The above application tries to connect to a server machine with IP address 172.168.0.1 and port number 5000. Change the values of *SERVER* and *PORT* to connect to a machine with different IP address and port number. If the connection is success, it sends the data "Hi from Client" to the server and waits for a response from the server and finally terminates the connection.

Under Windows, the socket function library *Winsock* should be initiated before using the socket related functions. The function *WSAStartup* performs this initiation. The *socket()* function call creates a socket. The socket type, connection type and protocols for communication are the parameters for socket creation. Here the socket type is *INET* (*AF_INET*) and connection type is stream socket (*SOCK_STREAM*). The protocol selected for communication is *TCP/IP* (*IPPROTO_TCP*). After creating the socket it is connected to a server. For connecting to server, the server address and port number should be indicated in the connection request. The *sockaddr_in* structure specifies the socket type, IP address and port of the server to be connected to. The *connect ()* function connects the socket with a specified server. If the server grants the connection request, the *connect ()* function returns success. The *send()* function is used for sending data to a server. It takes the socket name and data to be sent as parameters. Data from server is received using the function call *recv()*. It takes the socket name and details of buffer to hold the received data as parameters. The TCP/IP network stack expects network byte order (Big Endian: Higher order byte of the data is stored in lower memory address location) for data. The function *htons()* converts the byte order of an unsigned short integer to the network order. The *closesocket()* function closes the socket connection. On the server side, the server creates a socket using the function *socket()* and binds the socket with a port using the *bind()* function. It listens to the port bonded to the socket for any incoming connection request. The function *listen()* performs this. Upon receiving a connection request, the server accepts it. The function *accept()* performs the accepting operation. Now the connectivity is established. Server can receive and transmit data using the function calls *recv()* and *send()* respectively. The implementation of the server application is left to the readers as an exercise.

10.8 TASK SYNCHRONISATION

LO 8 State the need for task synchronisation in a multitasking environment

In a multitasking environment, multiple processes run concurrently (in pseudo parallelism) and share the system resources. Apart from this, each process has its own boundary wall and they communicate with each other with different IPC mechanisms including shared memory and variables. Imagine a situation where two processes try to access display hardware connected to the system or two processes try to access a shared memory area where one process tries to write to a memory location when the other process is trying to read from this. What could be the result in these scenarios? Obviously unexpected results. How these issues can be addressed? The solution is, make each process aware of the access of a shared resource either directly or indirectly. The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as '*Task/Process Synchronisation*'. Various synchronisation issues may arise in a multitasking environment if processes are not synchronised properly. The following sections describe the major task communication synchronisation issues observed in multitasking and the commonly adopted synchronisation techniques to overcome these issues.

10.8.1 Task Communication/Synchronisation Issues

10.8.1.1 Racing

Let us have a look at the following piece of code:

```
#include "stdafx.h"
#include <windows.h>
```

```
#include <stdio.h>
//************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two processes Process A and Process B
char Buffer[10] = { 1,2,3,4,5,6,7,8,9,10 };
short int counter = 0;
//************************************************************
// Process A
void Process_A(void) {
 int i;
 for (i = 0; i<5; i++)
 {
  if (Buffer[i] > 0)
   counter++;
 }
}
//************************************************************
// Process B
void Process_B(void) {
 int j;
 for (j = 5; j<10; j++)
 {
  if (Buffer[j] > 0)
   counter++;
 }
}
//************************************************************
//Main Thread.
int main() {
 DWORD id;
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_A, (LPVOID)0, 0, &id);
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_B, (LPVOID)0, 0, &id);
 Sleep(100000);
 return 0;
}
```

From a programmer perspective the value of counter will be 10 at the end of execution of processes A & B. But 'it need not be always' in a real world execution of this piece of code under a multitasking kernel. The results depend on the process scheduling policies adopted by the OS kernel. Now let's dig into the piece of code illustrated above. The program statement counter++; looks like a single statement from a high level programming language ('C' language) perspective. The low level implementation of this statement is dependent on the underlying processor instruction set and the (cross) compiler in use. The low level implementation of the high level program statement counter++; under Windows XP operating system running on an Intel Centrino Duo processor is given below. The code snippet is compiled with Microsoft Visual Studio 6.0 compiler.

```
mov eax,dword ptr [ebp-4];Load counter in Accumulator
add eax,1 ; Increment Accumulator by 1
mov dword ptr [ebp-4],eax ;Store counter with Accumulator
```

Whereas the same high level program statement when compiled with Visual Studio 2013 on an Intel i7 dual core Processor running Windows 10 OS yields the following low level implementation for the program statement ***counter++***

```
mov  ax,word ptr ds:[00A08140h]
add  ax,1
mov  word ptr ds:[00A08140h],ax
```

In the first scenario, at the processor instruction level, the value of the variable counter is loaded to the Accumulator register (EAX register). The memory variable counter is represented using a pointer. The base pointer register (EBP register) is used for pointing to the memory variable counter. After loading the contents of the variable counter to the Accumulator, the Accumulator content is incremented by one using the add instruction. Finally the content of Accumulator is loaded to the memory location which represents the variable counter. Both the processes Process A and Process B contain the program statement counter++; Translating this into the machine instruction

Process A	Process B
mov eax,dword ptr [ebp-4]	mov eax,dword ptr [ebp-4]
add eax,1	add eax,1
mov dword ptr [ebp-4],eax	mov dword ptr [ebp-4],eax

Imagine a situation where a process switching (context switching) happens from Process A to Process B when Process A is executing the *counter++;* statement. Process A accomplishes the *counter++;* statement through three different low level instructions. Now imagine that the process switching happened at the point where Process A executed the low level instruction, 'mov eax,dword ptr [ebp-4]' and is about to execute the next instruction 'add eax,1'. The scenario is illustrated in Fig. 10.23.

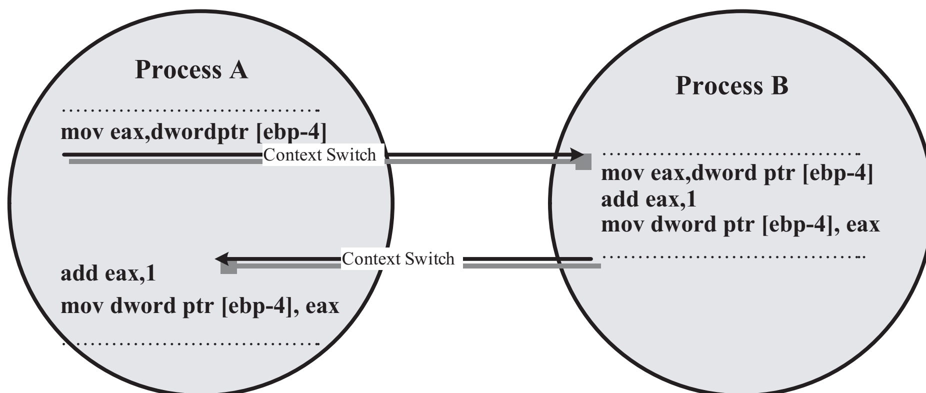

Fig. 10.23 Race condition

Process B increments the shared variable 'counter' in the middle of the operation where Process A tries to increment it. When Process A gets the CPU time for execution, it starts from the point where it got interrupted (If Process B is also using the same registers *eax* and *ebp* for executing *counter++;* instruction, the original content of these registers will be saved as part of the context saving and it will be retrieved back

as part of context retrieval, when process A gets the CPU for execution. Hence the content of *eax* and *ebp* remains intact irrespective of context switching). Though the variable *counter* is incremented by Process B, Process A is unaware of it and it increments the variable with the old value. This leads to the loss of one increment for the variable *counter*. This problem occurs due to non-atomic[§] operation on variables. This issue wouldn't have been occurred if the underlying actions corresponding to the program statement *counter++;* is finished in a single CPU execution cycle. The best way to avoid this situation is make the access and modification of shared variables mutually exclusive; meaning when one process accesses a shared variable, prevent the other processes from accessing it. We will discuss this technique in more detail under the topic '*Task Synchronisation techniques*' in a later section of this chapter.

To summarise, *Racing* or *Race condition* is the situation in which multiple processes compete (race) each other to access and manipulate shared data concurrently. In a Race condition the final value of the shared data depends on the process which acted on the data finally.

10.8.1.2 Deadlock

A race condition produces incorrect results whereas a deadlock condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlocked processes. A situation very similar to our traffic jam issues in a junction as illustrated in Fig. 10.24.

Fig. 10.24 Deadlock visualisation

In its simplest form 'deadlock' is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process (Fig. 10.25). To elaborate: Process A holds a resource *x* and it wants a resource *y* held by Process B. Process B is currently holding resource *y* and it wants the resource *x* which is currently held by Process A. Both hold the respective resources and they compete each other to get the resource held by the respective processes. The result of the competition is 'deadlock'. None of the competing process will be able to access the resources held by other processes since they are locked by the respective processes (If a mutual exclusion policy is implemented for shared resource access, the resource is locked by the process which is currently accessing it).

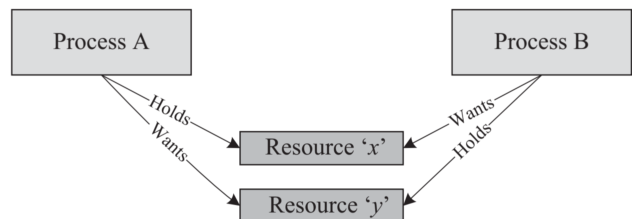

Fig. 10.25 Scenarios leading to deadlock

The different conditions favouring a deadlock situation are listed below.

Mutual Exclusion: The criteria that only one process can hold a resource at a time. Meaning processes should access shared resources with mutual exclusion. Typical example is the accessing of display hardware in an embedded device.

§ Atomic Operation: Operations which are non-interruptible.

Hold and Wait: The condition in which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.

No Resource Preemption: The criteria that operating system cannot take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process holding it.

Circular Wait: A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process. In general, there exists a set of waiting process P0, P1 … P*n* with P0 is waiting for a resource held by P1 and P1 is waiting for a resource held by P0, …, P*n* is waiting for a resource held by P0 and P0 is waiting for a resource held by P*n* and so on… This forms a circular wait queue.

'Deadlock' is a result of the combined occurrence of these four conditions listed above. These conditions are first described by E. G. Coffman in 1971 and it is popularly known as *Coffman conditions*.

Deadlock Handling A smart OS may foresee the deadlock condition and will act proactively to avoid such a situation. Now if a deadlock occurred, how the OS responds to it? The reaction to deadlock condition by OS is nonuniform. The OS may adopt any of the following techniques to detect and prevent deadlock conditions.

Ignore Deadlocks: Always assume that the system design is deadlock free. This is acceptable for the reason the cost of removing a deadlock is large compared to the chance of happening a deadlock. UNIX is an example for an OS following this principle. A life critical system cannot pretend that it is deadlock free for any reason.

Detect and Recover: This approach suggests the detection of a deadlock situation and recovery from it. This is similar to the deadlock condition that may arise at a traffic junction. When the vehicles from different directions compete to cross the junction, deadlock (traffic jam) condition is resulted. Once a deadlock (traffic jam) is happened at the junction, the only solution is to back up the vehicles from one direction and allow the vehicles from opposite direction to cross the junction. If the traffic is too high, lots of vehicles may have to be backed up to resolve the traffic jam. This technique is also known as 'back up cars' technique (Fig. 10.26).

Operating systems keep a resource graph in their memory. The resource graph is updated on each resource request and release. A deadlock condition can be detected by analysing the resource graph by graph analyser algorithms. Once a deadlock condition is detected, the system can terminate a process or preempt the resource to break the deadlocking cycle.

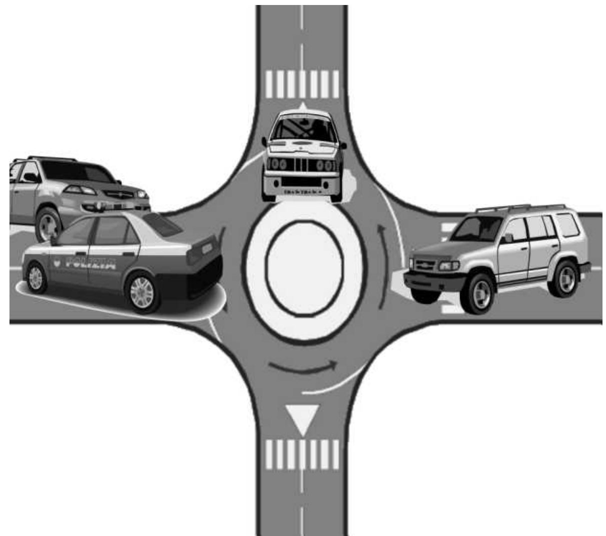

Fig. 10.26 'Back up cars' **technique for deadlock recovery**

Avoid Deadlocks: Deadlock is avoided by the careful resource allocation techniques by the Operating System. It is similar to the traffic light mechanism at junctions to avoid the traffic jams.

