Module 6: Resource Sharing and Management

The main motivation for scheduling various OS services is to maximize the usage of CPU resource, memory, and other IO resources. Consider the usage of a printer as an output resource. A user takes printouts only once in a while. A printer usage, therefore, can be shared amongst many users. The motivation to share a resource may come from several reasons. Sharing enhances utilization of resources immensely.

Sharing a resource is imperative in cases where we have a very expensive and specialized resource. For instance, an image processing resource, connected to a computer system, is a special resource. Such a resource is used in short periods of time, i.e. it is sparingly used. Similarly, in the context of a large project, there may be a file or a data-base which is shared amongst many users. Such a shared file may need to be updated from several sources. The shared file is then a shared resource. In this case, the sequencing of updates may be very critical for preserving data integrity and consistency. It may affect temporal semantics of the shared data. This is particularly true in transaction processing systems. In this chapter we shall study how the resources may be scheduled for shared usage. In particular, we shall study two very important concepts relating to mutual exclusion and deadlocks.

6.1 Need for Scheduling

Resources may be categorized depending upon the nature of their use. To enforce temporal sharing of a common resource, the OS needs a policy to schedule its usage. The policy may depend upon the nature of resource, frequency of its use and the context of its usage. In the case of a printer, the OS can spool printout requests. Printing, additionally, requires that once a process is engaged in printing, it must have its exclusive usage till it finishes the current print job. If that is not the case then the printouts from the printer shall be garbled. Some specialized resources, like a flat-bed plotter, require an elaborate initial set-up. So once assigned to a process, its usage better not be pre-empted. A process that gets such a resource should be permitted to keep it till either the process terminates or releases the resource. This is also true of a transaction which updates a shared data record. The transaction should complete the record's update before another process is given the access to the record.

Processes may need more than one resource. It is quite possible that a process may not be able to progress till it gets all the resources it needs. Let us suppose that a process P_1 needs resources r_1 and r_2. Process P_2 needs resources r_2 and r_3. Process P_1 can proceed only when it has both r_1 and r_2. If process P_2 has been granted r_2 then process P_1 has to wait till process P_2 terminates or releases r_2. Clearly, the resource allocation policy of an OS can affect the overall throughput of a system.

6.2 Mutual Exclusion

The mutual exclusion is required in many situations in OS resource allocation. We shall portray one such situation in the context of management of a print request. The print process usually maintains a queue of print jobs. This is done by maintaining a queue of pointers to the files that need to be printed. Processes that need to print a file store the file address (a file pointer) into this queue. The printer spooler process picks up a file address from this queue to print files. The spooler queue is a shared data structure amongst processes that are seeking the printer services and the printer spooler process. The printer spooler stores and manages the queue as shown in Figure 6.1. Let us consider just two

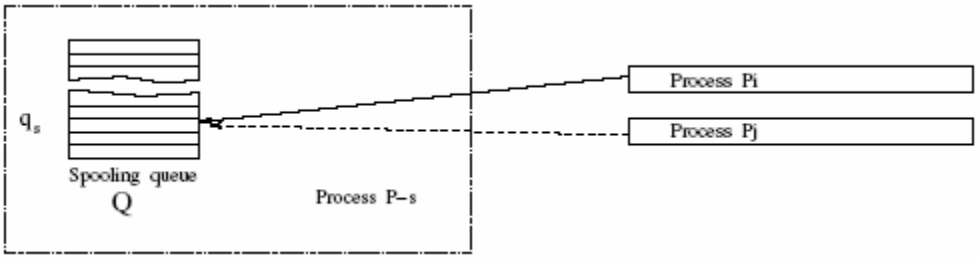

Figure 6.1: An example of mutual exclusion.

processes P_i, P_j that need to use printer. Let Ps denote the printer spooler process. The shared queue data area is denoted by Q. Let us now envision the situation as depicted below:

- Pi accesses Q and finds that a certain slot q_s is free.
- Pi decides to copy in this area the address of the file it needs to print.
- Next P_j accesses Q and also finds the slot q_s is free.
- P_j decides to copy in this area the address of file it needs to print.
- Pi copies the address of the area from which a file needs to be printed.
- Next P_j copies the address of the area from which a file needs to be printed.
- Ps reaches the slot at q_s and prints the file pointed to by the address in q_s.

