UNIT 3 INTERPROCESS COMMUNICATION AND SYNCHRONIZATION

Structure **Page Nos.**

3.0 INTRODUCTION

In the earlier unit we have studied the concept of processes. In addition to process scheduling, another important responsibility of the operating system is process synchronization. Synchronization involves the orderly sharing of system resources by processes.

Concurrency specifies two or more sequential programs (a sequential program specifies sequential execution of a list of statements) that may be executed concurrently as a parallel process. For example, an airline reservation system that involves processing transactions from many terminals has a natural specification as a concurrent program in which each terminal is controlled by its own sequential process. Even when processes are not executed simultaneously, it is often easier to structure as a collection of cooperating sequential processes rather than as a single sequential program.

A simple batch operating system can be viewed as 3 processes−a reader process, an executor process and a printer process. The reader reads cards from card reader and places card images in an input buffer. The executor process reads card images from input buffer and performs the specified computation and store the result in an output buffer. The printer process retrieves the data from the output buffer and writes them to a printer. Concurrent processing is the basis of operating system which supports multiprogramming.

The operating system supports concurrent execution of a program without necessarily supporting elaborate form of memory and file management. This form of operation is also known as multitasking. One of the benefits of multitasking is that several processes can be made to cooperate in order to achieve their goals. To do this, they must do one of the following:

Communicate: Interprocess communication (IPC) involves sending information from one process to another. This can be achieved using a "mailbox" system, a socket

which behaves like a virtual communication network (loopback), or through the use of "pipes". Pipes are a system construction which enable one process to open another process as if it were a file for writing or reading.

Share Data: A segment of memory must be available to both the processes. (Most memory is locked to a single process).

Waiting: Some processes wait for other processes to give a signal before continuing. This is an issue of synchronization.

In order to cooperate concurrently executing processes must communicate and synchronize. Interprocess communication is based on the use of *shared variables* (variables that can be referenced by more than one process) or *message passing*.

Synchronization is often necessary when processes communicate. Processes are executed with unpredictable speeds. Yet to communicate one process must perform some action such as setting the value of a variable or sending a message that the other detects. This only works if the events perform an action or detect an action are constrained to happen in that order. Thus one can view synchronization as a set of constraints on the ordering of events. The programmer employs a synchronization mechanism to delay execution of a process in order to satisfy such constraints.

In this unit, let us study the concept of interprocess communication and synchronization, need of semaphores, classical problems in concurrent processing, critical regions, monitors and message passing.

3.1 OBJECTIVES

After studying this unit, you should be able to:

- identify the significance of interprocess communication and synchronization;
- describe the two ways of interprocess communication namely shared memory and message passing;
- discuss the usage of semaphores, locks and monitors in interprocess and synchronization, and
- solve classical problems in concurrent programming.

3.2 INTERPROCESS COMMUNICATION

Interprocess communication (IPC) is a capability supported by operating system that allows one *process* to communicate with another process. The processes can be running on the same computer or on different computers connected through a network. IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprogramming systems, but it is not generally supported by single-process operating systems such as DOS. OS/2 and MS-Windows support an IPC mechanism called Dynamic Data Exchange.

IPC allows the process to communicate and to synchronize their actions without sharing the same address space. This concept can be illustrated with the example of a shared printer as given below:

Consider a machine with a single printer running a time-sharing operation system. If a process needs to print its results, it must request that the operating system gives it access to the printer's device driver. At this point, the operating system must decide whether to grant this request, depending upon whether the printer is already being used by another process. If it is not, the operating system should grant the request and allow the process to continue; otherwise, the operating system should deny the request and perhaps classify the process as a waiting process until the printer becomes

available. Indeed, if two processes were given simultaneous access to the machine's printer, the results would be worthless to both.

Consider the following related definitions to understand the example in a better way:

Critical Resource: It is a resource shared with constraints on its use (e.g., memory, files, printers, etc).

Critical Section: It is code that accesses a critical resource.

Mutual Exclusion: At most one process may be executing a critical section with respect to a particular critical resource simultaneously.

In the example given above, the printer is the *critical resource*. Let's suppose that the processes which are sharing this resource are called process A and process B. The *critical sections* of process A and process B are the sections of the code which issue the print command. In order to ensure that both processes do not attempt to use the printer at the same, they must be granted *mutually exclusive* access to the printer driver.