Prevent Deadlocks: Prevent the deadlock condition by negating one of the four conditions favouring the deadlock situation.

- Ensure that a process does not hold any other resources when it requests a resource.

This can be achieved by implementing the following set of rules/guidelines in allocating resources to processes.

1. A process must request all its required resource and the resources should be allocated before the process begins its execution.
2. Grant resource allocation requests from processes only if the process does not hold a resource currently.

- Ensure that resource preemption (resource releasing) is possible at operating system level. This can be achieved by implementing the following set of rules/guidelines in resources allocation and releasing.
  1. Release all the resources currently held by a process if a request made by the process for a new resource is not able to fulfil immediately.
  2. Add the resources which are preempted (released) to a resource list describing the resources which the process requires to complete its execution.
  3. Reschedule the process for execution only when the process gets its old resources and the new resource which is requested by the process.

Imposing these criterions may introduce negative impacts like low resource utilisation and starvation of processes.

Livelock The *Livelock* condition is similar to the deadlock condition except that a process in livelock condition changes its state with time. While in deadlock a process enters in wait state for a resource and continues in that state forever without making any progress in the execution, in a livelock condition a process always does something but is unable to make any progress in the execution completion. The livelock condition is better explained with the real world example, two people attempting to cross each other in a narrow corridor. Both the persons move towards each side of the corridor to allow the opposite person to cross. Since the corridor is narrow, none of them are able to cross each other. Here both of the persons perform some action but still they are unable to achieve their target, cross each other. We will make the livelock, the scenario more clear in a later section–*The Dining Philosophers' Problem*, of this chapter.

Starvation In the multitasking context, *starvation* is the condition in which a process does not get the resources required to continue its execution for a long time. As time progresses the process starves on resource. Starvation may arise due to various conditions like byproduct of preventive measures of deadlock, scheduling policies favouring high priority tasks and tasks with shortest execution time, etc.

10.8.1.3 The Dining Philosophers' Problem

The '*Dining philosophers' problem*' is an interesting example for synchronisation issues in resource utilisation. The terms 'dining', 'philosophers', etc. may sound awkward in the operating system context, but it is the best way to explain technical things abstractly using non-technical terms. Now coming to the problem definition:

Five philosophers (It can be '*n*'. The number 5 is taken for illustration) are sitting around a round table, involved in eating and brainstorming (Fig. 10.27). At any point of time each philosopher will be in any one of the three states: eating, hungry or brainstorming. (While eating the philosopher is not involved in brainstorming and while brainstorming the philosopher is not involved in eating). For eating, each philosopher requires 2 forks. There are only 5 forks available on the dining table ('*n*' for '*n*' number of philosophers) and they are arranged in a fashion one fork in between two philosophers. The philosopher can only use the forks on his/her immediate left and right that too in the order pickup the left fork first and then the right fork. Analyse the situation and explain the possible outcomes of this scenario.

Let's analyse the various scenarios that may occur in this situation.

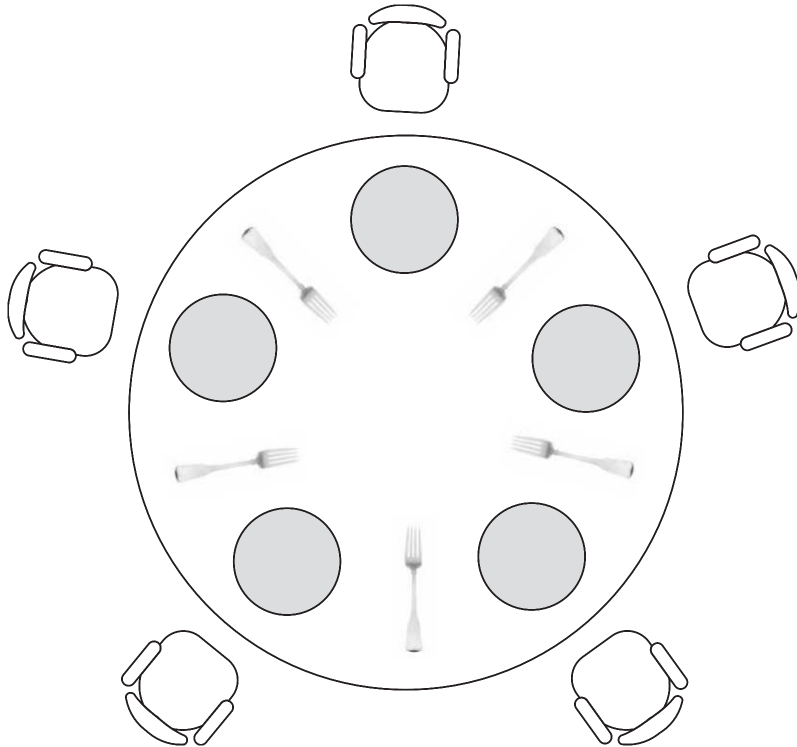

Fig. 10.27 Visualisation of the 'Dining Philosophers problem**'**

Scenario 1: All the philosophers involve in brainstorming together and try to eat together. Each philosopher picks up the left fork and is unable to proceed since two forks are required for eating the spaghetti present in the plate. Philosopher 1 thinks that Philosopher 2 sitting to the right of him/her will put the fork down and waits for it. Philosopher 2 thinks that Philosopher 3 sitting to the right of him/her will put the fork down and waits for it, and so on. This forms a circular chain of un-granted requests. If the philosophers continue in this state waiting for the fork from the philosopher sitting to the right of each, they will not make any progress in eating and this will result in *starvation* of the philosophers and *deadlock*.

Scenario 2: All the philosophers start brainstorming together. One of the philosophers is hungry and he/she picks up the left fork. When the philosopher is about to pick up the right fork, the philosopher sitting to his right also become hungry and tries to grab the left fork which is the right fork of his neighbouring philosopher who is trying to lift it, resulting in a '*Race condition*'.

Scenario 3: All the philosophers involve in brainstorming together and try to eat together. Each philosopher picks up the left fork and is unable to proceed, since two forks are required for eating the spaghetti present in the plate. Each of them anticipates that the adjacently sitting philosopher will put his/her fork down and waits for a fixed duration and after this puts the fork down. Each of them again tries to lift the fork after a fixed duration of time. Since all philosophers are trying to lift the fork at the same time, none of them will be able to grab two forks. This condition leads to *livelock* and *starvation* of philosophers, where each philosopher tries to do something, but they are unable to make any progress in achieving the target.

Figure 10.28 illustrates these scenarios.

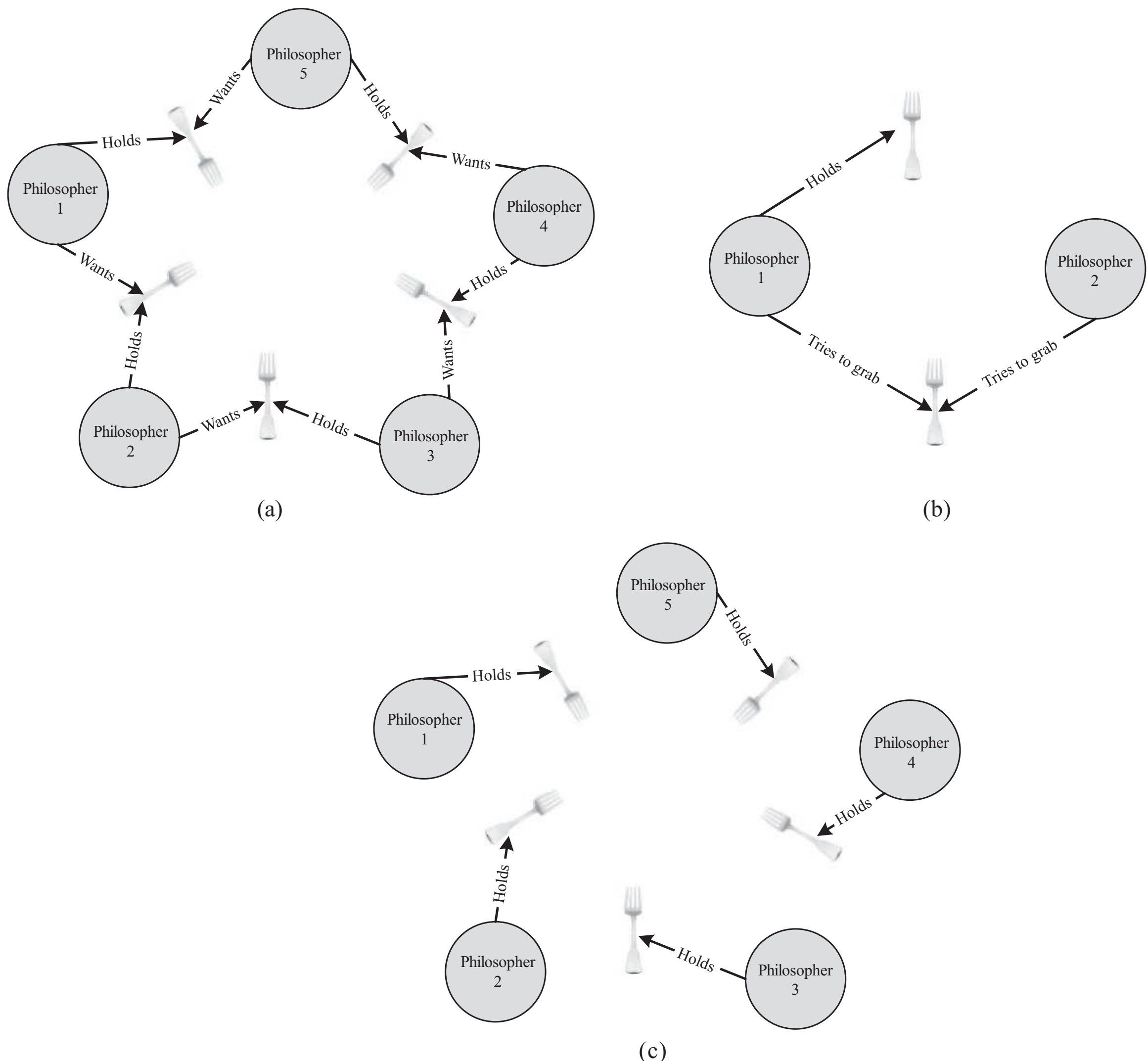

Fig. 10.28 The 'Real Problems' in the 'Dining Philosophers problem' (a) Starvation and Deadlock (b) Racing (c) Livelock and Starvation

Solution: We need to find out alternative solutions to avoid the *deadlock*, *livelock*, *racing* and *starvation* condition that may arise due to the concurrent access of forks by philosophers. This situation can be handled in many ways by allocating the forks in different allocation techniques including Round Robin allocation, FIFO allocation, etc. But the requirement is that the solution should be optimal, avoiding deadlock and starvation of the philosophers and allowing maximum number of philosophers to eat at a time. One solution that we could think of is:

- Imposing rules in accessing the forks by philosophers, like: The philosophers should put down the fork he/she already have in hand (left fork) after waiting for a fixed duration for the second fork (right fork) and should wait for a fixed time before making the next attempt.

This solution works fine to some extent, but, if all the philosophers try to lift the forks at the same time, a *livelock* situation is resulted.

Another solution which gives maximum concurrency that can be thought of is each philosopher acquires a semaphore (mutex) before picking up any fork. When a philosopher feels hungry he/she checks whether the philosopher sitting to the left and right of him is already using the fork, by checking the state of the associated semaphore. If the forks are in use by the neighbouring philosophers, the philosopher waits till the forks are available. A philosopher when finished eating puts the forks down and informs the philosophers sitting to his/her left and right, who are hungry (waiting for the forks), by signalling the semaphores associated with the forks. We will discuss about semaphores and mutexes at a latter section of this chapter. In the operating system context, the dining philosophers represent the processes and forks represent the resources. The dining philosophers' problem is an analogy of processes competing for shared resources and the different problems like racing, deadlock, starvation and livelock arising from the competition.

