UNIT 4. MUTUAL EXCLUSION AND SYNCHRONIZATION

4.1 INTRODUCTION

The OS design is concerned more about management of processes and threads with respect to multiprogramming, multiprocessing and distributed processing. Concurrency is a fundamental issue for all these areas. Concurrency includes design issues like communication among processes, sharing of and competing for resources, synchronization of the activities of multiple processes, and allocation of processor time to processes. Concurrency arises in three different contexts:

- **Multiple applications:** Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications.
- **Structured applications:** As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes.
- **Operating system structure:** The same structuring advantages apply to systems programs, and we have seen that operating systems are themselves often implemented as a set of processes or threads.

In this chapter, the importance of concurrency is discussed in detail. Some of the important key words in the study of concurrency are listed below.

atomic operation	A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

One can understand the meaning of concurrency with the definition: ***concurrency is the property of program, algorithm, or problem decomposability into order-independent or partially-ordered components or units.***

4.2 PRINCIPLES OF CONCURRENCY

In a single – processor systems, the processes are switched (interleaved) among themselves (Figure 4.1) and appear to be executed simultaneously. In multiprocessing systems, the processes may be switched as well as overlapped (Figure 4.2).

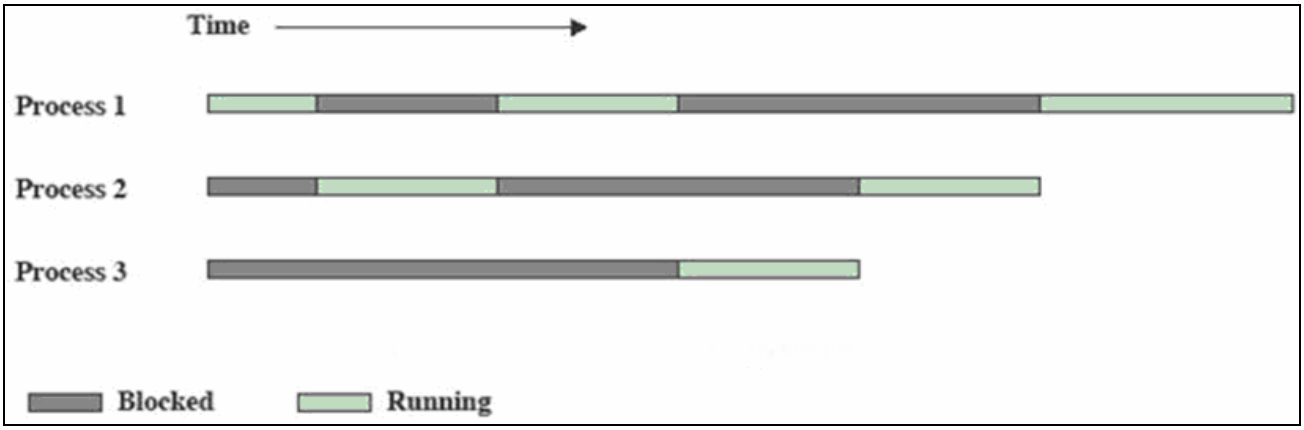

Figure 4.1 Single – processing system: Interleaving of processes

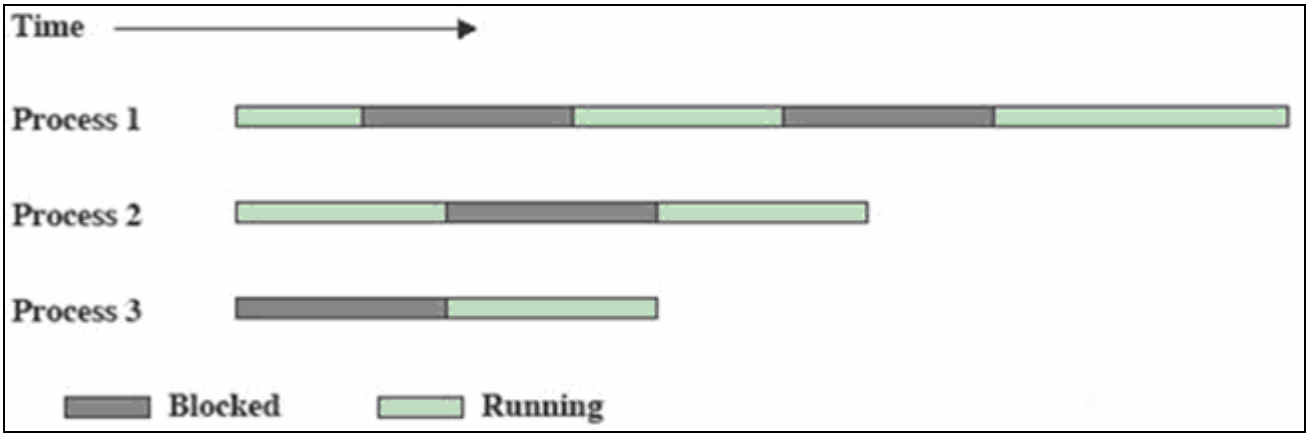

Figure 4.2 Multi – processing system: Interleaving and overlapping of processes

Interleaving and overlapping cause certain problems as listed below:

- Sharing of global resources may be dangerous
- Optimal allocation of resources may be difficult for OS
- Difficult to locating programming errors

To understand these problems, an example is considered in the following section.

4.2.1 A Simple Example

Consider the following function:

```
void echo()
{
     chin = getchar();
     chout = chin;
     putchar(chout);
}
```

It is easily observed that `chin` and `chout` are global variables. The function `echo()` reads one character from the keyboard and stores into `chin`. It is then assigned to `chout`. And then it is displayed. The `echo()` function is global and available to all applications. Hence, only a single copy of `echo()` is loaded into main memory. Sharing of such global variables will cause certain problems in both single-processor and multi-processor systems.

In a single-processor multiprogramming system, assume there is only one I/O devices (keyboard and monitor). Assume, a process P1 invokes `echo()` function and it is interrupted immediately (due to some reason) after reading a character, say, `x` from the keyboard. Now, the global variable `chin` contains `x`. At this moment of time, the process P2 invokes the function `echo()`and executes. Let the character read now be `y`. Assume, the process P2 completes its execution and displays `y`. Now, the process P1 is resumed and displays the character as `y`, because, the recent content of `chin` is being `y`.

