provides for memory partitioning. A task must create a memory partition or several memory partitions by using function OSMemCreate ( ). Then the task is permitted to use the partition or partitions. A partition has several memory blocks. A task gets a memory block or blocks from the partition by using function OSMemGet ( ). A task releases a memory block or blocks to the partition by using function OSMemPut ( ). Therefore, the task consists of several fixed size memory blocks. The fixed size memory blocks allocation and de-allocation time takes fixed time (deterministic). Therefore, it leads to a predictable task-performance.

Table 8.5 Memory Managing Strategy for a System

Managing Strategy	*Explanation*
Fixed blocks allocation	Memory address space is divided into blocks with processes having small address spaces getting a lesser number of blocks and processes with big address spaces getting a larger number of blocks.
Dynamic blocks allocation[1]	Memory address space is divided into fixed blocks as above and then later the memory manager later allocates variable size blocks (in units of say 64 or 256 bytes) dynamically allocated from a free (unused) list of memory blocks description table at the different computation phases of a process.
Dynamic page allocation[1]	Memory has fixed sized blocks called pages and the memory manager MMU (memory management unit) allocates the pages dynamically with a page descriptor table.
Dynamic data memory allocation	The manager allocates memory dynamically to different data structures like the nodes of a list, queues and stacks.
Dynamic address relocation[1]	The manager dynamically allocates the addresses initially bound to the relative addresses. It adds the relative address to address with relocation register. The memory manager now dynamically changes only the contents of relocation register. It also takes into account a limit-defining register so that the relocated addresses are within the limit of available addresses. This is also called run-time dynamic address binding.
Multiprocessor memory allocation	Refer to Section 6.3. The memory adopts an allocation strategy either shared with tight coupling between two or more processors or shared with loose coupling or multisegmented allocations.
Memory protection to OS functions[2]	Memory protection to the OS functions means that the system call (call to an OS function) and function call in user space are distinct. The OS function code, data and stack are in the protected memory area. It means that when a user function call attempts to write or read in the exclusive memory space allocated to the OS functions, it is blocked and the system generates an error. The memory of kernel functions is distinct and can be addressed only by the systems calls. The memory space is called kernel space.
Memory protection among the tasks[2]	Memory protection to the tasks means that a task function call cannot attempt to write or read in the exclusive area of memory space allocated to another task. The protection increases the memory requirement for each task and also the execution time of the code of the task.

[1] RTOS may disable the support to the dynamic block allocation, MMU support to the dynamic page allocation and dynamic binding as this increases the latency of servicing the tasks and ISRs.

[2] RTOS may not support memory protection of the OS functions from user function calls, as this increases the latency of servicing the tasks and ISRs. The user functions are then runnable in kernel space and run like kernel functions.

[3] RTOS may provide for disabling of the support to memory protection among the tasks as this increases the memory requirement for each task.

The memory manager manages the following: (i) use of memory address space by a process, (ii) specific mechanisms to share the memory space, (iii) specific mechanisms to restrict sharing of a given memory space and (iv) optimization of the access periods of a memory by using an hierarchy of memories (caches, primary and external secondary magnetic and optical memories). Remember that the access periods are in the following increasing order: caches, primary and external secondary magnetic and then optical.

The memory manager allocates memory to the processes and manages it with appropriate protection. There may be static and dynamic allocations of memory. The manager optimizes the memory needs and memory utilization. An RTOS may disable the support to the dynamic block allocation, MMU support to the dynamic page allocation and dynamic binding as this increases the latency of srvicing the tasks and ISRs. An RTOS may or may not support memory protection in order to reduce the latency and memory needs of the processes.