10.8.1.4 Producer-Consumer/Bounded Buffer Problem

Producer-Consumer problem is a common data sharing problem where two processes concurrently access a shared buffer with fixed size. A thread/process which produces data is called '*Producer thread/process*' and a thread/process which consumes the data produced by a producer thread/process is known as '*Consumer thread/process*'. Imagine a situation where the producer thread keeps on producing data and puts it into the buffer and the consumer thread keeps on consuming the data from the buffer and there is no synchronisation between the two. There may be chances where in which the producer produces data at a faster rate than the rate at which it is consumed by the consumer. This will lead to '*buffer overrun*' where the producer tries to put data to a full buffer. If the consumer consumes data at a faster rate than the rate at which it is produced by the producer, it will lead to the situation '*buffer under-run*' in which the consumer tries to read from an empty buffer. Both of these conditions will lead to inaccurate data and data loss. The following code snippet illustrates the producer-consumer problem

```
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#define N 20       //Define buffer size as 20
int buffer[N];     //Shared buffer for producer & consumer

//**********************************************************************
//Producer thread
void producer_thread(void) {
 int x;
 while (true) {
  for (x = 0; x<N; x++)
  {
   //Fill buffer with random data
   buffer[x] = rand() % 1000;
   printf("Produced : Buffer[%d] = % 4d\n", x, buffer[x]);
   Sleep(25);
  }
 }
}
//**********************************************************************
//Consumer thread
```

```
void consumer_thread(void) {
 int y = 0, value;
 while (true) {
  for (y = 0; y<N; y++)
  {
   value = buffer[y];
   printf("Consumed : Buffer[%d] = % 4d\n", y, value);
   Sleep(20);
  }
 }
}
//*************************************************************
//Main Thread
int main()
{
 DWORD thread_id;
 //Create Producer thread
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)producer_thread, NULL, 0,
 &thread_id);
 //Create Consumer thread
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)consumer_thread, NULL, 0,
 &thread_id);
 //Wait for some time and exit
 Sleep(500);
 return 0;
}
```

Here the 'producer thread' produces random numbers and puts it in a buffer of size 20. If the 'producer thread' fills the buffer fully it re-starts the filling of the buffer from the bottom. The 'consumer thread' consumes the data produced by the 'producer thread'. For consuming the data, the 'consumer thread' reads the buffer which is shared with the 'producer thread'. Once the 'consumer thread' consumes all the data, it starts consuming the data from the bottom of the buffer. These two threads run independently and are scheduled for execution based on the scheduling policies adopted by the OS. The different situations that may arise based on the scheduling of the 'producer thread' and 'consumer thread' is listed below.

1. 'Producer thread' is scheduled more frequently than the 'consumer thread': There are chances for overwriting the data in the buffer by the 'producer thread'. This leads to inaccurate data.
2. Consumer thread' is scheduled more frequently than the 'producer thread': There are chances for reading the old data in the buffer again by the 'consumer thread'. This will also lead to inaccurate data.

The output of the above program when executed on a Windows 10 machine is shown in Fig. 10.29.

The output shows that the consumer thread runs faster than the producer thread and most often leads to buffer under-run and thereby inaccurate data.

Note: It should be noted that the scheduling of the threads '*producer_thread*' and '*consumer_thread*' is OS kernel scheduling policy dependent and you may not get the same output all the time when you run this piece of code under Windows NT kernel (Say Windows 10 OS).

```
D:\ES\Samples\Sample\Debug\Sample.exe
Produced : Buffer [0] =   41
Consumed : Buffer [0] =   41
Consumed : Buffer [1] =    0
Produced : Buffer [1] =  467
Consumed : Buffer [2] =    0
Produced : Buffer [2] =  334
Consumed : Buffer [3] =    0
Produced : Buffer [3] =  500
Consumed : Buffer [4] =    0
Produced : Buffer [4] =  169
Consumed : Buffer [5] =    0
Consumed : Buffer [6] =    0
Produced : Buffer [5] =  724
Consumed : Buffer [7] =    0
Produced : Buffer [6] =  478
Consumed : Buffer [8] =    0
Produced : Buffer [7] =  358
Consumed : Buffer [9] =    0
Produced : Buffer [8] =  962
Consumed : Buffer [10] =    0
Consumed : Buffer [11] =    0
Produced : Buffer [9] =  464
Consumed : Buffer [12] =    0
Produced : Buffer [10] =  705
Consumed : Buffer [13] =    0
Produced : Buffer [11] =  145
Consumed : Buffer [14] =    0
Produced : Buffer [12] =  281
Consumed : Buffer [15] =    0
Consumed : Buffer [16] =    0
Produced : Buffer [13] =  827
Consumed : Buffer [17] =    0
```

Fig. 10.29 Output of Win32 program illustrating producer consumer problem

The producer-consumer problem can be rectified in various methods. One simple solution is the '*sleep and wake-up*'. The '*sleep and wake-up*' can be implemented in various process synchronisation techniques like semaphores, mutex, monitors, etc. We will discuss it in a latter section of this chapter.

10.8.1.5 Readers-Writers Problem

The Readers-Writers problem is a common issue observed in processes competing for limited shared resources. The Readers-Writers problem is characterised by multiple processes trying to read and write shared data concurrently. A typical real-world example for the Readers-Writers problem is the banking system where one process tries to read the account information like available balance and the other process tries to update the available balance for that account. This may result in inconsistent results. If multiple processes try to read a shared data concurrently it may not create any impacts, whereas when multiple processes try to write and read concurrently it will definitely create inconsistent results. Proper synchronisation techniques should be applied to avoid the readers-writers problem. We will discuss about the various synchronisation techniques in a later section of this chapter.

10.8.1.6 Priority Inversion

Priority inversion is the byproduct of the combination of blocking based (lock based) process synchronisation and pre-emptive priority scheduling. '*Priority inversion*' is the condition in which a high priority task needs

to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task, and a medium priority task which doesn't require the shared resource continue its execution by preempting the low priority task (Fig. 10.30). Priority based preemptive scheduling technique ensures that a high priority task is always executed first, whereas the lock based process synchronisation mechanism (like mutex, semaphore, etc.) ensures that a process will not access a shared resource, which is currently in use by another process. The synchronisation technique is only interested in avoiding conflicts that may arise due to the concurrent access of the shared resources and not at all bothered about the priority of the process which tries to access the shared resource. In fact, the priority based preemption and lock based synchronisation are the two contradicting OS primitives. Priority inversion is better explained with the following scenario:

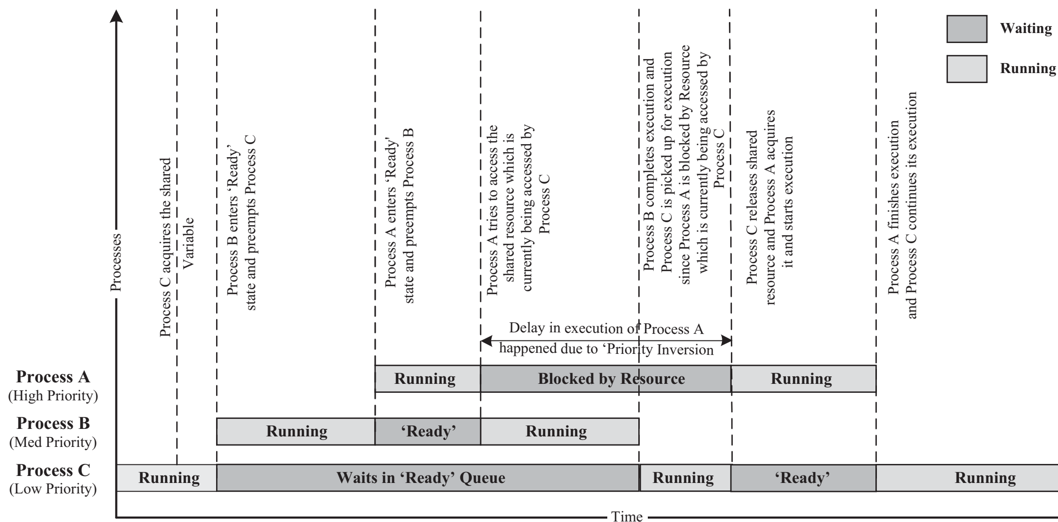

Fig. 10.30 Priority Inversion Problem

Let Process A, Process B and Process C be three processes with priorities High, Medium and Low respectively. Process A and Process C share a variable 'X' and the access to this variable is synchronised through a mutual exclusion mechanism like *Binary Semaphore S*. Imagine a situation where Process C is ready and is picked up for execution by the scheduler and 'Process C' tries to access the shared variable 'X'. 'Process C' acquires the 'Semaphore S' to indicate the other processes that it is accessing the shared variable 'X'. Immediately after 'Process C' acquires the 'Semaphore S', 'Process B' enters the 'Ready' state. Since 'Process B' is of higher priority compared to 'Process C', 'Process C' is preempted and 'Process B' starts executing. Now imagine 'Process A' enters the 'Ready' state at this stage. Since 'Process A' is of higher priority than 'Process B', 'Process B' is preempted and 'Process A' is scheduled for execution. 'Process A' involves accessing of shared variable 'X' which is currently being accessed by 'Process C'. Since 'Process C' acquired the semaphore for signalling the access of the shared variable 'X', 'Process A' will not be able to access it. Thus 'Process A' is put into blocked state (This condition is called Pending on resource). Now 'Process B' gets the CPU and it continues its execution until it relinquishes the CPU voluntarily or enters a wait state or preempted by another high priority task. The highest priority process 'Process A' has to wait till 'Process C' gets a chance to execute and release the semaphore. This produces unwanted delay in the execution of the high priority task which is supposed to be executed immediately when it was 'Ready'.

Priority inversion may be sporadic in nature but can lead to potential damages as a result of missing critical deadlines. Literally speaking, priority inversion 'inverts' the priority of a high priority task with that

of a low priority task. Proper workaround mechanism should be adopted for handling the priority inversion problem. The commonly adopted priority inversion workarounds are:

Priority Inheritance: A low-priority task that is currently accessing (by holding the lock) a shared resource requested by a high-priority task temporarily '*inherits*' the priority of that high-priority task, from the moment the high-priority task raises the request. Boosting the priority of the low priority task to that of the priority of the task which requested the shared resource holding by the low priority task eliminates the preemption of the low priority task by other tasks whose priority are below that of the task requested the shared resource and thereby reduces the delay in waiting to get the resource requested by the high priority task. The priority of the low priority task which is temporarily boosted to high is brought to the original value when it releases the shared resource. Implementation of Priority inheritance workaround in the priority inversion problem discussed for Process A, Process B and Process C example will change the execution sequence as shown in Fig. 10.31.

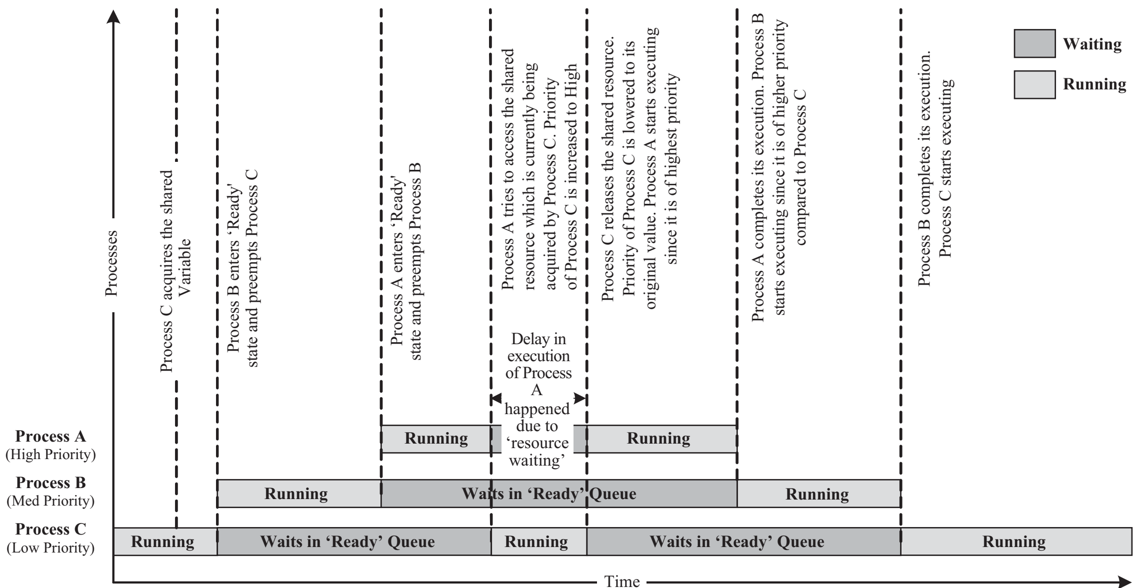

Fig. 10.31 Handling Priority Inversion Problem with Priority Inheritance

Priority inheritance is only a work around and it will not eliminate the delay in waiting the high priority task to get the resource from the low priority task. The only thing is that it helps the low priority task to continue its execution and release the shared resource as soon as possible. The moment, at which the low priority task releases the shared resource, the high priority task kicks the low priority task out and grabs the CPU – A true form of selfishness☺. Priority inheritance handles priority inversion at the cost of run-time overhead at scheduler. It imposes the overhead of checking the priorities of all tasks which tries to access shared resources and adjust the priorities dynamically.