First we consider the interprocess communication part. There exist two complementary inter-process communication types: a) shared-memory system and b) message-passing system. It is clear that these two schemes are not mutually exclusive, and could be used simultaneously within a single operating system.

3.2.1 Shared-Memory System

Shared-memory systems require communication processes to share some variables. The processes are expected to exchange information through the use of these shared variables. In a shared-memory scheme, the responsibility for providing communication rests with the application programmers. The operating system only needs to provide shared memory.

A critical problem occurring in shared-memory system is that two or more processes are reading or writing some shared variables or shared data, and the final results depend on who runs precisely and when. Such situations are called *race conditions*. In order to avoid race conditions we must find some way to prevent more than one process from reading and writing shared variables or shared data at the same time, i.e., we need the concept of *mutual exclusion* (which we will discuss in the later section*)*. It must be sure that if one process is using a shared variable, the other process will be excluded from doing the same thing.

3.2.2 Message-Passing System

Message passing systems allow communication processes to exchange messages. In this scheme, the responsibility rests with the operating system itself.

The function of a message-passing system is to allow processes to communicate with each other without the need to resort to shared variable. An interprocess communication facility basically provides two operations: *send* (message) and *receive* (message). In order to send and to receive messages, a communication link must exist between two involved processes. This link can be implemented in different ways. The possible basic implementation questions are:

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of process?
- What is the capacity of a link? That is, does the link have some buffer space? If so, how much?
- What is the size of the message? Can the link accommodate variable size or fixed-size message?
- Is the link unidirectional or bi-directional?

In the following we consider several methods for logically implementing a communication link and the send/receive operations. These methods can be classified

into two categories: a) Naming, consisting of direct and indirect communication and b) Buffering, consisting of capacity and messages proprieties.

Direct Communication

In direct communication, each process that wants to send or receive a message must explicitly name the recipient or sender of the communication. In this case, the send and receive primitives are defined as follows:

- send (P, message) To send a message to process P
- receive (Q, message) To receive a message from process Q

This scheme shows the *symmetry* in addressing, i.e., both the sender and the receiver have to name one another in order to communicate. In contrast to this, *asymmetry* in addressing can be used, i.e., only the sender has to name the recipient; the recipient is not required to name the sender. So the send and receive primitives can be defined as follows:

- send (P, message) To send a message to the process P
- receive (id, message) To receive a message from any process; *id* is set to the name of the process with whom the communication has taken place.

Indirect Communication

With indirect communication, the messages are sent to, and received from a *mailbox*. A mailbox can be abstractly viewed as an object into which messages may be placed and from which messages may be removed by processes. In order to distinguish one from the other, each mailbox owns a unique identification. A process may communicate with some other process by a number of different mailboxes. The send and receive primitives are defined as follows:

- send (A, message) To send a message to the mailbox A
- receive (A, message) To receive a message from the mailbox A

Mailboxes may be owned either by a process or by the system. If the mailbox is owned by a process, then we distinguish between the *owner* who can only receive from this mailbox and *user* who can only send message to the mailbox. When a process that owns a mailbox terminates, its mailbox disappears. Any process that sends a message to this mailbox must be notified in the form of an exception that the mailbox no longer exists.

If the mailbox is owned by the operating system, then it has an existence of its own, i.e., it is independent and not attached to any particular process. The operating system provides a mechanism that allows a process to: a) create a new mailbox, b) send and receive message through the mailbox and c) destroy a mailbox. Since all processes with access rights to a mailbox may terminate, a mailbox may no longer be accessible by any process after some time. In this case, the operating system should reclaim whatever space was used for the mailbox.

Capacity Link

A link has some capacity that determines the number of messages that can temporarily reside in it. This propriety can be viewed as a queue of messages attached to the link. Basically there are three ways through which such a queue can be implemented:

***Zero capacity*:** This link has a message queue length of zero, i.e., no message can wait in it. The sender must wait until the recipient receives the message. The two processes must be synchronized for a message transfer to take place. The zero-capacity link is referred to as a message-passing system without buffering.

***Bounded capacity*:** This link has a limited message queue length of *n*, i.e., at most *n* messages can reside in it. If a new message is sent, and the queue is not full, it is placed in the queue either by copying the message or by keeping a pointer to the

message and the sender should continue execution without waiting. Otherwise, the sender must be delayed until space is available in the queue.