To solve this problem, only one process should be allowed to invoke `echo()` function for a given moment of time. That is, when P1 has invoked the function and got interrupted, P2 may try to invoke the same function. But, P2 must be suspended and should be made to wait. Only after P1 comes out of the `echo()` function, P2 must be resumed.

Thus, it is necessary to control the code for protecting shared global variable (or any other global resources).

In a multiprocessor system, similar problem occurs. Consider there are two processes P1 and P2 executing on two different processors. The execution of instructions of `echo()` function for both P1 and P2 are as shown below. Note that, the instructions on same line executes in parallel.

	Process P1	Process P2
1	----------	----------
2	chin = getchar();	----------
3	----------	chin = getchar();
4	chout=chin;	chout=chin;
5	putchar(chout);	----------
6	----------	putchar(chout);
7	----------	----------

One can observe that, the global variable `chin`, initially read through P1, is overwritten by P2. Here also, to avoid the problem, control the access of shared resources.

4.2.2 Race Condition

A race condition occurs when multiple threads/processes read/write data items so that the final result depends on order of execution of instructions. Consider two examples to understand this concept.

Example 1: Assume that two processes P1 and P2 are sharing a global variable *a.* The process P1 has an instruction to update the value of *a* as –

a= 1;

The process P2 also has an instruction –

a= 2;

Both P1 and P2 will be executing their own set of instructions. Now, the value of *a* depends on the order of execution of above statements. In other words, the loser of the race (the process which executes last) will decide the value of *a*.

Example 2: Let there are two processes P3 and P3 sharing two global variables *b* and *c.* Initially,

b=1 and c = 2

The process P3 has a statement –

b=b+c;

whereas, the process P4 has a statement –

c=b+c;

One can easily make out that, the final value of *b* and *c* depends on order of execution of both of these statements. If P3 executes first,

b =3 and

c = 5

On the other hand, if P4 executes first,

b= 4 and

c = 3

4.2.3 Operating System Concerns

With respect to concurrency, following design issues OS have to be considered. The OS must

- keep track of processes
- allocate and de-allocate resources to active processes
- Protect the data and resources against interference by other processes.
- Ensure that the processes and outputs are independent of the processing speed

To understand the speed independence, the various ways of process interaction has to be considered.

4.2.4 Process Interaction

The ways in which processes are interacted is depending on how/whether they are aware of each other's existence. There are three ways as listed in Table 4.1. Now, let us discuss pros and cons of various types of relationships among processes.