Priority Ceiling: In '*Priority Ceiling*', a priority is associated with each shared resource. The priority associated to each resource is the priority of the highest priority task which uses this shared resource. This priority level is called 'ceiling priority'. Whenever a task accesses a shared resource, the scheduler elevates the priority of the task to that of the ceiling priority of the resource. If the task which accesses the shared resource is a low priority task, its priority is temporarily boosted to the priority of the highest priority task to which the

resource is also shared. This eliminates the pre-emption of the task by other medium priority tasks leading to priority inversion. The priority of the task is brought back to the original level once the task completes the accessing of the shared resource. 'Priority Ceiling' brings the added advantage of sharing resources without the need for synchronisation techniques like locks. Since the priority of the task accessing a shared resource is boosted to the highest priority of the task among which the resource is shared, the concurrent access of shared resource is automatically handled. Another advantage of 'Priority Ceiling' technique is that all the overheads are at compile time instead of run-time. Implementation of 'priority ceiling' workaround in the priority inversion problem discussed for Process A, Process B and Process C example will change the execution sequence as shown in Fig. 10.32.

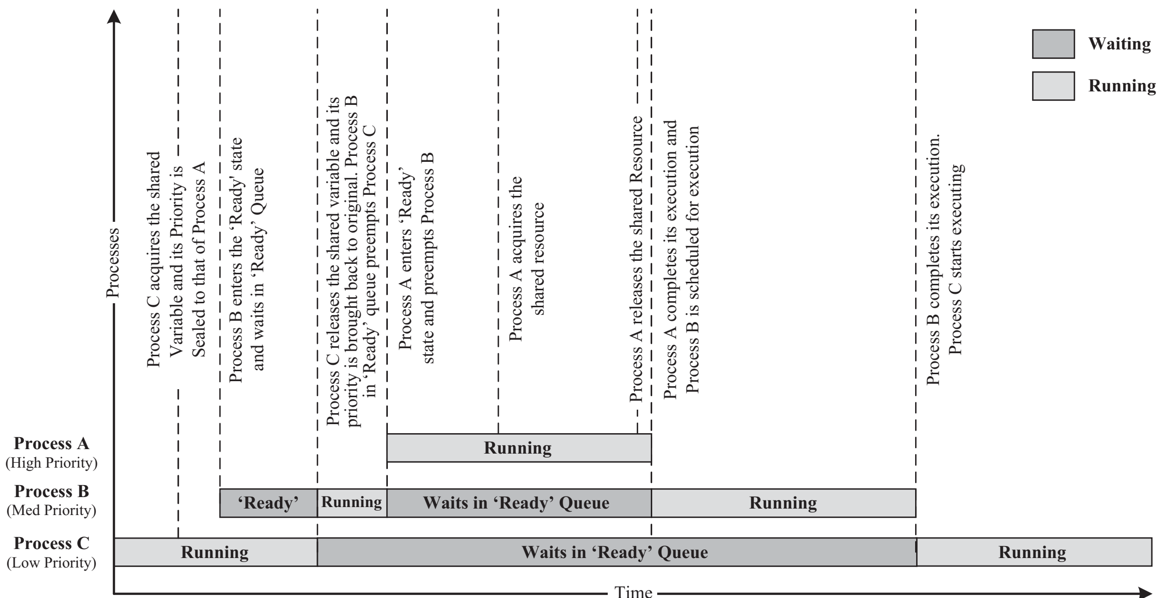

Fig. 10.32 Handling Priority Inversion Problem with Priority Ceiling

The biggest drawback of 'Priority Ceiling' is that it may produce *hidden priority inversion*. With 'Priority Ceiling' technique, the priority of a task is always elevated no matter another task wants the shared resources. This unnecessary priority elevation always boosts the priority of a low priority task to that of the highest priority tasks among which the resource is shared and other tasks with priorities higher than that of the low priority task is not allowed to preempt the low priority task when it is accessing a shared resource. This always gives the low priority task the luxury of running at high priority when accessing shared resources☺.

10.8.2 Task Synchronisation Techniques

So far we discussed about the various task/process synchronisation issues encountered in multitasking systems due to concurrent resource access. Now let's have a discussion on the various techniques used for synchronisation in concurrent access in multitasking. Process/Task synchronisation is essential for

1. Avoiding conflicts in resource access (racing, deadlock, starvation, livelock, etc.) in a multitasking environment.

2. Ensuring proper sequence of operation across processes. The producer consumer problem is a typical example for processes requiring proper sequence of operation. In producer consumer problem, accessing the shared buffer by different processes is not the issue, the issue is the writing process should write to the shared buffer only if the buffer is not full and the consumer thread should not read from the buffer if it is empty. Hence proper synchronisation should be provided to implement this sequence of operations.
3. Communicating between processes.

The code memory area which holds the program instructions (piece of code) for accessing a shared resource (like shared memory, shared variables, etc.) is known as '*critical section*'. In order to synchronise the access to shared resources, the access to the critical section should be exclusive. The exclusive access to critical section of code is provided through mutual exclusion mechanism. Let us have a look at how mutual exclusion is important in concurrent access. Consider two processes *Process* A and *Process B* running on a multitasking system. *Process A* is currently running and it enters its critical section. Before *Process A* completes its operation in the critical section, the scheduler preempts *Process A* and schedules *Process B* for execution (*Process B* is of higher priority compared to *Process A*). *Process B* also contains the access to the critical section which is already in use by *Process A*. If *Process B* continues its execution and enters the critical section which is already in use by *Process A*, a racing condition will be resulted. A mutual exclusion policy enforces mutually exclusive access of critical sections.

Mutual exclusions can be enforced in different ways. Mutual exclusion blocks a process. Based on the behaviour of the blocked process, mutual exclusion methods can be classified into two categories. In the following section we will discuss them in detail.

10.8.2.1 Mutual Exclusion through Busy Waiting/Spin Lock

'*Busy waiting*' is the simplest method for enforcing mutual exclusion. The following code snippet illustrates how 'Busy waiting' enforces mutual exclusion.

```
//Inside parent thread/main thread corresponding to a process
bool bFlag; //Global declaration of lock Variable.
bFlag= FALSE; //Initialise the lock to indicate it is available.
//..............................................................................................................................
//Inside the child threads/threads of a process
while(bFlag == TRUE); //Check the lock for availability
bFlag=TRUE; //Lock is available. Acquire the lock
//Rest of the source code dealing with shared resource access
```

The 'Busy waiting' technique uses a lock variable for implementing mutual exclusion. Each process/thread checks this lock variable before entering the critical section. The lock is set to '***1***' by a process/thread if the process/thread is already in its critical section; otherwise the lock is set to '***0***'. The major challenge in implementing the lock variable based synchronisation is the non-availability of a single atomic instruction[¶] which combines the reading, comparing and setting of the lock variable. Most often the three different operations related to the locks, viz. the operation of Reading the lock variable, checking its present value and setting it are achieved with multiple low level instructions. The low level implementation of these operations are dependent on the underlying processor instruction set and the (cross) compiler in use. The low level implementation of the 'Busy waiting' code snippet, which we discussed earlier, under Windows 10 Operating system running on an Intel i7 dual core Processor is given below. The code snippet is compiled with Microsoft Visual Studio 2013 compiler.

¶ Atomic Instruction: Instruction whose execution is uninterruptible.

```
--- d:\es\samples\rev1\counter.cpp ------------------------------------
1:   #include "stdafx.h"
2:   #include <stdio.h>
3:   #include <windows.h>
4:
5:   int main()
6:   {
//Code memory     Opcode       Operand
00A21380          push         ebp
00A21381          mov          ebp,esp
00A21383          sub          esp,0CCh
00A21389          push         ebx
00A2138A          push         esi
00A2138B          push         edi
00A2138C          lea          edi,[ebp+FFFFFF34h]
00A21392          mov          ecx,33h
00A21397          mov          eax,0CCCCCCCCh
00A2139C          rep stos     dword ptr es:[edi]
7:   //Inside parent thread/ main thread corresponding to a process
8:   bool bFlag; //Global declaration of lock Variable.
9:   bFlag = FALSE; //Initialise the lock to indicate it is available.
00A2139E          mov          byte ptr [ebp-5],0
10:  //..............................................................
11:  //Inside the child threads/ threads of a process
12:  while (bFlag == TRUE); //Check the lock for availability
00A213A2          movzx        eax,byte ptr [ebp-5]
00A213A6          cmp          eax,1
00A213A9          jne          main+2Dh (0A213ADh)
00A213AB          jmp          main+22h (0A213A2h)
13:  bFlag = TRUE; //Lock is available. Acquire the lock
00A213AD          mov          byte ptr [ebp-5],1
14:  //Rest of the source code dealing with shared resource access
```

The assembly language instructions reveals that the two high level instructions (*while(bFlag==false);* and *bFlag=true;*), corresponding to the operation of reading the lock variable, checking its present value and setting it is implemented in the processor level using five low level instructions. Imagine a hypothetical situation where 'Process 1' read the lock variable and tested it and found that the lock is available and it is about to set the lock for acquiring the critical section (Fig. 10.33). But just before 'Process 1' sets the lock variable, 'Process 2' preempts 'Process 1' and starts executing. 'Process 2' contains a critical section code and it tests the lock variable for its availability. Since 'Process 1' was unable to set the lock variable, its state is still '**0**' and 'Process 2' sets it and acquires the critical section. Now the scheduler preempts 'Process 2' and schedules 'Process 1' before 'Process 2' leaves the critical section. Remember, 'Process 1' was preempted at a point just before setting the lock variable ('Process 1' has already tested the lock variable just before it is preempted and found that the lock is available). Now 'Process 1' sets the lock variable and enters the critical section. It violates the mutual exclusion policy and may produce unpredicted results.