***Unbounded capacity*:** This queue has potentially infinite length, i.e., any number of messages can wait in it. That is why the sender is never delayed.

Bounded and unbounded capacity link provide message-passing system with automatic buffering.

Messages

Messages sent by a process may be one of three varieties: a) fixed-sized, b) variable-sized and c) typed messages. If only fixed-sized messages can be sent, the physical implementation is straightforward. However, this makes the task of programming more difficult. On the other hand, variable-size messages require more complex physical implementation, but the programming becomes simpler. Typed messages, i.e., associating a type with each mailbox, are applicable only to indirect communication. The messages that can be sent to, and received from a mailbox are restricted to the designated type.

3.3 INTERPROCESS SYNCHRONIZATION

When two or more processes work on the same data simultaneously strange things can happen. Suppose, when two parallel threads attempt to update the same variable simultaneously, the result is unpredictable. The value of the variable afterwards depends on which of the two threads was the last one to change the value. This is called a *race condition*. The value depends on which of the threads wins the race to update the variable. What we need in a mulitasking system is a way of making such situations predictable. This is called *serialization*. Let us study the serialization concept in detail in the next section.

3.3.1 Serialization

The key idea in process synchronization is *serialization*. This means that we have to go to some pains to *undo* the work we have put into making an operating system perform several tasks in parallel. As we mentioned, in the case of print queues, parallelism is not always appropriate.

Synchronization is a large and difficult topic, so we shall only undertake to describe the problem and some of the principles involved here.

There are essentially two strategies to serializing processes in a multitasking environment.

- The scheduler can be disabled for a short period of time, to prevent control being given to another process during a critical action like modifying shared data. This method is very inefficient on multiprocessor machines, since all other processors have to be halted every time one wishes to execute a critical section.

- A protocol can be introduced which all programs sharing data must obey. The protocol ensures that processes have to queue up to gain access to shared data. Processes which ignore the protocol ignore it at their own peril (and the peril of the remainder of the system!). This method works on multiprocessor machines also, though it is more difficult to visualize. The responsibility of serializing important operations falls on programmers. The OS cannot impose any restrictions on silly behaviour−it can only provide tools and mechanisms to assist the solution of the problem.

3.3.2 Mutexes: Mutual Exclusion

When two or more processes must share some object, an arbitration mechanism is needed so that they do not try to use it at the same time. The particular object being

shared does not have a great impact on the choice of such mechanisms. Consider the following examples: two processes sharing a printer must take turns using it; if they attempt to use it simultaneously, the output from the two processes may be mixed into an arbitrary jumble which is unlikely to be of any use. Two processes attempting to update the same bank account must take turns; if each process reads the current balance from some database, updates it, and then writes it back, one of the updates will be lost.

Both of the above examples can be solved if there is some way for each process to exclude the other from using the shared object during critical sections of code. Thus the general problem is described as the mutual exclusion problem. The mutual exclusion problem was recognised (and successfully solved) as early as 1963 in the Burroughs AOSP operating system, but the problem is sufficiently difficult widely understood for some time after that. A significant number of attempts to solve the mutual exclusion problem have suffered from two specific problems, the lockout problem, in which a subset of the processes can conspire to indefinitely lock some other process out of a critical section, and the deadlock problem, where two or more processes simultaneously trying to enter a critical section lock each other out.

On a uni-processor system with non-preemptive scheduling, mutual exclusion is easily obtained: the process which needs exclusive use of a resource simply refuses to relinquish the processor until it is done with the resource. A similar solution works on a preemptively scheduled uni-processor: the process which needs exclusive use of a resource disables interrupts to prevent preemption until the resource is no longer needed. These solutions are appropriate and have been widely used for short critical sections, such as those involving updating a shared variable in main memory. On the other hand, these solutions are not appropriate for long critical sections, for example, those which involve input/output. As a result, users are normally forbidden to use these solutions; when they are used, their use is restricted to system code.