On examining the above sequence we find that both the processes $P_{i,}$ and P_j may record that their print jobs are spooled. As a matter of fact only the job from P_j was spooled. The print job of Pi never gets done. If we had mutual exclusion then either process Pi or process P_j , but not both, could have had an access to q_s . A point to be noted here is that Q is a shared data area (a resource) used by three processes $P_{i,}$ P_j , and P_s. It also establishes an inter-process communication (IPC) between processes that need printing and the process which prints. Access to shared data resource that establishes inter-process communication must be mutually exclusive. We shall later revisit mutual exclusion in more detail in Section 6.5. There we shall discuss how to ensure mutually exclusive access to resources. For now let us examine the conditions for deadlocks.

6.3 Deadlocks

We can understand the notion of a deadlock from the following simple real-life example. To be able to write a letter one needs a letter pad and a pen. Suppose there in one letter pad and one pen on a table with two persons seated around the table. We shall identify these two persons as Mr. A and Ms. B. Both Mr. A and Ms. B are desirous of writing a letter. So both try to acquire the resources they need. Suppose Mr. A was able to get the letter pad. In the meantime, Ms. B was able to grab the pen. Note that each of them has one of the two resources they need to proceed to write a letter. If they hold on to the resource they possess and await the release of the resource by the other, then neither of them can proceed. They are deadlocked. We can transcribe this example for processes seeking resources to proceed with their execution.

Consider an example in which process P_1 needs three resources r_1; r_2, and r_3 before it can make any further progress. Similarly, process P_2 needs two resources r_2 and r_3. Also, let us assume that these resources are such that once granted, the permission to use is not withdrawn till the processes release these resources. The processes proceed to acquire these resources. Suppose process P_1 gets resources r_1 and r_3 and process P_2 is able to get resource r_2 only. Now we have a situation in which process P_1 is waiting for process P_2 to release r_2 before it can proceed. Similarly, process P_2 is waiting for process P_1 to release resource r_3 before it can proceed. Clearly, this situation can be recognized as a deadlock condition as neither process P_1 nor process P_2 can make progress. Formally, a deadlock is a condition that may involve two or more processes in a state such that each is waiting for release of a resource which is currently held by some other process.

A graph model: In Figure 6.2 we use a directed graph model to capture the sense of deadlock. The figure uses the following conventions.

- There are two kinds of nodes - circles and squares. Circles denote processes and squares denote resources.
- A directed arc from a process node (a circle) to a resource node denotes that the process needs that resource to proceed with its execution.
- A directed arc from a square (a resource) to a circle denotes that the resource is held by that process.

With the conventions given above, when a process has all the resources it needs, it can execute. This condition corresponds to the following.

- The process node has no arcs directed out to a resource node.
- All the arcs incident into this process node are from resource nodes.

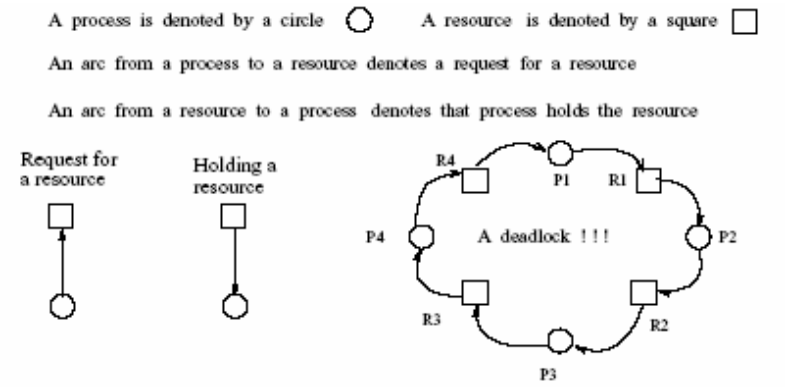

Figure 6.2: A directed graph model.

In Figure 6.2, P_1 holds r_4 but awaits release of r_1 to proceed with execution; P_2 holds r_1 but awaits release of r_2 to proceed with execution; P_3 holds r_2 but awaits release of r_3 to proceed with execution; P_4 holds r_3 but awaits release of r_4 to proceed with execution. Clearly, all the four processes are deadlocked.

Formally, a deadlock occurs when the following four conditions are present simultaneously.

- **Mutual exclusion**: Each resource can be assigned to at most one process only.
- **Hold and wait:** Processes hold a resource and may seek an additional resource.
- **No pre-emption:** Processes that have been given a resource cannot be pre-empted to release their resources.