Table 4.1 Process Interaction

Degree of Awareness	Relationship	Influence That One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

- **Competition among processes for resources:** Concurrent processes conflict with each other when they are competing for a same resource. In such a situation, three control problems will arise. The first one is need for ***mutual exclusion***. Suppose two processes require an access to single non-sharable resource like printer. During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data, and/or receiving data. We will refer to such a resource as a **critical resource**, and the portion of the program that uses it a **critical section** of the program. It is important that only one program at a time be allowed in its critical section. We cannot simply rely on the OS to understand and enforce this restriction. For example, in case of the printer, we want only one process to have control of the printer till it finishes printing the entire document. Otherwise, lines from competing processes will be interleaved.

  The enforcement of mutual exclusion will lead to two other control problems: ***deadlock*** and ***starvation***. Assume, process P1 is holding the resource R1 and P2 is holding the resource R2. Now, P1 is waiting for R2 for its completion, whereas P2 is waiting for R1 for its completion. None of P1 and P2 releases the respective resources and hence it is a deadlock. Consider another situation: The processes P1, P2 and P3 require periodic access to a resource R. If OS allocates R to only P1 and P2 periodically one after the other (in a round-robin manner), then the P2 will never get resource R and it will be a starvation.
  OS should provide some solution to such problems by say, locking the resource before its use.

- **Cooperation among processes by sharing:** Here, the processes are aware of existence of other processes. Hence, processes may cooperate while updating the shared data. But, as data are held on resources, the problems of mutual exclusion, deadlock and starvation are still present. On top of these, a new problem – data coherence is introduced. Data coherence means the requirement of consistency in the shared data. For example, if two global variables *a* and *b* are equal at the beginning, they should remain equal even after execution of multiple processes. If the relationship among them is *a less than b*, it should remain as it is. To maintain such data integrity, the entire sequence of instructions in the processes must be made as critical region.
- **Cooperation among processes by Communication:** Here, the processes communicate with each other by sending messages. Hence, there won't be any requirement for mutual exclusion. But, the problem of deadlock and starvation still persist.

4.2.5 Requirements for Mutual Exclusion

The facility of mutual exclusion should meet the following requirements:

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its non-critical section must do so without interfering with other processes
- No deadlock or starvation
- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

These requirements of mutual exclusion can be satisfied by number of ways:

- Leaving the responsibility of concurrent execution to processes themselves
- Usage of special-purpose machine instructions
- Providing the support within the OS or programming language

4.3 MUTUAL EXCLUSION: HARDWARE SUPPORT

There are many software algorithms for enforcing mutual exclusion. But they have high processing overhead and the risk of logical errors. Hence, we will here consider few hardware approaches.

4.3.1 Interrupt Disabling

In a single-processor system, concurrent processes cannot be overlapped, but can be interleaved. Moreover, a process will continue to execute until it is interrupted. Hence, to guarantee mutual exclusion, a process has to be prevented from being interrupted. This capability can be provided in the form of primitives defined by OS kernel for disabling and enabling interrupts. So, a process should not be interrupted till it completes its critical section. But, by doing so, the efficiency of OS will slow-down, as OS cannot switch between the processes. Moreover, this approach does not guarantee mutual exclusion in case of multi-processor systems.

4.3.2 Special Machine Instructions

Normally, access to a memory location excludes any other access to that same location. Hence, processor designers have proposed several machine instructions that carry out two actions atomically (that is, an instruction is a single step and not interrupted), such as reading and writing or reading and testing, of a single memory location with one instruction fetch cycle. During execution of the instruction, access to the memory location is blocked for any other instruction referencing that location. Two most commonly implemented instructions are explained below:

- **Compare and Swap Instruction:** It compares the contents of a memory location to a given value. If they are same, it modifies the contents of that memory location to a given new value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another process in the meantime, the write would fail. The result of the operation must indicate whether it performed the substitution; this can be done either with a simple boolean response, or by returning the value read from the memory location.
- **Exchange Instruction:** The instruction exchanges the contents of a register with that of a memory location.

The use of a special machine instruction to enforce mutual exclusion has a number of advantages:

- It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.
- It is simple and therefore easy to verify.
- It can be used to support multiple critical sections; each critical section can be defined by its own variable.

There are some serious disadvantages as well:

- **Busy waiting is employed:** Thus, while a process is waiting for access to a critical section, it continues to consume processor time.
- **Starvation is possible:** When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary. Thus, some process could indefinitely be denied access.
- **Deadlock is possible:** Consider the following scenario on a single-processor system. Process P1 executes the special instruction and enters its critical section. P1 is then interrupted to give the processor to P2, which has higher priority. If P2 now attempts to use the same resource as P1, it will be denied access because of the mutual exclusion mechanism. Thus, it will go into a busy waiting loop. However, P1 will never be dispatched because it is of lower priority than another ready process, P2.