8.6 DEVICE, FILE AND IO SUBSYSTEMS MANAGEMENT

8.6.1 Device Management

Recall Section 4.2. There are number of device drive ISRs for each device in a system, each driver-function of a device (e.g. open, close, read) calls a separate ISR. Device managr (inside or outside the kernel space) is the software that manages these for all. When device driver functions are a part of the OS (inside or outside the kernel space), the device manager effectively operates and adopts appropriate strategy for obtaining optimal performance for the devices. The manager coordinates between application process, driver- and device-controller. A process sends a request to the driver functions by an interrupt using SWI; and the driver provides the actions on calling and executing the ISR. The device manager polls the requests at the devices and the actions occur as per their priorities. The device manager manages IO interrupt (requests) queues. The device manager creates an appropriate kernel interface and API, and that activates the control register-specific actions of the device. The device controller is activated through the API and kernel interface (recall Section 1.4.6). An OS device manager provides and executes the modules for managing the devices and their driver ISRs.

1. It manages the physical as well as virtual devices like the pipes and sockets through a common strategy.
2. Device management has three standard approaches to three types of device drivers: (i) programmed I/Os by polling the service need from each device; (ii) interrupt(s) from the device driver ISR and (iii) DMA operation used by the devices to access the memory. Most common is the use of device driver ISRs.
3. A device manager has the functions given in Table 8.6.

Table 8.6 Functions of a Device Manager

Function	*Action(s)*
Device detection and addition	Provides the codes for detecting the presence of various devices, then adding (initializing, configuring and testing) them for the use of OS device driver functions. A manager can provide for tracking the hardware inventory (list of devices present in the system and connected to the system).
Device deletion	Provides the codes for denying the device resources.
Device allocation and registration	Allocates and registers the port (it may be a register or memory) addresses for the various devices at distinctly different addresses and also includes codes for detecting any collision between them.

(Contd)

Function	*Action(s)*
Detaching and deregistration	Detaches and deregisters the port (it may be a register or memory) addresses for the various devices at distinctly different addresses and also includes codes for detecting any collision between existing addresses in case of addresses reallocation to the remaining attached (registered) devices.
Restricting device to a specific process	Restricts a device access to one process (task) only, at an instant.
Device sharing	Permits sharing of access of a device to the set of processes, but to one process (task) at an instant.
Device control	A manager can also provide for remote control of the devices from the remote server at the service provider. (For example, mobile devices with server at the service-provider)
Device access management	(i) sequential access, (ii) random access, (iii) semi-random access, (iv) serial communication may be by UART or USB, and (v) 4 (or more) serial bits in parallel during IOs (for example, SDIO) (Chapter 3). The device manager provides the necessary interface.
Device buffer management	Device hardware may merely have a single byte buffer, or double buffer or 8-byte buffer. A device buffer manager uses a memory manager to buffer the I/O data streams from the device that sends the data and manages computations without wait while the buffer receives the data at a slow rate.[1] Also used are the multiple buffers and producer–consumer-type bounded buffers (Section 7.7.6).
Device queue, circular-queue or blocks of queue management	Device IO data streams from the device can be organized as the queues, circular queues and blocks of queues (Section 5.4.3).
Device driver	A manager manages the device drivers. A device driver or a software driver is the software for interface with the device hardware through the buses on the one hand and for interface with the OS and application on other hand. The software commands for read and write enables the read and write functions through the ISRs called by using the SWIs.[2] The interface sofware to OS enables the creation, connection, binding, opening and closing of the device[3] (Section 4.1).
Device drivers updating and uploading of new device functions	A manager can also provide for updating the driver software from the Internet and uploading the new device functions, which become available at a later date.
Backup and restoration	A manager can also provide for the backup and restoration for drivers.

[1] For example, when the computations for deciphering the input data is slower than the receiving data in the buffer, the buffer(s) will soon choke. When the computations for deciphering the input data is faster than the receiving data in the buffer, the computations will wait for data in buffer.

[2] For hardware devices, a device ISR can also be called a system ISR or a system interrupt handler.

[3] For example, Unix device driver components are: (i) device ISR, (ii) device initialization codes (codes for configuring device control registers) and (iii) system initialization codes, which run just after the system resets (at bootstrapping).

Table 8.7 gives the set of OS command functions for a device.