- **Circular wait:** Every process awaits release of at least one resource held by some other processes.

Dead-lock Avoidance: A deadlock requires the above four conditions to occur at the same time, i.e. mutual exclusion, hold and wait, no pre-emption and circular wait to occur at the same time. An analysis and evaluation of the first three conditions reveals that these are necessary conditions. Also, we may note that the circular wait implies hold and wait. The question is how does one avoid having a deadlock? We shall next examine a few arguments. The first one favors having multiple copies of resources. The second one argues along preventive lines, i.e. do not permit conditions for deadlock from occurring. These arguments bring out the importance of pre-empting.

The infinite resource argument: One possibility is to have multiple resources of the same kind. In that case, when one copy is taken by some process, there is always another copy available. Sometimes we may be able to break a deadlock by having just a few additional copies of a resource. In Figure 6.3 we show that there are two copies of resource r_2. At the moment, processes P_1 and P_2 are deadlocked. When process P_3 terminates a copy of resource r_2 is released. Process P_2 can now have all the resources it needs and the deadlock is immediately broken. P_1 will get r_1 once P_2 terminates and releases the resources held.

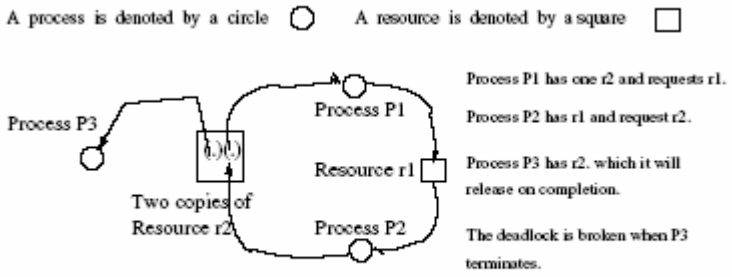

Figure 6.3: Multiple resource availability.

The next pertinent question is: how many copies of each resource do we need? Unfortunately, theoretically, we need an infinite number of copies of each resource!! Note even in this example, if P_3 is deadlocked, then the deadlock between P_1 and P_2 cannot be broken. So, we would need one more copy of resource r_2. That clearly demonstrates the limitation of the multiple copies argument.

Never let the conditions occur: It takes some specific conditions to occur at the same time to cause deadlock. This deadlock avoidance simply states that do not let these

conditions occur at the same time. Let us analyze this a bit deeper to determine if we can indeed prevent these conditions from occurring at the same time. The first condition is mutual exclusion. Unfortunately, many resources do require mutual exclusion!! So we must live with it and design our systems bearing in mind that mutual exclusion would have to be provided for. Next, let us consider the condition of hold and wait. Since, hold and wait is also implied by circular wait, we may look at the possibility of preventing any circular waits. This may be doable by analyzing program structures. Now let us examine pre-emption. It may not be the best policy to break a deadlock, but it works. Pre-emption is clearly enforceable in most, if not all, situations. Pre-emption results in releasing resources which can help some processes to progress, thereby breaking the deadlock. In fact, many real-time OSs require pre-emption for their operation. For example, when a certain critical condition arises, alarms must be set or raised. In some other cases an emergency process may even take over by pre-empting the currently running process.

6.3.1 A Deadlock Prevention Method

In a general case, we may have multiple copies of resources. Also, processes may request multiple copies of a resource. Modeling such a scenario as a graph is difficult. In such a situation, it is convenient to use a matrix model. We shall use Figure 6.4 to explain the matrix-based method. In Figure 6.4 we assume n processes and m kinds of resources. We denote the *i*th resource by ri. We now define two vectors, each of size m.

Vector R = (r_1; r_2; :::::; r_m) : r_i = resources of type i with the system.

Vector A = (a_1; a_2; :::::; a_m) : a_i = resources of type i presently available for allocation. Initially with no allocations made, we have R = A. However, as allocations happen, vector A shall be depleted. Also, when processes terminate and release their resources, vector A gets updated to show additional resources that become available now. We also define two matrices to denote allocations made and requests for the resources. There is a row for each process and a column for each resource. Matrix AM and matrix RM respectively have entries for allocation and requests. An entry $c_{i,j}$ in matrix AM denotes the number of resources of type j currently allocated to process Pi. Similarly, $q_{i,j}$ in matrix RM denotes the number of resources of type j requested by process Pi. This is depicted in Figure 6.4. Below we state the three conditions which capture the constraints for the model. The first condition always holds. The second condition holds when requests on resources exceed capacity. In this condition not all processes can execute simultaneously.