4.4 SEMAPHORES

As discussed earlier, apart from hardware-enabled mechanism for concurrency, there are OS and programming language mechanisms as well. Semaphore is one such mechanism for providing concurrency. ***Semaphore is an integer value used for signaling among processes.*** Only three operations may be performed on a semaphore, all of which are atomic: ***initialize, decrement,*** and ***increment.*** The decrement operation is for blocking of a process, and the increment operation is for unblocking of a process.

The fundamental principle is: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal. For signaling, special variables called semaphores are used. To transmit a signal via semaphore *s*, a process executes the primitive *semSignal(s)*. To receive a signal via semaphore *s*, a process executes the primitive *semWait(s)*. If the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which only three operations are defined:

1. A semaphore may be initialized to a nonnegative integer value.
2. The *semWait* operation decrements the semaphore value. If the value becomes negative, then the process executing the *semWait* is blocked. Otherwise, the process continues execution.
3. The *semSignal* operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a *semWait* operation, if any, is unblocked.

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

Figure 4.3 Definition of semaphore primitives

From the definition of semaphores, it can be observed that:

- Before process decrements a semaphore, it is not possible to know whether the process will get blocked or not.
- After a process increments a semaphore and another process gets woken up, both processes continue running concurrently. There is no way to know which process will continue immediately on a uniprocessor system.
- When you signal a semaphore, you don't necessarily know whether another process is waiting, so the number of unblocked processes may be zero or one.

The formal definition of semaphores can be understood using the code segment given in Figure 4.3.

There a stricter version of semaphore, viz. ***binary semaphore.*** It is defined by three operations as below and explained using code segment given in Figure 4.4.

1. A binary semaphore may be initialized to 0 or 1.
2. The *semWaitB* operation checks the semaphore value. If the value is zero, then the process executing the *semWaitB* is blocked. If the value is one, then the value is changed to zero and the process continues execution.
3. The *semSignalB* operation checks to see if any processes are blocked on this semaphore (semaphore value equals 0). If so, then a process blocked by a *semWaitB* operation is unblocked. If no processes are blocked, then the value of the semaphore is set to one.

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};
void semWaitB(binary_semaphore s)
{
     if (s.value == one)
          s.value = zero;
     else {
             /* place this process in s.queue */;
             /* block this process */;
     }
}
void semSignalB(semaphore s)
{
     if (s.queue is empty())
          s.value = one;
     else {
             /* remove a process P from s.queue */;
             /* place process P on ready list */;
     }
}
```

Figure 4.4 Definition of Binary Semaphore primitives

The non-binary semaphore is also known as ***counting semaphore*** or ***general semaphore.***

A concept related to the binary semaphore is the ***mutex***. The process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1). In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.

For both counting semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore. Normally, the process that has been blocked the longest is released from the queue first – that is, in a FIFO manner. A semaphore whose definition includes this policy is called a ***strong semaphore***. A semaphore that does not specify the order in which processes are removed from the queue is a ***weak semaphore***.

4.4.1 Mutual Exclusion

The code segment given in Figure 4.5 gives a straight forward solution to the problem of mutual exclusion using a semaphore *s*. The construct **parbegin (P1, P2, …, Pn)** in this code means the following: suspend the execution of the main program; initiate concurrent execution of procedures P1, P2, …, Pn ; when all of P1, P2, …, Pn have terminated, resume the main program.

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 4.5 Mutual exclusion using semaphores

Consider *n* processes, P(i), all of which need access the same resource. Each process has a critical section used to access the resource. In each process, a *semWait(s)* is executed just before its critical section. If the value of *s* becomes negative, the process is blocked. If the value is 1, then it is decremented to 0 and the process immediately enters its critical section; because *s* is no longer positive, no other process will be able to enter its critical section.

The semaphore is initialized to 1. Thus, the first process that executes a *semWait* will be able to enter the critical section immediately, setting the value of s to 0. Any other process attempting to enter the critical section will find it busy and will be blocked, setting the value of s to –1.

4.4.2 The Producer/Consumer Problem

The producer/consumer problem is one of the most commonly faced problems in concurrent processing. The general statement is this: There are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time. **The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.**

Let us assume that the buffer is of infinite size. Consider the code segments for producer and consumer as given below:

Producer