Mutual exclusion can be achieved by a system of *locks*. A mutual exclusion lock is colloquially called a *mutex*. You can see an example of mutex locking in the multithreaded file reader in the previous section. The idea is for each thread or process to try to obtain locked-access to shared data:

```
Get_Mutex(m);

// Update shared data

Release_Mutex(m);
```

This protocol is meant to ensure that only one process at a time can get past the function Get_Mutex. All other processes or threads are made to wait at the function Get_Mutex until that one process calls Release_Mutex to release the lock. A method for implementing this is discussed below. Mutexes are a central part of multithreaded programming.

3.3.3 Critical Sections: The Mutex Solution

A critical section is a part of a program in which is it necessary to have exclusive access to shared data. Only one process or a thread may be in a critical section at any one time. The characteristic properties of the code that form a Critical Section are:

- Codes that refer one or more variables in a “read-update-write” fashion while any of those variables is possibly being altered by another thread.
- Codes that alter one or more variables that are possibly being referenced in “read-update-write” fashion by another thread.
- Codes use a data structure while any part of it is possibly being altered by another thread.
- Codes alter any part of a data structure while it possibly in use by another thread.

In the past it was possible to implement this by generalising the idea of interrupt masks. By switching off interrupts (or more appropriately, by switching off the scheduler) a process can guarantee itself uninterrupted access to shared data. This method has drawbacks:

i) Masking interrupts can be dangerous– there is always the possibility that important interrupts will be missed;

ii) It is not general enough in a multiprocessor environment, since interrupts will continue to be serviced by other processors–so all processors would have to be switched off;

iii) It is too harsh. We only need to prevent two programs from being in their critical sections simultaneously if they share the same data. Programs A and B might share different data to programs C and D, so why should they wait for C and D?

In 1981 G.L. Peterson discovered a simple algorithm for achieving mutual exclusion between two processes with PID equal to 0 or 1. The code is as follows:

```
int turn;
int interested[2];

void Get_Mutex (int pid)

{
int other;
other = 1 - pid;
interested[process] = true;
turn = pid;

while (turn == pid && interested[other])  // Loop until no one
  {                                       // else is interested
  }
}

Release_Mutex (int pid)

{
interested[pid] = false;
}
```

Where more processes are involved, some modifications are necessary to this algorithm. The key to serialization here is that, if a second process tries to obtain the mutex, when another already has it, it will get caught in a loop, which does not terminate until the other process has released the mutex. This solution is said to involve *busy waiting*–i.e., the program actively executes an empty loop, wasting CPU cycles, rather than moving the process out of the scheduling queue. This is also called a *spin lock*, since the system 'spins' on the loop while waiting. Let us see another algorithm which handles critical section problem for ***n*** processes.

3.3.4 Dekker's solution for Mutual Exclusion

Mutual exclusion can be assured even when there is no underlying mechanism such as the test-and-set instruction. This was first realised by *T. J. Dekker* and published (by *Dijkstra*) in 1965. Dekker's algorithm uses busy waiting and works for only two processes. The basic idea is that processes record their interest in entering a critical section (in Boolean variables called "need") and they take turns (using a variable called "turn") when both need entry at the same time. Dekker's solution is shown below:

```
type processid = 0..1;
```

```
var need: array [ processid ] of boolean { initially false };
   turn: processid { initially either 0 or 1 };

procedure dekkerwait( me: processid );
var other: processid;
begin
    other := 1 - me;
    need[ me ] := true;
    while need[ other ] do begin { there is contention }
        if turn = other then begin
            need[ me ] := false { let other take a turn };
            while turn = other do { nothing };
            need[ me ] := true { re-assert my interest };
        end;
    end;
end { dekkerwait };

procedure dekkersignal( me: processid );
begin
    need[ me ] := false;
    turn := 1 - me { now, it is the other's turn };
end { dekkersignal };
```

Dekkers solution to the mutual exclusion problem requires that each of the contending processes have a unique process identifier which is called “me” which is passed to the wait and signal operations. Although none of the previously mentioned solutions require this, most systems provide some form of process identifier which can be used for this purpose.

It should be noted that Dekker’s solution does rely on one very simple assumption about the underlying hardware; it assumes that if two processes attempt to write two different values in the same memory location at the same time, one or the other value will be stored and not some mixture of the two. This is called the atomic update assumption. The atomically updatable unit of memory varies considerably from one system to another; on some machines, any update of a word in memory is atomic, but an attempt to update a byte is not atomic, while on others, updating a byte is atomic while words are updated by a sequence of byte updates.