The above issue can be effectively tackled by combining the actions of reading the lock variable, testing its state and setting the lock into a single step. This can be achieved with the combined hardware and software support. Most of the processors support a single instruction '*Test and Set Lock (TSL)*' for testing and setting

the lock variable. The '*Test and Set Lock (TSL)*' instruction call copies the value of the lock variable and sets it to a nonzero value. It should be noted that the implementation and usage of '*Test and Set Lock (TSL)*' instruction is processor architecture dependent. The *Intel 486* and the above family of processors support the '*Test and Set Lock (TSL)*' instruction with a special instruction *CMPXCHG*—Compare and Exchange. The usage of *CMPXCHG* instruction is given below.

```
CMPXCHG dest,src
```

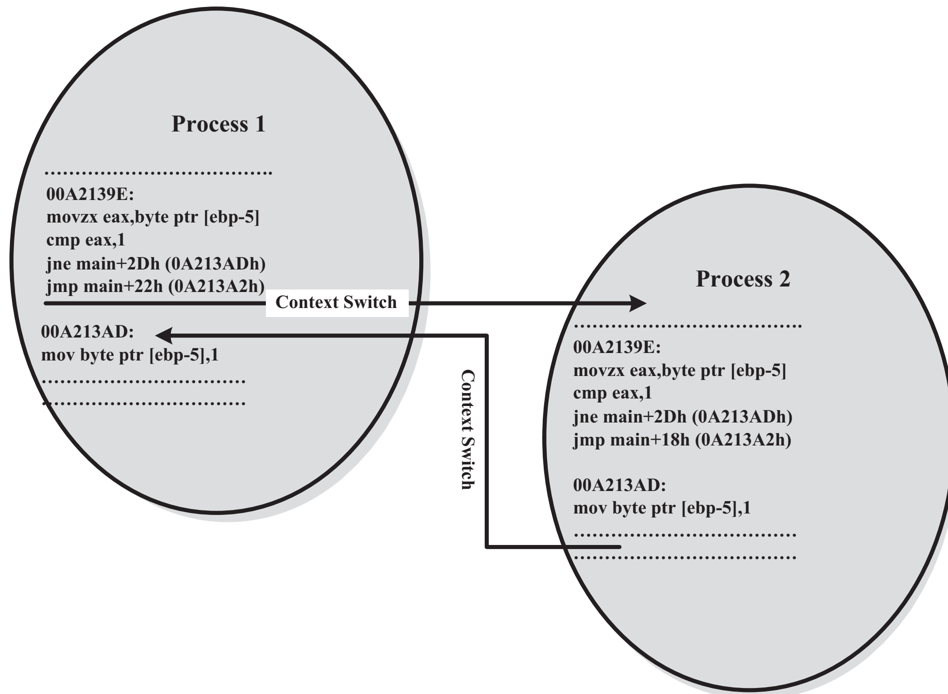

Fig. 10.33 Illustration of the issues with locks

This instruction compares the Accumulator (*EAX* register) with 'dest'. If the Accumulator and 'dest' contents are equal, 'dest' is loaded with 'src'. If not, the Accumulator is loaded with 'dest'. Executing this instruction changes the six status bits of the Program Control and Status register *EFLAGS*. The destination ('dest') can be a register or a memory location. The source ('src') is always a register. From a programmer's perspective the operation of *CMPXCHG* instruction can be viewed as:

```
if (accumulator == destination)
{
    ZF = 1; //Set the Zero Flag of EFLAGS Register
    destination = source;
}
else
{
    ZF = 0; //Reset the Zero Flag of EFLAGS Register
    accumulator = destination;
}
```

The process/thread checks the lock variable to see whether its state is '0' and sets its state to '1' if its state is '0', for acquiring the lock. To implement this at the 486 processor level, load the accumulator with '0' and a general purpose register with '1' and compare the memory location holding the lock variable with accumulator using *CMPXCHG* instruction. This instruction makes the accessing, testing and modification of the lock variable a single atomic instruction. How the *CMPXCHG* instruction support provided by the Intel® family of processors (486 and above) is made available to processes/threads is OS kernel implementation dependent. Let us see how this feature is implemented by Windows Operating systems. Windows Embedded Compact/Windows NT kernels support the compare and exchange hardware feature provided by Intel® family of processors, through the API call *InterlockedCompareExchange (LPLONG Destination, LONG Exchange, LONG Comperand)*. The variable *Destination* is the long pointer to the destination variable. The *Destination* variable should be of type '*long*'. The variable *Exchange* represents the exchange value. The value of *Destination* variable is replaced with the value of *Exchange* variable. The variable *Comperand* specifies the value which needs to be compared with the value of *Destination* variable. The function returns the initial value of the variable '*Destination*'. The following code snippet illustrates the usage of this API call for thread/process synchronisation.

```
//Inside parent thread/ main thread corresponding to a process
long bFlag; //Global declaration of lock Variable.
bFlag=0; //Initialise the lock to indicate it is available.
//..............................................................................
//Inside the child threads/ threads of a process
//Check the lock for availability & acquire the lock if available.
while (InterlockedCompareExchange (&bFlag, 1, 0) == 1);

//Rest of the source code dealing with shared resource access
```

The *InterlockedCompareExchange* function is implemented as '*Compiler intrinsic function*'. The 'code for *Compiler intrinsic functions*' are inserted inline while compiling the code. This avoids the function call overhead and makes use of the built-in knowledge of the optimisation technique for intrinsic functions. The compiler can be instructed to use the intrinsic implementation for a function using the compiler directive *#pragma intrinsic (intrinsic-function-name)*. A sample implementation of the *InterlockedCompareExchange* interlocked intrinsic function for desktop Windows OS is given below.

```
#include "stdafx.h"
#include <intrin.h>
#include <windows.h>
long bFlag; //Global declaration of lock Variable.
//Declare InterlockedCompareExchange as intrinsic function
#pragma intrinsic(_InterlockedCompareExchange)
void child_thread(void)
{
//Inside the child thread of a process
//Check the lock for availability & acquire the lock if available.
//The lock can be set by any other threads
while (_InterlockedCompareExchange (&bFlag, 1, 0) == 1);
//Rest of the source code dealing with shared resource access
//...................................................................
return;
```

```
}
//.............................................................
int _tmain(int argc, _TCHAR* argv[])
{
//Inside parent thread/ main thread corresponding to a process
DWORD thread_id;
//Define handle to the child thread

HANDLE tThread;
//Initialise the lock to indicate it is available.
bFlag =0;
//Create child thread
tThread = CreateThread  (NULL,0,
                        (LPTHREAD_START_ROUTINE) child_thread,
                        NULL, 0, &thread_id);
if(NULL== tThread)
{
//Child thread creation failed.
printf (“Creation of Child thread failed. Error Code =
%d”,GetLastError());
return -1;
}
//Wait for the completion of the child thread.
WaitForSingleObject(tThread,INFINITE);
return 0;
}
```

Note: Visual Studio 2005 or a later version of the compiler, which supports interlocked intrinsic functions, is required for compiling this application. The assembly code generated for the intrinsic interlocked function *while (_InterlockedCompareExchange (&bFlag, 1, 0) == 1);* when compiled using Visual Studio 2013 compiler, on Windows 10 platform running on an Intel® i7 Dual core processor is given below. It clearly depicts the usage of the cmpxchg instruction

```
//Inside the child thread of a process
//Check the lock for availability & acquire the lock if available.
//The lock can be set by any other threads
while (_InterlockedCompareExchange(&bFlag, 1, 0) == 1);
012013EE   mov        ecx,1
012013F3   mov        edx,1208130h
012013F8   xor        eax,eax
012013FA   lock cmpxchg dword ptr [edx],ecx
012013FE   cmp        eax,1
01201401   jne        child_thread+35h (01201405h)
01201403   jmp        child_thread+1Eh (012013EEh)
//Rest of the source code dealing with shared resource access
//................................................................
```

The Intel 486 and above family of processors provide hardware level support for atomic execution of increment and decrement operations also. The *XADD* low level instruction implements atomic execution of increment and decrement operations. Windows Embedded Compact/NT kernel makes these features available to the users through a set of *Interlocked* function API calls. The API call *InterlockedIncrement (LPLONG lpAddend)* increments the value of the variable pointed by *lpAddend* and the API *InterlockedDecrement (LPLONG lpAddend)* decrements the value of the variable pointed by *lpAddend*.

The lock based mutual exclusion implementation always checks the state of a lock and waits till the lock is available. This keeps the processes/threads always busy and forces the processes/threads to wait for the availability of the lock for proceeding further. Hence this synchronisation mechanism is popularly known as '*Busy waiting*'. The '*Busy waiting*' technique can also be visualised as a lock around which the process/thread spins, checking for its availability. Spin locks are useful in handling scenarios where the processes/threads are likely to be blocked for a shorter period of time on waiting the lock, as they avoid OS overheads on context saving and process re-scheduling. Another drawback of Spin lock based synchronisation is that if the lock is being held for a long time by a process and if it is preempted by the OS, the other threads waiting for this lock may have to spin a longer time for getting it. The '*Busy waiting*' mechanism keeps the process/threads always active, performing a task which is not useful and leads to the wastage of processor time and high power consumption.

The interlocked operations are the most efficient synchronisation primitives when compared to the classic lock based synchronisation mechanism. Interlocked function based synchronisation technique brings the following value adds.

- The interlocked operation is free from waiting. Unlike the mutex, semaphore and critical section synchronisation objects which may require waiting on the object, if they are not available at the time of request, the interlocked function simply performs the operation and returns immediately. This avoids the blocking of the thread which calls the interlocked function.
- The interlocked function call is directly converted to a processor specific instruction and there is no user mode to kernel mode transition as in the case of mutex, semaphore and critical section objects. This avoids the user mode to kernel mode transition delay and thereby increases the overall performance.

The types of interlocked operations supported by an OS are underlying processor hardware dependent and so they are limited in functionality. Normally the bit manipulation (Boolean) operations are not supported by interlocked functions. Also the interlocked operations are limited to integer or pointer variables only. This limits the possibility of extending the interlocked functions to variables of other types. Under windows operating systems, each process has its own virtual address space and so the interlocked functions can only be used for synchronising the access to a variable that is shared by multiple threads of a process (Multiple threads of a process share the same address space) (Intra Process Synchronisation). The interlocked functions can be extended for synchronising the access of the variables shared across multiple processes if the variable is kept in shared memory.

10.8.2.2 Mutual Exclusion through Sleep & Wakeup

The '*Busy waiting*' mutual exclusion enforcement mechanism used by processes makes the CPU always busy by checking the lock to see whether they can proceed. This results in the wastage of CPU time and leads to high power consumption. This is not affordable in embedded systems powered on battery, since it affects the battery backup time of the device. An alternative to '*busy waiting*' is the '*Sleep & Wakeup*' mechanism. When a process is not allowed to access the critical section, which is currently being locked by another

process, the process undergoes '*Sleep*' and enters the '*blocked*' state. The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section. The process which owns the critical section sends a wakeup message to the process, which is sleeping as a result of waiting for the access to the critical section, when the process leaves the critical section. The '*Sleep & Wakeup*' policy for mutual exclusion can be implemented in different ways. Implementation of this policy is OS kernel dependent. The following section describes the important techniques for '*Sleep & Wakeup*' policy implementation for mutual exclusion by Windows NT/CE OS kernels.

Semaphore Semaphore is a sleep and wakeup based mutual exclusion implementation for shared resource access. Semaphore is a system resource and the process which wants to access the shared resource can first acquire this system object to indicate the other processes which wants the shared resource that the shared resource is currently acquired by it. The resources which are shared among a process can be either for exclusive use by a process or for using by a number of processes at a time. The display device of an embedded system is a typical example for the shared resource which needs exclusive access by a process. The Hard disk (secondary storage) of a system is a typical example for sharing the resource among a limited number of multiple processes. Various processes can access the different sectors of the hard-disk concurrently. Based on the implementation of the sharing limitation of the shared resource, semaphores are classified into two; namely '*Binary Semaphore*' and '*Counting Semaphore*'. The binary semaphore provides exclusive access to shared resource by allocating the resource to a single process at a time and not allowing the other processes to access it when it is being owned by a process. The implementation of binary semaphore is OS kernel dependent. Under certain OS kernel it is referred as *mutex*. Unlike a binary semaphore, the '*Counting Semaphore*' limits the access of resources by a fixed number of processes/threads. '*Counting Semaphore*' maintains a count between zero and a maximum value. It limits the usage of the resource to the maximum value of the count supported by it. The state of the counting semaphore object is set to 'signalled' when the count of the object is greater than zero. The count associated with a '*Semaphore object*' is decremented by one when a process/thread acquires it and the count is incremented by one when a process/thread releases the '*Semaphore object*'. The state of the '*Semaphore object*' is set to non-signalled when the semaphore is acquired by the maximum number of processes/threads that the semaphore can support (i.e. when the count associated with the '*Semaphore object*' becomes zero). A real world example for the counting semaphore concept is the dormitory system for accommodation (Fig. 10.34). A dormitory contains a fixed number of beds (say 5) and at any point of time it can be shared by the maximum number of users supported by the dormitory. If a person wants to avail the dormitory facility, he/she can contact the dormitory caretaker for checking the availability. If beds are available in the dorm the caretaker will hand over the keys to the user. If beds are not available currently, the user can register his/her name to get notifications when a slot is available. Those who are availing the dormitory shares the dorm facilities like TV, telephone, toilet, etc. When a dorm user vacates, he/she gives the keys back to the caretaker. The caretaker informs the users, who booked in advance, about the dorm availability.