```
while (true)
{
      /* produce item v */;
      b[in] = v;
      in++;
}
```

Consumer

```
while (true)
{
      while (in <= out)
            /* do nothing */;

      w = b[out];
      out++;
      /* consume item w */;
}
```

The Figure 4.6 illustrates the structure of buffer *b*. The producer can generate items and store them in the buffer at its own speed. Each time, an index *in* is incremented. The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer. Hence, the consumer makes sure that the producer is ahead of consumer (*in* > *out*).

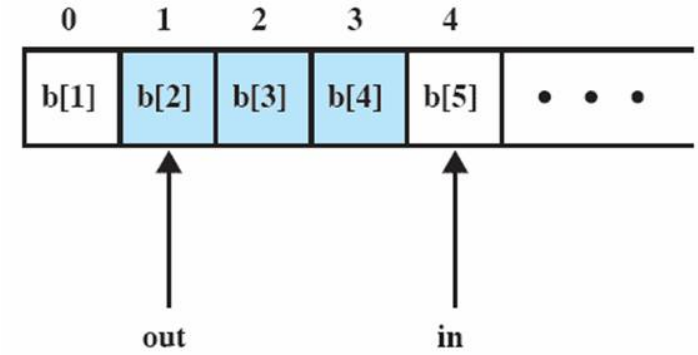

Figure 4.6 Infinite buffer for producer/consumer problem

If we add a realistic restriction on this problem, it can be solved easily using semaphore. The restriction is: instead of taking infinite buffer, consider a finite buffer of size *n* having a circular storage as shown in Figure 4.7.

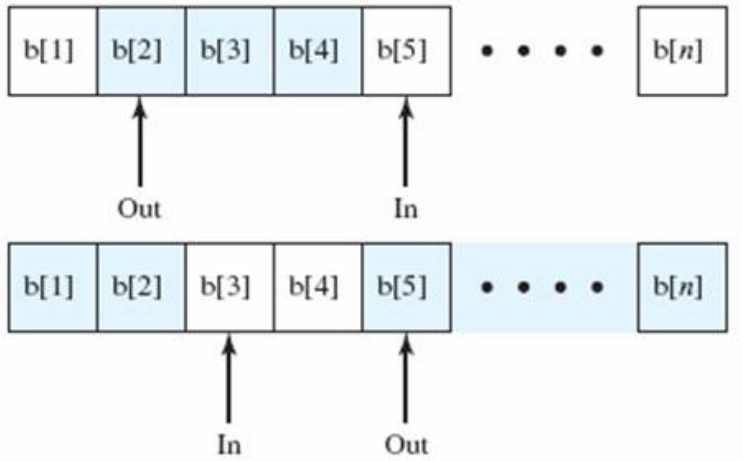

Figure 4.7 Finite circular buffer for producer/consumer problem

Since, it will work in circular manner, the pointers *in* and *out* will be considered with modulo *n*. Now, the situations would be –

Block on	Unblock on
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed

Now, the producer and consumer functions can be given as –

Producer

```
while (true)
{
      /*produce item v */
      while((in+1)%n==out)
            //do nothing
      b[in]=v;
      in=(in+1)%n;
}
```

Consumer