3.3.5 Bakery’s Algorithm

Bakery algorithm handles critical section problem for *n* processes as follows:

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
- Notation <= lexicographical order (ticket #, process id #)
    - (a,b) < (c,d) if a < c or if a = c and b < d
    - max(a_0, . . . , a_{n-1}) is a number, k , such that $k >= a_i$ for i = 0, . . . , n - 1
- Shared data

  boolean choosing[n]; //initialise all to false
  int number[n]; //initialise all to 0

- Data structures are initialized to false and 0, respectively.

The algorithm is as follows:

```
do
 {
  choosing[i] = true;
  number[i] = max(number[0], number[1], ...,number[n-1]) + 1;
  choosing[i] = false;

    for(int j = 0; j < n; j++)
     {
            while (choosing[j]== true)
            {
            /*do nothing*/
            }

            while ((number[j]!=0) &&
             (number[j],j)< (number[i],i))
                    // see Reference point
                {
                    /*do nothing*/
                 }
      }

        do critical section
             number[i] = 0;
        do remainder section
}while (true)
```

In the next section we will study how the semaphores provides a much more organise approach of synchronization of processes.

3.4 SEMAPHORES

Semaphores provide a much more organised approach to controlling the interaction of multiple processes than would be available if each user had to solve all interprocess communications using simple variables, but more organization is possible. In a sense, semaphores are something like the ***goto*** statement in early programming languages; they can be used to solve a variety of problems, but they impose little structure on the solution and the results can be hard to understand without the aid of numerous comments. Just as there have been numerous control structures devised in sequential programs to reduce or even eliminate the need for ***goto*** statements, numerous specialized concurrent control structures have been developed which reduce or eliminate the need for semaphores.

Definition: The effective synchronization tools often used to realise mutual exclusion in more complex systems are semaphores. A semaphore ***S*** is an integer variable which can be accessed only through two standard atomic operations: ***wait and signal***. The definition of the wait and signal operation are:

wait(S): while S ≤ 0 do skip;
S := S – 1;

signal(S): S := S + 1;

or in C language notation we can write it as:

```
wait(s)
 {
  while (S<=0)
```

```
  {
    /*do nothing*/
  }
  S= S-1;
}

signal(S)
{
  S = S + 1;
}
```

It should be noted that the test *(S ≤ 0)* and modification of the integer value of S which is S := S – 1 must be executed without interruption. In general, if one process modifies the integer value of S in the wait and signal operations, no other process can simultaneously modify that same S value. We briefly explain the usage of semaphores in the following example:

Consider two currently running processes: P_1 with a statement S_1 and P_2 with a statement S_2. Suppose that we require that S_2 be executed only after S_1 has completed. This scheme can be implemented by letting P_1 and P_2 share a common semaphore *synch*, initialised to 0, and by inserting the statements:

S_1;
signal(synch);
in the process P_1 and the statements:
wait(synch);
S_2;
in the process P_2.

Since *synch* is initialised to 0, P_2 will execute S_2 only after P_1 has involved signal (synch), which is after S_1.

The disadvantage of the semaphore definition given above is that it requires *busy-waiting*, i.e., while a process is in its critical region, any either process it trying to enter its critical region must continuously loop in the entry code. It's clear that through busy-waiting, CPU cycles are wasted by which some other processes might use those productively.

To overcome busy-waiting, we modify the definition of the wait and signal operations. When a process executes the wait operation and finds that the semaphore value is not positive, the process *blocks* itself. The block operation places the process into a waiting state. Using a scheduler the CPU then can be allocated to other processes which are ready to run.

A process that is blocked, i.e., waiting on a semaphore S, should be restarted by the execution of a signal operation by some other processes, which changes its state from blocked to ready. To implement a semaphore under this condition, we define a semaphore as:

```
struct semaphore
 {
  int value;
  List *L;   //a list of processes
 }
```

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore, it is added to this list. A signal operation removes one process from the list of waiting processes, and awakens it. The semaphore operation can be now defined as follows:

```
wait(S)
{
  S.value = S.value -1;
  if (S.value <0)
```

```
  {
    add this process to S.L;
    block;
  }
 }
 signal(S)
 {
  S.value = S.value + 1;
  if (S.value <= 0)
  {
    remove a process P from S.L;
    wakeup(P);
  }
 }
```