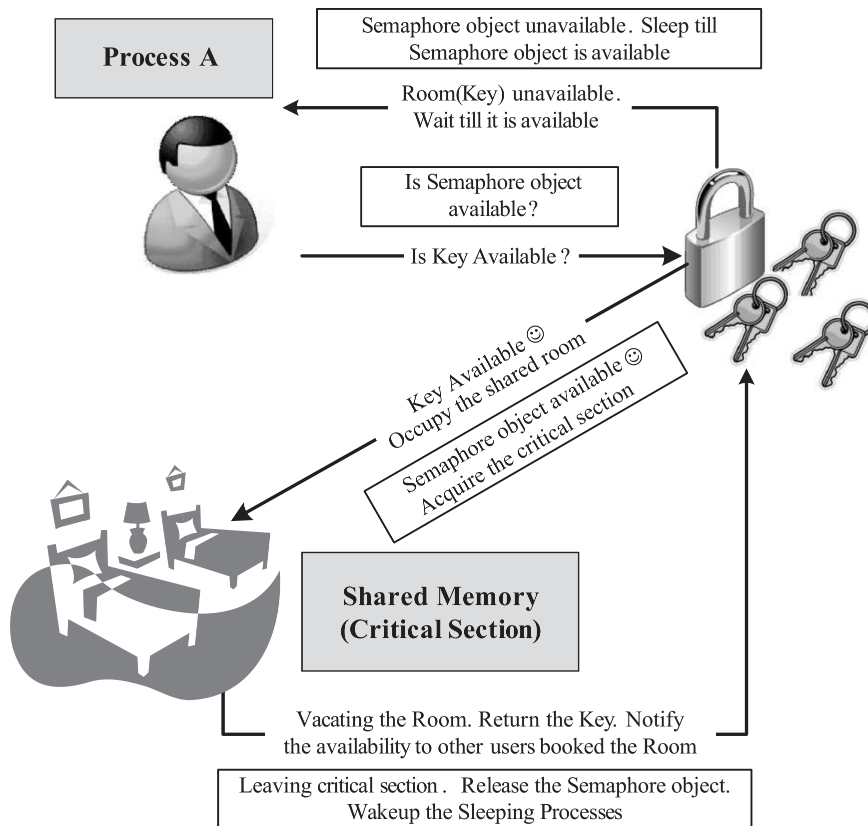

Fig. 10.34 The Concept of Counting Semaphore

The creation and usage of '*counting semaphore object*' is OS kernel dependent. Let us have a look at how we can implement semaphore based synchronisation for the '*Racing*' problem we discussed in the beginning, under the Windows kernel. The following code snippet explains the same.

```
##include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#define MAX_SEMAPHORE_COUNT 1   //Make the semaphore object for exclusive use
#define thread_count 2          //No.of Child Threads
//*******************************************************************
//counter is an integer variable and Buffer is a byte array shared //between
two threads Process_A and Process_B
char Buffer[10] = { 1,2,3,4,5,6,7,8,9,10 };
short int counter = 0;
//Define the handle to Semaphore object
HANDLE hSemaphore;
//*******************************************************************
// Child Thread 1
void Process_A(void) {
```

```
 int i;
 for (i = 0; i<5; i++)
 {
  if (Buffer[i] > 0)
  {
   //Wait for the signaling of Semaphore object
   WaitForSingleObject(hSemaphore, INFINITE);
   //Semaphore is acquired
   counter++;
   printf("Process A : Counter = %d\n", counter);
   //Release the Semaphore Object
   if (!ReleaseSemaphore(
                    hSemaphore, // handle to semaphore
                    1, // increase count by one
                    NULL)) // not interested in previous count
   {
    //Semaphore Release failed. Print Error code & return.
    printf("Release Semaphore Failed with Error Code : %d\n",
    GetLastError());
    return;
   }
  }
 }
 return;
}
//*******************************************************************
// Child Thread 2
void Process_B(void) {
 int j;
 for (j = 5; j<10; j++)
 {
  if (Buffer[j] > 0)
  {
   //Wait for the signalling of Semaphore object
   WaitForSingleObject(hSemaphore, INFINITE);
   //Semaphore is acquired
   counter++;
   printf("Process B : Counter = %d\n", counter);
   //Release Semaphore
   if (!ReleaseSemaphore(
                    hSemaphore, // handle to semaphore
                    1, // increase count by one
                    NULL)) // not interested in previous count
   {
    //Semaphore Release failed. Print Error code &
    //return.
    printf("Release Semaphore Failed Error Code : %d\n", GetLastError());
    return;
```

```
    }
   }
  }
  return;
 }
//*******************************************************************
// Main Thread
void main() {
 //Define HANDLE for child threads
 HANDLE child_threads[thread_count];
 DWORD thread_id;
 int i;
 //Create Semaphore object
 hSemaphore = CreateSemaphore(
              NULL, // default security attributes
              MAX_SEMAPHORE_COUNT, // initial count. Create as signaled
              MAX_SEMAPHORE_COUNT, // maximum count
             TEXT("Semaphore")); // Semaphore object with name "Semaphore"
 if (NULL == hSemaphore)
 {
  printf("Semaphore Object Creation Failed : Error Code : %d",
  GetLastError());
  //Semaphore Object Creation failed. Return
  return;
 }
 //Create Child thread 1
 child_threads[0] = CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Process_A,
                                 (LPVOID)0, 0, &thread_id);
 //Create Child thread 2
 child_threads[1] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_B,
                                 (LPVOID)0, 0, &thread_id);
 //Check the success of creation of child threads
 for (i = 0; i<thread_count; i++)
 {
  if (NULL == child_threads[i])
  {
   //Child thread creation failed.
   printf("Child thread Creation failed with Error Code : %d",
   GetLastError());
   return;
  }
 }
 // Wait for the termination of child threads
 WaitForMultipleObjects(thread_count, child_threads, TRUE, INFINITE);
 //Close handles of child threads
 for (i = 0; i < thread_count; i++)
  CloseHandle(child_threads[i]);
 //Close Semaphore object handle
```

```
 CloseHandle(hSemaphore);
 return;
}
```

Please refer to the Online Learning Centre for details on the various Win32 APIs used in the program for counting semaphore creation, acquiring, signalling, and releasing. The VxWorks and MicroC/OS-II Real-Time kernels also implements the Counting semaphore based task synchronisation/shared resource access. We will discuss them in detail in a later chapter.

Counting Semaphores are similar to *Binary Semaphores* in operation. The only difference between *Counting Semaphore* and *Binary Semaphore* is that *Binary Semaphore* can only be used for exclusive access, whereas *Counting Semaphores* can be used for both exclusive access (by restricting the maximum count value associated with the semaphore object to one (1) at the time of creation of the semaphore object) and limited access (by restricting the maximum count value associated with the semaphore object to the limited number at the time of creation of the semaphore object).

Binary Semaphore (Mutex) Binary Semaphore (Mutex) is a synchronisation object provided by OS for process/thread synchronisation. Any process/thread can create a '*mutex object*' and other processes/threads of the system can use this '*mutex object*' for synchronising the access to critical sections. Only one process/thread can own the '*mutex object*' at a time. The state of a mutex object is set to signalled when it is not owned by any process/thread, and set to non-signalled when it is owned by any process/thread. A real world example for the mutex concept is the hotel accommodation system (lodging system) Fig. 10.35. The rooms in a hotel are shared for the public. Any user who pays and follows the norms of the hotel can avail the

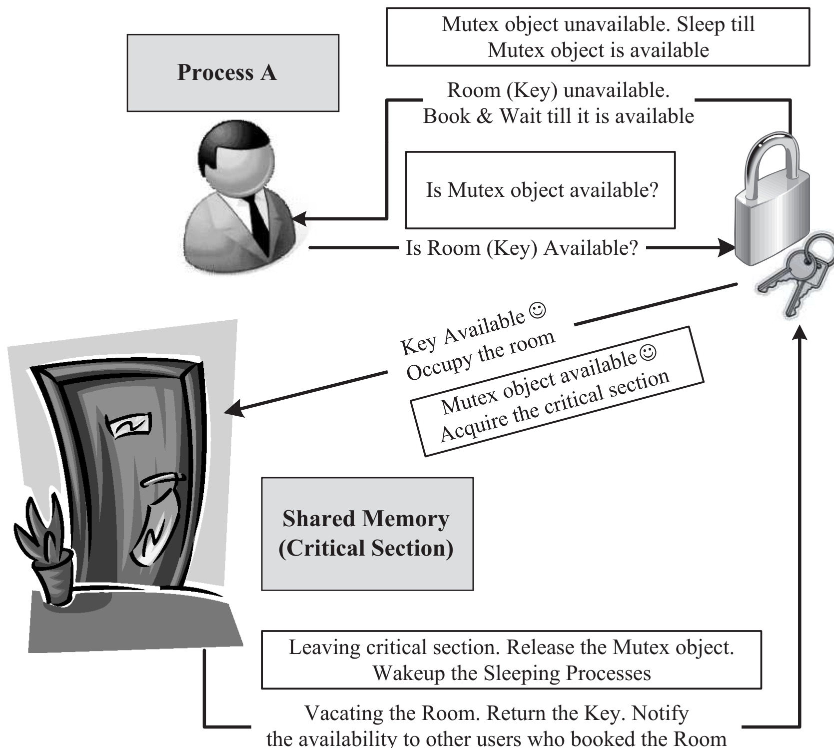

Fig. 10.35 The Concept of Binary Semaphore (Mutex)

rooms for accommodation. A person wants to avail the hotel room facility can contact the hotel reception for checking the room availability (see Fig. 10.35). If room is available the receptionist will handover the room key to the user. If room is not available currently, the user can book the room to get notifications when a room is available. When a person gets a room he/she is granted the exclusive access to the room facilities like TV, telephone, toilet, etc. When a user vacates the room, he/she gives the keys back to the receptionist. The receptionist informs the users, who booked in advance, about the room's availability.

Let's see how we can implement mutual exclusion with mutex object in the '*Racing*' problem example given under the section '*Racing*', under Windows kernel.

```
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#define thread_count 2 //No.of Child Threads
//*******************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two
//threads Process_A and Process_B
char Buffer[10] = { 1,2,3,4,5,6,7,8,9,10 };
short int counter = 0;
//Define the handle to Mutex Object
HANDLE hMutex;
//*******************************************************************
// Child Thread 1
void Process_A(void) {
 int i;
 for (i = 0; i<5; i++)
 {
  if (Buffer[i] > 0)
  {
   //Wait for signaling of the Mutex object
   WaitForSingleObject(hMutex, INFINITE);
   //Mutex is acquired
   counter++;
   printf("Process A : Counter = %d\n", counter);
   //Release the Mutex Object
   if (!ReleaseMutex(hMutex)) // handle to Mutex Object
   {
    //Mutex object Releasing failed. Print Error code & return.
    printf("Release Mutex Failed with Error Code : %d\n", GetLastError());
    return;
   }
  }
 }
 return;
}
//*******************************************************************
// Child Thread 2
void Process_B(void) {
```

```
 int j;
 for (j = 5; j<10; j++)
 {
  if (Buffer[j] > 0)
  {
   //Wait for signaling of the Mutex object
   WaitForSingleObject(hMutex, INFINITE);
   //Mutex object is acquired
   counter++;
   printf("Process B : Counter = %d\n", counter);
   //Release Mutex object
   if (!ReleaseMutex(hMutex)) // handle to Mutex Object
   {
    //Mutex object Release failed. Print Error code & return.
    printf("Release Mutex Failed with Error Code : %d\n", GetLastError());
    return;
   }
  }
 }
 return;
}
//*******************************************************************
// Main Thread
void main() {
 //Define HANDLE for child threads
 HANDLE child_threads[thread_count];
 DWORD thread_id;
 int i;
 //Create Mutex object
 hMutex = CreateMutex(
                 NULL, // default security attributes
                 FALSE, // Not initial ownership
                 TEXT("Mutex")); // Mutex object with name "Mutex"
 if (NULL == hMutex)
 {
  printf("Mutex Object Creation Failed : Error Code : %d",GetLastError());
  //Mutex Object Creation failed. Return
  return;
 }
 //Create Child thread 1
 child_threads[0] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_A,
                               (LPVOID)0, 0, &thread_id);
 //Create Child thread 2
 child_threads[1] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_B,
                               (LPVOID)0, 0, &thread_id);
 //Check the success of creation of child threads
 for (i = 0; i<thread_count; i++)
 {
```

```
   if (NULL == child_threads[i])
   {
    //Child thread creation failed.
   printf("Child thread Creation failed with Error Code : %d", GetLastError());
    return;
   }
  }
  // Wait for the termination of child threads
  WaitForMultipleObjects(thread_count, child_threads, TRUE, INFINITE);
  //Close child thread handles
  for (i = 0; i < thread_count; i++)
  {
   CloseHandle(child_threads[i]);
  }
  //Close Mutex object handle
  CloseHandle(hMutex);
  return;
}
```

Please refer to the Online Learning Centre for details on the various Win32 APIs used in the program for mutex creation, acquiring, signalling, and releasing.