```
while (true)
{
      while(in==out)
            //do nothing
      w=b[out];
      out=(out+1)%n;
      /*consume item w */
}
```

4.5 MONITORS

The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control. The monitor construct has been implemented many programming languages like Concurrent Pascal, Pascal-Plus, Java etc. It has also been implemented as a program library. This allows programmers to put a monitor lock on any object. For example, we can lock all linked list with one lock, or one lock for each list or one lock for each element of every list.

4.5.1 Monitor with Signal

A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data. The chief characteristics of a monitor are the following:

1. The local variables are accessible only by the monitor's procedures and not by any external procedure.
2. A process enters the monitor by invoking one of its procedures.
3. Only one process may be executing in the monitor at a time; any other processes that have invoked the monitor are blocked, waiting for the monitor to become available.

Monitor must include synchronization tools for helping concurrency. For example, suppose a process invokes the monitor and, while in the monitor, it must be blocked until some condition is satisfied. A facility is needed by which the process is not only blocked but releases the monitor so that some other process may enter it. Later, when the condition is satisfied and the monitor is again available, the process needs to be resumed and allowed to reenter the monitor at the point of its suspension.

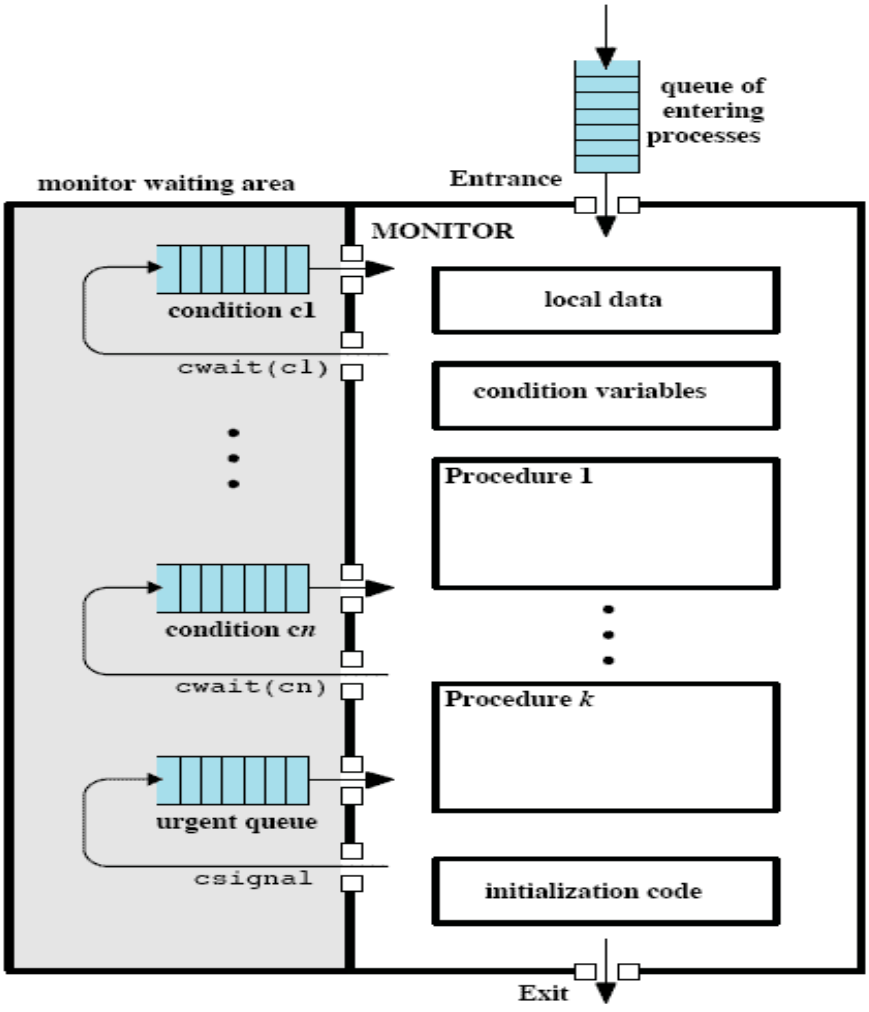

Figure 4.8 Structure of Monitor

A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor. Condition variables are a special data type in monitors, which are operated on by two functions:

- cwait(c) : Suspend execution of the calling process on condition c . The monitor is now available for use by another process.
- csignal(c) : Resume execution of some process blocked after a *cwait* on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

Figure 4.8 shows the structure of monitor.

4.6 MESSAGE PASSING

When processes interact with one another, two fundamental requirements must be satisfied:

- synchronization
- communication.

Message Passing is one solution to the second requirement. It also works with shared memory *and* with distributed systems. The actual function of message passing is normally provided in the form of a pair of primitives:

- `send (destination, message)`
- `receive (source, message)`

Various design issues related to message passing are:

- Synchronization
- Addressing
- Message Passing
- Queuing Discipline

They are discussed in the following sections.

4.6.1 Synchronization

When a message is being passed from one process to the other, there must be some level of synchronization. That is, wee need to specify what happens to a process after it issues a `send` or `receive` primitive. Three situations may arise now:

- **Blocking send, blocking receive:** Both the sender and receiver are blocked until the message is delivered; this is sometimes referred to as a ***rendezvous***. This combination allows for tight synchronization between processes.
- **Nonblocking send, blocking receive:** The sender may continue, but the receiver is blocked until the requested message arrives. This is probably the most useful combination. It allows a process to send one or more messages to a variety of destinations as quickly as possible. A process that must receive a message before it can do useful work needs to be blocked until such a message arrives.