1. $\left[\sum_{i=1}^{n} c_{i,j} + a_j\right] \leq r_j$ .This condition states that the allocation of resource j to all the processes plus the now available resource of kind j is always less than the ones available with the system.
2. $\left[\sum_{i=1}^{n} q_{i,j}\right] \geq r_j$. This condition states that the requests for resources made by every process may exceed what is available on the system.
3. In addition, we have the physical constraint $\left[\forall j c_{i,j}\right] \leq q_{i,j}$. This condition states that allocation of a resource j to a process may be usually less than the request made by the process. At the very best the process's request may be fully granted.

The matrix model captures the scenario where n processes compete to acquire one or more copies of the m kinds of resources.

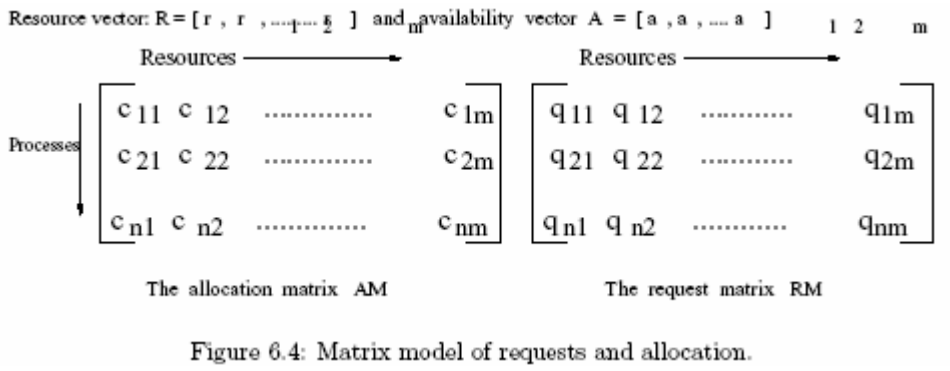

Figure 6.4: Matrix model of requests and allocation.

6.4 Deadlock Detection and Prevention Algorithms

In this section we shall discuss a few deadlock detection and prevention algorithms. We shall employ both the graph and matrix models. We begin with the simple case when we have one copy of each resource and have to make allocation to a set of processes. A graph based detection algorithm: In our digraph model with one resource of one kind, the detection of a deadlock requires that we detect a directed cycle in a processor resource digraph. This can be simply stated as follows.

- Choose a process node as a root node (to initiate a depth first traversal).
- Traverse the digraph in depth first mode.
- Mark process nodes as we traverse the graph.
- If a marked node is revisited then a deadlock exists.

In the above steps we are essentially trying to detect the presence of a directed cycle.

Bankers' Algorithm: This is a simple deadlock prevention algorithm. It is based on a banker's mind-set: “offer services at no risk to the bank". It employs a policy of resource denial if it suspects a risk of a deadlock. In other words, for a set of processes, resources are allocated only when it shall not lead to a deadlock. The algorithm checks for the four necessary conditions for deadlock. A deadlock may happen when any one of the four conditions occurs. If a deadlock is likely to happen with some allocation, then the algorithm simply does not make that allocation.

The manner of operation is as follows: The request of process i is assessed to determine whether the process request can be met from the available resources of each kind. This means $\forall j\left(q_{i,j}\right) \leq a_j$. In that case, process i may be chosen to execute. In fact, the policy always chooses that subset amongst the processes which can be scheduled to execute without a deadlock.

Let us now offer a critique of the algorithm.

1. If there are deadlocked processes, they shall remain deadlocked. Bankers' algorithm does not eliminate an existing deadlock.
2. Bankers' algorithm makes an unrealistic assumption. It stipulates that the resource requirements for processes are known in advance. This may not be rare but then there are processes which generate resource requests on the fly. These dynamically generated requirements may change during the lifetime of a process.
3. With multi-programming, the number of live processes at any one time may not be known in advance.
4. The algorithm does not stipulate any specific order in which the processes should be run. So in some situations, it may choose an order different from the desired order. Sometimes we do need processes to follow a specific order. This is true when the processes must communicate in a particular sequence (see synchronization example in Section 6.5).
5. Also, the algorithm assumes a fixed number of resources initially available on a system. This too may vary over time.