The mutual exclusion semaphore is a special implementation of the binary semaphore by certain real-time operating systems like VxWorks and MicroC/OS-II to prevent priority inversion problems in shared resource access. The mutual exclusion semaphore has an option to set the priority of a task owning it to the highest priority of the task which is being pended while attempting to acquire the semaphore which is already in use by a low priority task. This ensures that the low priority task which is currently holding the semaphore, when a high priority task is waiting for it, is not pre-empted by a medium priority task. This is the mechanism supported by the mutual exclusion semaphore to prevent priority inversion.

VxWorks kernel also supports binary semaphores for synchronising shared resource access. We will discuss about it in detail in a later chapter.

Critical Section Objects In Windows Embedded Compact, the '*Critical Section object*' is same as the '*mutex object*' except that '*Critical Section object*' can only be used by the threads of a single process (Intra process). The piece of code which needs to be made as '*Critical Section*' is placed at the '*Critical Section*' area by the process. The memory area which is to be used as the '*Critical Section*' is allocated by the process. The process creates a '*Critical Section*' area by creating a variable of type *CRITICAL_SECTION*. The *Critical Section*' must be initialised before the threads of a process can use it for getting exclusive access. The *InitialiseCritica lSection(LPCRITICAL_SECTION lpCriticalSection)* API initialises the critical section pointed by the pointer *lpCriticalSection* to the critical section. Once the critical section is initialised, all threads in the process can use it. Threads can use the API call *EnterCriticalSection (LPCRITICAL_SECTION lpCriticalSection)* for getting the exclusive ownership of the critical section pointed by the pointer *lpCriticalSection*. Calling the *EnterCriticalSection()* API blocks the execution of the caller thread if the critical section is already in use by other threads and the thread waits for the critical section object. Threads which are blocked by the *EnterCriticalSection()* call, waiting on a critical section are added to a wait queue and are woken when the critical section is available to the requested thread. The API call *TryEnterCriticalSection(LPCRITICA L_SECTION lpCriticalSection)* attempts to enter the critical section pointed by the pointer *lpCriticalSection* without blocking the caller thread. If the critical section is not in use by any other thread, the calling thread gets the ownership of the critical section. If the critical section is already in use by another thread, the

TryEnterCriticalSection() call indicates it to the caller thread by a specific return value and the thread resumes its execution. A thread can release the exclusive ownership of a critical section by calling the API *LeaveCriticalSection(LPCR-ITICAL_SECTION lpCriticalSection)*. The threads of a process can use the API *DeleteCriticalSection (LPCRITICAL_SECTION lpCriticalSection)* to release all resources used by a critical section object which was created by the process with the *CRITICAL_SECTION* variable.

Now let's have a look at the '*Racing*' problem we discussed under the section '*Racing*'. The racing condition can be eliminated by using a critical section object for synchronisation. The following code snippet illustrates the same.

```
##include "stdafx.h"
#include <stdio.h>
#include <windows.h>
//****************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two threads
char Buffer[10] = { 1,2,3,4,5,6,7,8,9,10 };
short int counter = 0;
//Define the critical section
CRITICAL_SECTION CS;
//****************************************************************
// Child Thread 1
void Process_A(void) {
 int i;
 for (i = 0; i<5; i++)
 {
  if (Buffer[i] > 0)
  {
   //Use critical section object for synchronisation
   EnterCriticalSection(&CS);
   counter++;
   LeaveCriticalSection(&CS);
  }
  printf("Process A : Counter = %d\n", counter);
 }
}
//****************************************************************
// Child Thread 2
void Process_B(void) {
 int j;
 for (j = 5; j<10; j++)
 {
  if (Buffer[j] > 0)
  {
   //Use critical section object for synchronisation
   EnterCriticalSection(&CS);
   counter++;
   LeaveCriticalSection(&CS);
  }
```

```
  printf("Process B : Counter = %d\n", counter);
 }
}
//*******************************************************************
// Main Thread
int main() {
 DWORD id;
 //Initialise critical section object
 InitialiseCriticalSection(&CS);
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_A, (LPVOID)0, 0, &id);
 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_B, (LPVOID)0, 0, &id);
 Sleep(100000);
 return 0;
}
```

Here the shared resource is the shared variable '*counter*'. The concurrent access to this variable by the threads '*Process_A*' and '*Process_B*' may create race condition and may produce incorrect results. The critical section object '*CS*' holds the piece of code corresponding to the access of the shared variable '*counter*' by each threads. This ensures that the memory area containing the low level instructions corresponding to the high level instruction '*counter++*' is accessed exclusively by threads '*Process_A*' and '*Process_B*' and avoids a race condition. The output of the above piece of code when executed on an Intel Centrino Duo processor running Windows XP OS is given in Fig. 10.36.

```
"C:\Program Files\Microsoft Visual Studio\MyProjects\
Process A : Counter = 1
Process A : Counter = 1
Process B : Counter = 2
Process A : CouPProcess B : Counter = 4
nProcess B : Counter = 4
ter = 3
Process B : Counter = 5
Process A : Counter = 6
Process B : Counter = 7
8
8
Process B : Counter = 9
Process A : Counter = 10
```

Fig. 10.36 Output of the Win32 application resolving racing condition through critical section object

The final value of '*counter*' is obtained as 10, which is the expected result for this piece of code. If you observe this output window you can see that the text is not outputted to the o/p window in the expected manner. The *printf ()* library routine used in this sample code is re-entrant and it can be preempted while in execution. That is why the outputting of text happened in a non expected way.

Note: It should be noted that the scheduling of the threads '*Process_A*' and '*Process_B*' is OS kernel scheduling policy dependent and you may not get the same output all the time when you run this piece of code under **Windows XP**.

The critical section object makes the piece of code residing inside it non-reentrant. Now let's try the above piece of code by putting the *printf ()* library routine in the critical section object.

```
##include "stdafx.h"
#include <stdio.h>
#include <windows.h>
//*******************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two threads
char Buffer[10] = { 1,2,3,4,5,6,7,8,9,10 };
short int counter = 0;
//Define the critical section
CRITICAL_SECTION CS;
//*******************************************************************
// Child Thread 1
void Process_A(void) {
 int i;
 for (i = 0; i<5; i++)
 {
  if (Buffer[i] > 0)
  {
   //Use critical section object for synchronisation
   EnterCriticalSection(&CS);
   counter++;
   printf("Process A : Counter = %d\n", counter);
   LeaveCriticalSection(&CS);
  }
 }
}
//*******************************************************************
// Child Thread 2
void Process_B(void) {
 int j;
 for (j = 5; j<10; j++)
 {
  if (Buffer[j] > 0)
  {
   //Use critical section object for synchronisation
   EnterCriticalSection(&CS);
   counter++;
   printf("Process B : Counter = %d\n", counter);
   LeaveCriticalSection(&CS);
  }
 }
```

```
}
//*******************************************************************
// Main Thread
int main() {
 DWORD id;
 //Initialise critical section object
 InitialiseCriticalSection(&CS);
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_A, (LPVOID)0, 0, &id);
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Process_B, (LPVOID)0, 0, &id);
 Sleep(100000);
 return 0;
}
```

The output of the above piece of code when executed on a Windows 10 machine is given below.

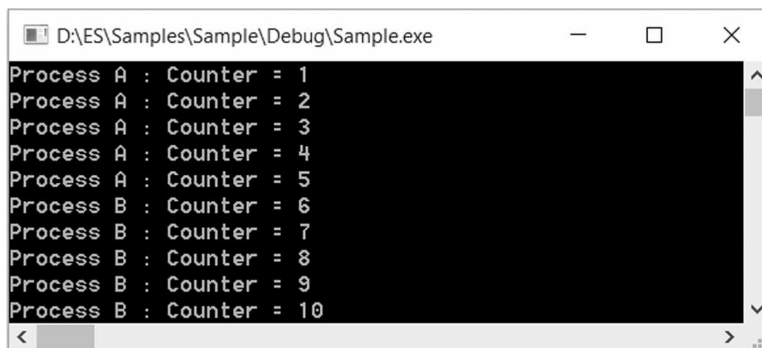

Fig. 10.37 Output of the Win32 application resolving racing condition through critical section object

> **Note:** It should be noted that the scheduling of the threads '*Process_A*' and '*Process_B*' is OS kernel scheduling policy dependent and you may not get the same output all the time when you run this piece of code in Windows 10. The output of the above program when executed at three different instances of time is given shown in Fig. 10.38.

Events Event object is a synchronisation technique which uses the notification mechanism for synchronisation. In concurrent execution we may come across situations which demand the processes to wait for a particular sequence for its operations. A typical example of this is the producer consumer threads, where the consumer thread should wait for the consumer thread to produce the data and producer thread should wait for the consumer thread to consume the data before producing fresh data. If this sequence is not followed it will end up in producer-consumer problem. Notification mechanism is used for handling this scenario. Event objects are used for implementing notification mechanisms. A thread/process can wait for an event and another thread/process can set this event for processing by the waiting thread/process. The creation and handling of event objects for notification is OS kernel dependent. Please refer to the Online Learning Centre for information on the usage of 'Events' under Windows Kernel for process/thread synchronisation.

The MicroC/OS-II kernel also uses 'events' for task synchronisation. We will discuss it in a later chapter.

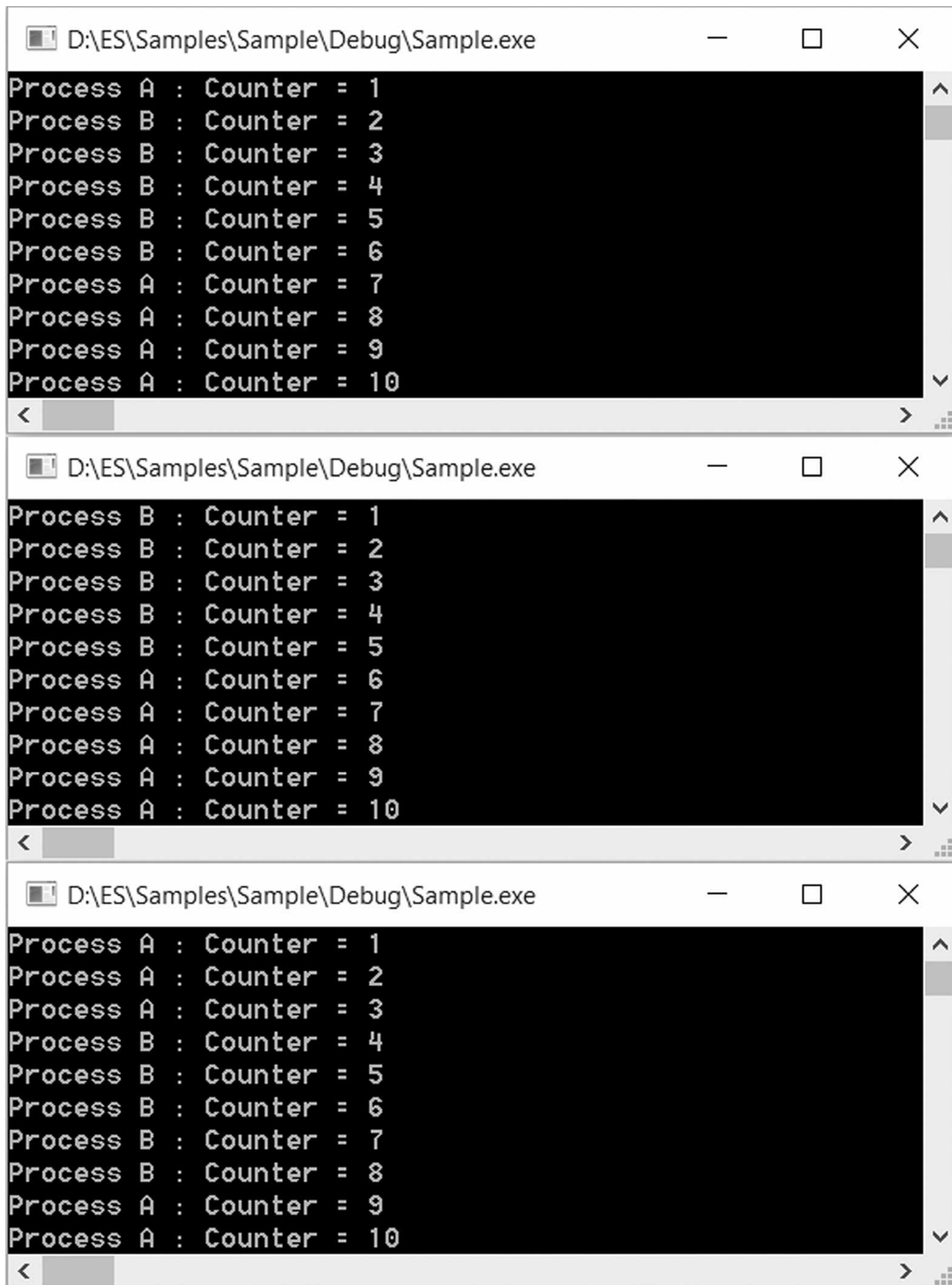

Fig. 10.38 Illustration of scheduler behaviour under Windows NT (E.g. Windows 10) kernel

10.9 DEVICE DRIVERS

LO 9 Analyse device drivers, their role in an operating system based embedded system design, the structure of a device driver, and interrupt handling inside device drivers

Device driver is a piece of software that acts as a bridge between the operating system and the hardware. In an operating system based product architecture, the user applications talk to the Operating System kernel for all necessary information exchange including communication with the hardware peripherals. The architecture of the OS kernel will not allow direct device access from the user application. All the device related access should flow through the OS kernel and the OS kernel routes it to the concerned hardware peripheral. OS provides interfaces in the form of Application Programming Interfaces (APIs) for accessing the hardware. The device driver abstracts the hardware from user applications. The topology of user applications and hardware interaction in an RTOS based system is depicted in Fig. 10.39.