- **Nonblocking send, nonblocking receive**: Neither party is required to wait.

The non-blocking send is most generic and useful in many situations. Still, it has a problem: an error could lead to a situation in which a process repeatedly generates messages. Because there is no blocking to discipline the process, these messages could consume system resources, including processor time and buffer space. Also, the nonblocking send places the burden on the programmer to determine that a message has been received: Processes must employ reply messages to acknowledge receipt of a message.

For `receive` primitive, blocking seems to be good idea. But, if the message is lost (say, in a distributed system setup) before it is being received, then the process may remain blocked for indefinite time.

4.6.2 Addressing

It is necessary to specify in the `send` primitive that to which process the message is being sent. Similarly, a `receive` primitive may get to know from where it is receiving the message. To maintain this information, two types of addressing are used.

- **Direct Addressing:** Here, the `send` primitive includes the identifier (or address) of the destination process. The `receive` primitive may include the information about the source process. That is, the receiving process has prior knowledge about the sending process. But, in some cases, it is impossible to know the sender. For example, a printer connected to a networked system will not know from where it is receiving the print command before receiving the actual command. Hence, after receiving the message, the receiver can acknowledge the sender.
- **Indirect Addressing:** Here, messages are not sent directly from sender to receiver but rather are sent to a shared data structure consisting of queues that can temporarily hold messages. Such queues are generally referred to as *mailboxes*. Thus, for two processes to communicate, one process sends a message to the appropriate mailbox and the other process picks up the message from the mailbox.

The indirect addressing facilitates decoupling of sender and receiver and hence allows greater flexibility. The relationship between sender and receiver may be (Figure 4.9):

- **one-to-one:** allows a private communications link to be set up between two processes. There won't be any interference.
- **one-to-many:** allows for one sender and multiple receivers; it is useful for applications where a message or some information is to be broadcast to a set of processes.
- **many-to-one:** is useful for client/server interaction; one process provides service to a number of other processes. In this case, the mailbox is often referred to as a port.
- **many-to-many:** A many-to-many relationship allows multiple server processes to provide concurrent service to multiple clients.

The association of mailboxes to processes may be static or dynamic. Usually, one-to-one relationship is static. Ports are statically associated and owned by the receivers. In other two cases, the association will by dynamic and managed by OS.

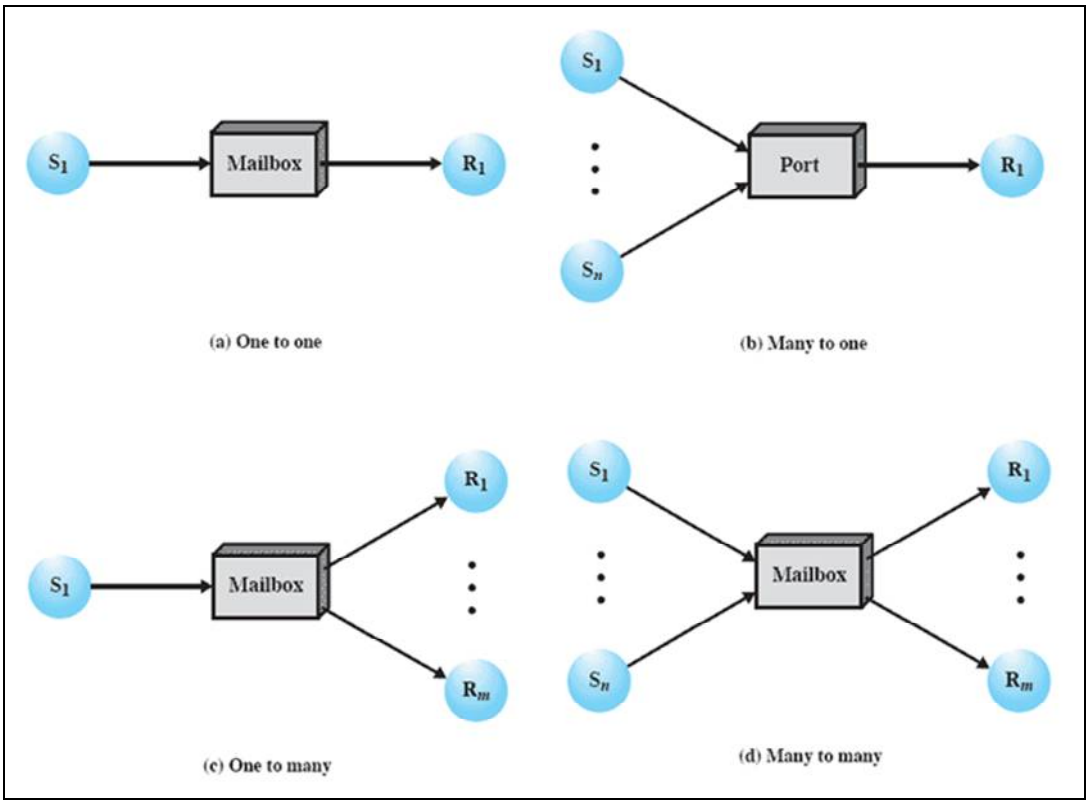

Figure 4.9 Indirect process communication

4.6.3 Message Format

The format of the message depends on the objectives of the messaging facility and whether the facility runs on a single computer or on a distributed system. Figure 4.10 shows a typical message format for operating systems that support variable-length messages.

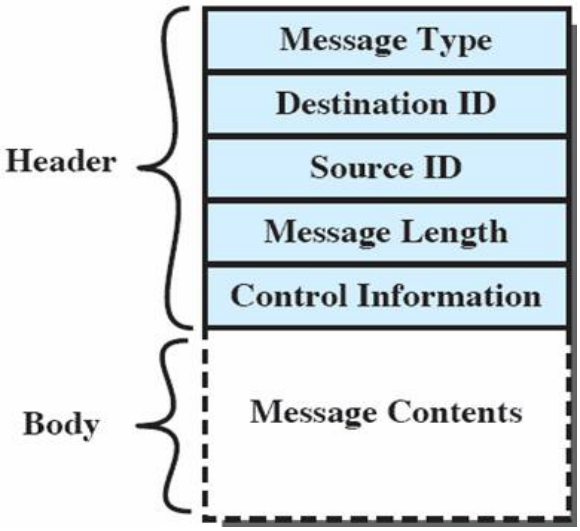

Figure 4.10 General Message format

The message is divided into two parts:

- **header**, which contains information about the message.