A matrix based deadlock detection method: When multiple resources of each kind are available, we use the matrix model shown in Figure 6.4 to detect deadlocks. We analyze the requests of the processes (matrix RM) against initially available copies of each

resource (vector A). This is what the algorithm below indicates. Following the description of the algorithm there is a brief explanation of the same.

Procedure detect deadlock

begin

$\forall i, 1 \leq i \leq n$ set *marked(i) = false.* These flags help us to detect marked processes whose requirements can be satisfied.

deadlockpresent = false

All entries in matrix AM are initialized to zero.

While there are processes yet to be examined do

{

Pick a process Pi whose requests have not been examined yet.

For process P_i check if $RM_i \leq A$ then

{

allocate the resources;

marked(i) = true

Add allocation made to row AM_i

Subtract this allocation from A to update *A*

}

}

If $\forall i$, *marked(i)* is *true* then *deadlockpresent = false* else *deadlockpresent = true*

end

We offer an alternative explanation for the above algorithm; let us assume that processes P_1 through P_n are to be allocated m kinds of resources. We begin with some tentative allocation of resources starting with, say, P_1 in sequence. Now let us consider an intermediate step: the allocation for process during which we are determining allocation for process P_i. The row corresponding to process P_i in matrix RM denotes its resource requests. This row gives the number for each kind of resource requested by P_i. Recall that vector A denotes the resources presently available for allocation. Now, let us suppose that resource requests of process Pi can be met. In that case, vector $RM_i \leq$ A. This means that this process could be scheduled to execute and no deadlock as yet has manifested. This allocation can then be reflected in matrix AM_i. Also, vector A needs to be modified

accordingly. Next, with the revised vector A, we try to meet the requirements of the next process P_{i+1} which is yet to be allocated with its resources. If we can exhaustively run through the sequence then we do not have a deadlock. However, at some stage we may find that its requirement of resources exceeds the available resources. Suppose this happens during the time we attempt allocation for process P_{i+k}. In that case, we have a deadlock for the subset of processes P_1 through $P_{i+k.}$ Recall that we are marking the processes that obtain their allocation. So if we have all the processes marked then there is no deadlock. If there is a set of processes that remain unmarked then we have a deadlock. Notwithstanding the non-deterministic nature of this algorithm it always detects a deadlock. Like the bankers' algorithm, this algorithm also does not help to eliminate an existing deadlock. Deadlock elimination may require pre-emption or release of resources. This may also result in a roll back in some transaction-oriented systems. This further reinforces pre-emption as an effective deadlock elimination strategy.

6.5 Mutual Exclusion Revisited: Critical Sections

We have earlier seen that for devices like printers, an OS must provide for mutual exclusion in operation. It is required for memory access as well. Whenever there is a shared area accessed by two or more processes, we have to ensure that only one process has write access at one time. The main motivation is to avoid a race condition amongst processes. When a race occurs between processes (acting independent of each other), each may annul others' operations (as we saw in our spooler example). Transaction-oriented processing is notorious for the race conditions as it may lead to data integrity problems. These problems are best taken care of by ensuring the process has exclusive access to the data area (or the resource) where it has to perform the operation. So each process operates in exclusion of access to the others. Hence, the term mutual exclusion. Next we define a critical section of a program code in this context. By a “critical section” we mean that section of code (in a process) which is executed exclusively, i.e. none of its operations can be annulled. To prevent shared variables from being overwritten by another process, a process must enter its critical section. Operating in critical sections ensures mutual exclusion of processes. Well, how does one ensure such an operation? OSs, including Unix, provides a facility called semaphore to allow processes to make use of critical sections in exclusion to other processes. A semaphore is essentially a variable which is treated in a special way. Access to a semaphore and operations on it are

permitted only when it is in a free state. If a process locks a semaphore, others cannot get access to it. However, every process must release the semaphore upon exiting its critical section. In other words, a process may enter a critical section by checking and manipulating a semaphore. When a process has entered its critical section, other processes are prevented from accessing this shared variable. When a process leaves the critical section, it changes the state of semaphore from locked to free. This permits anyone of the waiting processes to now enter their critical sections and use the shared variable. To make sure that the system actually works correctly, a notion of atomicity or indivisibility is invoked, i.e. semaphore operations are run to completion without interruptions as explained in the next section.

6.5.1 Basic Properties of Semaphores

Semaphores have the following properties.