The block operation suspends the process. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

One of the almost critical problem concerning implementing semaphore is the situation where two or more processes are waiting *indefinitely* for an event that can be only caused by one of the waiting processes: these processes are said to be *deadlocked*. To illustrate this, consider a system consisting of two processes P_1 and P_2, each accessing two semaphores S and Q, set to the value one:

P_1	P_2
wait(S);	wait(Q);
wait(Q);	wait(S);
...	...
signal(S);	signal(Q);
signal(Q);	signal(S);

Suppose P_1 executes wait(S) and then P_2 executes wait(Q). When P_1 executes wait(Q), it must wait until P_2 executes signal(Q). It is no problem, P_2 executes wait(Q), then signal(Q). Similarly, when P_2 executes wait(S), it must wait until P_1 executes signal(S). Thus these signal operations cannot be carried out, P_1 and P_2 are deadlocked. It is clear, that a set of processes are in a deadlocked state, when every process in the set is waiting for an event that can only be caused by another process in the set.

3.5 CLASSICAL PROBLEMS IN CONCURRENT PROGRAMMING

In this section, we present a large class of concurrency control problems. These problems are used for testing nearly every newly proposed synchronization scheme.

3.5.1 Producers/Consumers Problem

Producer – Consumer processes are common in operating systems. The problem definition is that, a producer (process) produces the information that is consumed by a consumer (process). For example, a compiler may produce assembly code, which is consumed by an assembler. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized. These problems can be solved either through unbounded buffer or bounded buffer.

- **With an unbounded buffer**

  The unbounded-buffer producer- consumer problem places no practical limit on the size of the buffer .The consumer may have to wait for new items, but the

producer can always produce new items; there are always empty positions in the buffer.

- **With a bounded buffer**

  The bounded buffer producer problem assumes that there is a fixed buffer size. In this case, the consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

Shared Data

```
char item;              //could be any data type
char                buffer[n];
semaphore full = 0;     //counting semaphore
semaphore empty = n;    //counting semaphore
semaphore mutex = 1;    //binary semaphore
char        nextp,nextc;
```

Producer Process

```
do
{
produce an item in nextp
wait (empty);
wait (mutex);
add nextp to buffer
signal (mutex);
signal (full);
}
while (true)
```

Consumer Process

```
do
{
wait( full );
wait( mutex );
remove an item from buffer to nextc
signal( mutex );
signal( empty );
consume the item in nextc;
}
```

3.5.2 Readers and Writers Problem

The readers/writers problem is one of the classic synchronization problems. It is often used to compare and contrast synchronization mechanisms. It is also an eminently used practical problem. A common paradigm in concurrent applications is isolation of shared data such as a variable, buffer, or document and the control of access to that data. This problem has two types of clients accessing the shared data. The first type, referred to as readers, only wants to read the shared data. The second type, referred to as writers, may want to modify the shared data. There is also a designated central data server or controller. It enforces exclusive write semantics; if a writer is active then no other writer or reader can be active. The server can support clients that wish to both read and write. The readers and writers problem is useful for modeling processes which are competing for a limited shared resource. Let us understand it with the help of a practical example:

An airline reservation system consists of a huge database with many processes that read and write the data. Reading information from the database will not cause a problem since no data is changed. The problem lies in writing information to the database. If no constraints are put on access to the database, data may change at any moment. By the time a reading process displays the result of a request for information

to the user, the actual data in the database may have changed. What if, for instance, a process reads the number of available seats on a flight, finds a value of one, and reports it to the customer? Before the customer has a chance to make their reservation, another process makes a reservation for another customer, changing the number of available seats to zero.