  - The header may contain an identification of the source and intended destination of the message, a length field, and a type field to discriminate among various types of messages.
  - additional control information, e.g. pointer field so a linked list of messages can be created; a sequence number, to keep track of the number and order of messages passed between source and destination; and a priority field.
- **body**, which contains the actual contents of the message.

4.6.4 Queuing Discipline

The simplest queuing discipline is first-in-first-out, but this may not be sufficient if some messages are more urgent than others. An alternative is to allow the specifying of message priority, on the basis of message type or by designation by the sender. Another alternative is to allow the receiver to inspect the message queue and select which message to receive next.

4.7 READERS/WRITERS PROBLEM

The readers/writers problem is defined as follows:

There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or a set of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers). The conditions that must be satisfied are as follows:

1. Any number of readers may simultaneously read the file.
2. Only one writer at a time may write to the file.
3. If a writer is writing to the file, no reader may read it.

Thus, we can make out that, a writer wants every other process to be blocked for its execution; whereas, a reader need not block other processes.

The problem can be solved with the help of semaphores. Here, there are two ways: readers having the priority and writers having priority.

4.7.1 Readers Have Priority

Here, once a single reader has begun to access the data area, it is possible for readers to retain control of the data area as long as there is at least one reader in the act of reading. The procedure is explained here –

- The first reader blocks if there is a writer; any other readers who try to enter block on mutex.
- While exiting, the last reader gives signal to a waiting writer.
- When a writer exits, if there is both a reader and writer waiting, which goes next depends on the scheduler.
- If a writer exits and a reader goes next, then all other waiting readers will get a chance to access.

But, in this methodology, writers are subject to starvation.

4.7.2 Writers have Priority

Unlike the previous case, here writers are decision makers. No new readers are allowed access to the data area once at least one writer has declared a desire to write. This is accomplished by forcing every reader to lock and release the semaphore individually. The writers on the other hand don't need to lock it individually. Only the first writer will lock the semaphore and then all subsequent writers can simply use the resource as it gets freed by the previous writer. The very last writer must release the semaphore, thus opening the gate for readers to try reading.

In this methodology, the readers may starve.

As both solutions using semaphore will lead to starvation, another solution using message passing can be adopted. In this case, there is a controller process that has access to the shared data area. Other processes wishing to access the data area send a request message to the controller, are granted access with an “OK” reply message, and indicate completion of access with a “finished” message. The controller is equipped with three mailboxes, one for each type of message (i.e. – a request, an OK, a finished) that it may receive. The controller process services write request messages before read request messages to give writers priority. In addition, mutual exclusion must be enforced.