- A semaphore takes only integer values. We, however, would limit to semaphores that take only binary values. In general, we may even have a data-structure in which every entry is a semaphore. Usually, such structures are required for establishing a set of processes that need to communicate. These are also required when a complex data structure like a record is shared.
- There are only two operations possible on a semaphore.
  - A *wait* operation on a semaphore decreases its value by one.
    wait(s): **while** *s < 0* **do** noop; s := s - 1;
  - A signal operation increments its value, i.e. signal(s): s : = s + 1;
  - A semaphore operation is atomic and indivisible. This essentially ensures both *wait* and *signal* operations are carried out without interruption. This may be done using some hardware support and can be explained as follows. Recall in Section 1.2, we noticed that in Figure 1.3, the fetch, execute and decode steps are done indivisibly. The interrupt signal is recognized by the processor only after these steps have been carried out. Essentially, this means it is possible to use disable and enable signals to enforce indivisibility of operations. The wait and signal operations are carried out indivisibly in this sense.

Note that a process is blocked (busy waits) if its wait operation evaluates to a negative semaphore value. Also, a blocked process can be unblocked when some other process executes signal operation to increment semaphore to a zero or positive value. We next show some examples using semaphores.

6.5.2 Usage of Semaphore

Suppose we have two processes P1 and P2 that need mutual exclusion. We shall use a shared semaphore variable use with an initial value = 0. This variable is accessible from both P1 and P2. We may require that both these processes have a program structure that uses repeat – until pair as a perpetual loop. The program shall have the structure as shown below:

repeat

Some process code here

wait (use);

enter the critical section (the process manipulates a shared area);

signal (use);

the rest of the process code.

until *false;*

With the repeat{until sequence as defined above, we have an infinite loop for both the processes. On tracing the operations for P_1 and P_2 we notice that only one of these processes can be in its critical section. The following is a representative operational sequence. Initially, neither process is in critical section and, therefore, use is 0.

- Process P_1 arrives at the critical section first and calls *wait (use).*
- It succeeds and enters the critical section setting *use* = - 1.
- Process P_2 wants to enter its critical section. Calls *wait* procedure.
- As *use* < 0. P_2 does a busy wait.
- Process P_1 executes *signal* and exits its critical section. *use* = 0 now.
- Process P_2 exits busy wait loop. It enters its critical section *use* = -1.

The above sequence continues.

Yet another good use of a semaphore is in synchronization amongst processes. A process typically may have a synchronizing event. Typically one process generates an event and the other process awaits the occurrence of that event to proceed further. Suppose we have our two processes, P_i and P_j . P_j can execute some statement s_j only after a statement s_i in

process Pi has been executed. This synchronization can be achieved with a semaphore se initialized to -1 as follows:

- In P_i execute the sequence s_i; signal (se);
- In P_j execute wait (se); s_j ;

Now process P_j must wait completion of s_i before it can execute s_j . With this example, we have demonstrated use of semaphores for inter-process communication. We shall next discuss a few operational scenarios where we can use semaphores gainfully.

6.5.3 Some Additional Points

The primary use of semaphore which we have seen so far was to capture when a certain resource is “in use" or “free". Unix provides a wait instruction to invoke a period of indefinite wait till a certain resource may be in use. Also, when a process grabs a resource, the resource is considered to be “locked".

So far we have seen the use of a two valued or binary semaphore. Technically, one may have a multi-valued semaphore. Such a semaphore may have more than just two values to capture the sense of multiple copies of a type of resource. Also, we can define an array of semaphores i.e. each element of the array is a semaphore. In that case, the array can be used as a combination for several resources or critical operations. This is most useful in databases where we sometimes need to lock records, and even lock fields. This is particularly true of transaction processing systems like bank accounts, air lines ticket booking, and such other systems.

Since, semaphore usually has an integer value which is stored somewhere, it is information the system can use. Therefore, there are processes with permission to access, a time stamp of creation and other system-based attributes. Lastly, we shall give syntax for defining semaphore in Unix environment.

semaphoreId = semget(key_sem, no_sem, flag_sem)

Here *semget* is a system call, *key_sem* provides a key to access, *no_sem*= defines the number of semaphores required in the set. Finally, *flag_sem* is a standard access control defined by IPC_CREAT | 644 to give a rw-r--r-- access control.

In the next chapter we shall see the use of semaphore, as also, the code for other interprocess communication mechanisms.