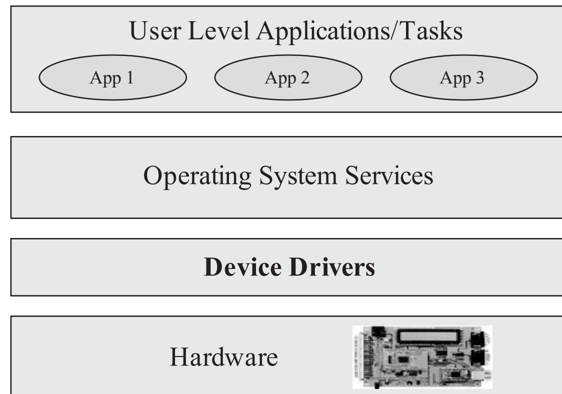

Fig. 10.39 Role of Device driver in Embedded OS based products

Device drivers are responsible for initiating and managing the communication with the hardware peripherals. They are responsible for establishing the connectivity, initialising the hardware (setting up various registers of the hardware device) and transferring data. An embedded product may contain different types of hardware components like Wi-Fi module, File systems, Storage device interface, etc. The initialisation of these devices and the protocols required for communicating with these devices may be different. All these requirements are implemented in drivers and a single driver will not be able to satisfy all these. Hence each hardware (more specifically each class of hardware) requires a unique driver component.

Certain drivers come as part of the OS kernel and certain drivers need to be installed on the fly. For example, the program storage memory for an embedded product, say NAND Flash memory requires a NAND Flash driver to read and write data from/to it. This driver should come as part of the OS kernel image. Certainly the OS will not contain the drivers for all devices and peripherals under the Sun. It contains only the necessary drivers to communicate with the onboard devices (Hardware devices which are part of the platform) and for certain set of devices supporting standard protocols and device class (Say USB Mass storage device or HID devices like Mouse/keyboard). If an external device, whose driver software is not available with OS kernel image, is connected to the embedded device (Say a medical device with custom USB class implementation is connected to the USB port of the embedded product), the OS prompts the user to instal its driver manually. Device drivers which are part of the OS image are known as 'Built-in drivers' or 'On-board drivers'. These drivers are loaded by the OS at the time of booting the device and are always kept in the RAM. Drivers which need to be installed for accessing a device are known as 'Installable drivers'. These drivers are loaded by the OS on a need basis. Whenever the device is connected, the OS loads the corresponding driver to memory. When the device is removed, the driver is unloaded from memory. The Operating system maintains a record of the drivers corresponding to each hardware.

The implementation of driver is OS dependent. There is no universal implementation for a driver. How the driver communicates with the kernel is dependent on the OS structure and implementation. Different Operating Systems follow different implementations.

It is very essential to know the hardware interfacing details like the memory address assigned to the device, the Interrupt used, etc. of on-board peripherals for writing a driver for that peripheral. It varies on the hardware design of the product. Some Real-Time operating systems like 'Windows CE' support a layered architecture for the driver which separates out the low level implementation from the OS specific interface. The low level implementation part is generally known as Platform Dependent Device (PDD) layer. The OS specific interface part is known as Model Device Driver (MDD) or Logical Device Driver (LDD). For a standard driver, for a specific operating system, the MDD/LDD always remains the same and only the PDD part needs to be modified according to the target hardware for a particular class of devices.

Most of the time, the hardware developer provides the implementation for all on board devices for a specific OS along with the platform. The drivers are normally shipped in the form of *Board Support Package*. The *Board Support Package* contains low level driver implementations for the onboard peripherals and OEM Adaptation Layer (OAL) for accessing the various chip level functionalities and a bootloader for loading the operating system. The OAL facilitates communication between the Operating System (OS) and the target device and includes code to handle interrupts, timers, power management, bus abstraction, generic I/O control codes (IOCTLs), etc. The driver files are usually in the form of a dll file. Drivers can run on either user space or kernel space. Drivers which run in user space are known as *user mode drivers* and the drivers which run in kernel space are known as *kernel mode drivers*. User mode drivers are safer than kernel mode drivers. If an error or exception occurs in a user mode driver, it won't affect the services of the kernel. On the other hand, if an exception occurs in the kernel mode driver, it may lead to the kernel crash. The way how a device driver is written and how the interrupts are handled in it are operating system and target hardware specific. However regardless of the OS types, a device driver implements the following:

1. Device (Hardware) Initialisation and Interrupt configuration
2. Interrupt handling and processing
3. Client interfacing (Interfacing with user applications)

The Device (Hardware) initialisation part of the driver deals with configuring the different registers of the device (target hardware). For example configuring the I/O port line of the processor as Input or output line and setting its associated registers for building a General Purpose IO (GPIO) driver. The interrupt configuration part deals with configuring the interrupts that needs to be associated with the hardware. In the case of the GPIO driver, if the intention is to generate an interrupt when the Input line is asserted, we need to configure the interrupt associated with the I/O port by modifying its associated registers. The basic Interrupt configuration involves the following.

1. Set the interrupt type (Edge Triggered (Rising/Falling) or Level Triggered (Low or High)), enable the interrupts and set the interrupt priorities.
2. Bind the Interrupt with an Interrupt Request (IRQ). The processor identifies an interrupt through IRQ. These IRQs are generated by the Interrupt Controller. In order to identify an interrupt the interrupt needs to be bonded to an IRQ.
3. Register an Interrupt Service Routine (ISR) with an Interrupt Request (IRQ). ISR is the handler for an Interrupt. In order to service an interrupt, an ISR should be associated with an IRQ. Registering an ISR with an IRQ takes care of it.

With these the interrupt configuration is complete. If an interrupt occurs, depending on its priority, it is serviced and the corresponding ISR is invoked. The processing part of an interrupt is handled in an ISR. The whole interrupt processing can be done by the ISR itself or by invoking an Interrupt Service Thread (IST). The IST performs interrupt processing on behalf of the ISR. To make the ISR compact and short, it is always advised to use an IST for interrupt processing. The intention of an interrupt is to send or receive command or data to and from the hardware device and make the received data available to user programs for application specific processing. Since interrupt processing happens at kernel level, user applications may not have direct access to the drivers to pass and receive data. Hence it is the responsibility of the Interrupt

processing routine or thread to inform the user applications that an interrupt is occurred and data is available for further processing. The client interfacing part of the device driver takes care of this. The client interfacing implementation makes use of the Inter Process communication mechanisms supported by the embedded OS for communicating and synchronising with user applications and drivers. For example, to inform a user application that an interrupt is occurred and the data received from the device is placed in a shared buffer, the client interfacing code can signal (or set) an event. The user application creates the event, registers it and waits for the driver to signal it. The driver can share the received data through shared memory techniques. IOCTLs, shared buffers, etc. can be used for data sharing. The story line is incomplete without performing an interrupt done (Interrupt processing completed) functionality in the driver. Whenever an interrupt is asserted, while vectoring to its corresponding ISR, all interrupts of equal and low priorities are disabled. They are re-enable only on executing the interrupt done function (Same as the Return from Interrupt RETI instruction execution for *8051*) by the driver. The interrupt done function can be invoked at the end of corresponding ISR or IST.

We will discuss more about device driver development in a dedicated book coming under this book series.

10.10 HOW TO CHOOSE AN RTOS

LO 10 Discuss the different functional and non-functional requirements that need to be addressed in the selection of a Real-Time Operating System

The decision of choosing an RTOS for an embedded design is very crucial. A lot of factors needs to be analysed carefully before making a decision on the selection of an RTOS. These factors can be either functional or non-functional. The following section gives a brief introduction to the important functional and non-functional requirements that needs to be analysed in the selection of an RTOS for an embedded design.

10.10.1 Functional Requirements

Processor Support It is not necessary that all RTOS's support all kinds of processor architecture. It is essential to ensure the processor support by the RTOS.

Memory Requirements The OS requires ROM memory for holding the OS files and it is normally stored in a non-volatile memory like FLASH. OS also requires working memory RAM for loading the OS services. Since embedded systems are memory constrained, it is essential to evaluate the minimal ROM and RAM requirements for the OS under consideration.

Real-time Capabilities It is not mandatory that the operating system for all embedded systems need to be Real-time and all embedded Operating systems are 'Real-time' in behaviour. The task/process scheduling policies plays an important role in the 'Real-time' behaviour of an OS. Analyse the real-time capabilities of the OS under consideration and the standards met by the operating system for real-time capabilities.

Kernel and Interrupt Latency The kernel of the OS may disable interrupts while executing certain services and it may lead to interrupt latency. For an embedded system whose response requirements are high, this latency should be minimal.

Inter Process Communication and Task Synchronisation The implementation of Inter Process Communication and Synchronisation is OS kernel dependent. Certain kernels may provide a bunch of options whereas others provide very limited options. Certain kernels implement policies for avoiding priority inversion issues in resource sharing.

Modularisation Support Most of the operating systems provide a bunch of features. At times it may not be necessary for an embedded product for its functioning. It is very useful if the OS supports modularisation where in which the developer can choose the essential modules and re-compile the OS image for functioning. Windows CE is an example for a highly modular operating system.

Support for Networking and Communication The OS kernel may provide stack implementation and driver support for a bunch of communication interfaces and networking. Ensure that the OS under consideration provides support for all the interfaces required by the embedded product.

Development Language Support Certain operating systems include the run time libraries required for running applications written in languages like Java and C#. A Java Virtual Machine (JVM) customised for the Operating System is essential for running java applications. Similarly the .NET Compact Framework (.NETCF) is required for running Microsoft® .NET applications on top of the Operating System. The OS may include these components as built-in component, if not, check the availability of the same from a third party vendor for the OS under consideration.

10.10.2 Non-functional Requirements

Custom Developed or Off the Shelf Depending on the OS requirement, it is possible to go for the complete development of an operating system suiting the embedded system needs or use an off the shelf, readily available operating system, which is either a commercial product or an Open Source product, which is in close match with the system requirements. Sometimes it may be possible to build the required features by customising an Open source OS. The decision on which to select is purely dependent on the development cost, licensing fees for the OS, development time and availability of skilled resources.

Cost The total cost for developing or buying the OS and maintaining it in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.

Development and Debugging Tools Availability The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design. Certain Operating Systems may be superior in performance, but the availability of tools for supporting the development may be limited. Explore the different tools available for the OS under consideration.

Ease of Use How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.

After Sales For a commercial embedded RTOS, after sales in the form of e-mail, on-call services, etc. for bug fixes, critical patch updates and support for production issues, etc. should be analysed thoroughly.

Summary

- ✓ The *Operating System* is responsible for making the system convenient to use, organise and manage system resources efficiently and properly. **LO1**
- ✓ Process/Task management, Primary memory management, File system management, I/O system (Device) management, Secondary Storage Management, protection implementation, Time management, Interrupt handling, etc. are the important services handled by the OS kernel. **LO1**