The following is the solution using semaphores:

Semaphores can be used to restrict access to the database under certain conditions. In this example, semaphores are used to prevent any writing processes from changing information in the database while other processes are reading from the database.

```
semaphore mutex = 1;    // Controls access to the reader count
semaphore db = 1;       // Controls access to the database
int reader_count; // The number of reading processes accessing the data

Reader()
{
  while (TRUE) {                // loop forever
    down(&mutex);               // gain access to reader_count
    reader_count = reader_count + 1;  // increment the reader_count
    if (reader_count == 1)
       down(&db);   //If this is the first process to read the database,
                            // a down on db is executed to prevent access to the
                            // database by a writing process
    up(&mutex);                     // allow other processes to access reader_count
    read_db();                      // read the database
    down(&mutex);                   // gain access to reader_count
    reader_count = reader_count - 1;       // decrement reader_count
    if (reader_count == 0)
       up(&db);                     // if there are no more processes reading from the
                            // database, allow writing process to access the data
    up(&mutex);                     // allow other processes to access
reader_countuse_data();
                            // use the data read from the database (non-critical)
}

Writer()
{
  while (TRUE) {                // loop forever
    create_data();                  // create data to enter into database (non-critical)
    down(&db);                      // gain access to the database
    write_db();                     // write information to the database
    up(&db);                        // release exclusive access to the database
}
```

3.5.3 Dining Philosophers Problem

Five philosophers sit around a circular table. Each philosopher spends his life alternatively thinking and eating. In the centre of the table is a large bowl of rice. A philosopher needs two chopsticks to eat. Only 5 chop sticks are available and a chopstick is placed between each pair of philosophers. They agree that each will only use the chopstick to his immediate right and left. From time to time, a philosopher gets hungry and tries to grab the two chopsticks that are immediate left and right to him. When a hungry philosopher has both his chopsticks at the same time, he eats without releasing his chopsticks. When he finishes eating, he puts down both his chopsticks and starts thinking again.

Here's a solution for the problem which does not require a process to write another process's state, and gets equivalent parallelism.

```
#define N 5                                     /* Number of philosphers */
#define RIGHT(i) (((i)+1) %N)
```

```
#define LEFT(i) (((i)==N) ? 0 : (i)+1)

typedef enum { THINKING, HUNGRY, EATING } phil_state;
phil_state state[N];
semaphore mutex =1;
semaphore s[N];

/* one per philosopher, all 0 */
void get_forks(int i) {
        state[i] = HUNGRY;
        while ( state[i] == HUNGRY ) {
                P(mutex);
                if ( state[i] == HUNGRY &&
                state[LEFT] != EATING &&
                state[RIGHT(i)] != EATING ) {
                state[i] = EATING;
                V(s[i]);
              }
        V(mutex);
        P(s[i]);
        }
 }

void put_forks(int i) {
        P(mutex);
        state[i]= THINKING;
        if ( state[LEFT(i)] == HUNGRY ) V(s[LEFT(i)]);
        if ( state[RIGHT(i)] == HUNGRY) V(s[RIGHT(i)]);
        V(mutex);
}
void philosopher(int process) {
        while(1) {
        think();
        get_forks(process);
        eat();
        put_forks();
        }
}
```

3.5.4 Sleeping Barber Problem

A barber shop consists of a waiting room with chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barber shop and all chairs are occupied, then the customer leaves the shop. if the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

The following is a sample solution for the sleeping barber problem.

```
# define CHAIRS 5              // chairs for waiting customers

typedef int semaphore;         // use this for imagination

semaphore customers = 0;       // number of customers waiting for service
semaphore barbers – 0;         // number of barbers waiting for customers
semaphore mutex = 1;           // for mutual exclusion
int waiting = 0;               //customers who are waiting for a haircut

void barber(void)
{
```

```
 while (TRUE) {
        down(&customers);       //go to sleep if no of customers are zero
        down(&mutex);           //acquire access to waiting
        waiting = waiting -1 ;  //decrement count of waiting customers
        up(&barbers);           //one barber is now ready for cut hair
        up(&mutex);             //release waiting
        cut_hair();             //this is out of critical region for hair cut
        }
}

void customer(void)
{
        down(&mutex);           //enter critical region
        if (waiting < CHAIRS)  //if there are no free chairs, leave
        {
        waiting = waiting +1;   //increment count of waiting customers
        up(&customers);         //wake up barber if necessary
        up(&mutex);             //release access to waiting
        down(&barbers);         //go to sleep if no of free barbers is zero
        get_haircut();          //be seated and be serviced
        } else
                {
                up (&mutex);    // shop is full: do no wait
                }
}
```

Explanation:

- This problem is similar to various queuing situations
- The problem is to program the barber and the customers without getting into race conditions
    - Solution uses three semaphores:
        - *customers*; counts the waiting customers
        - *barbers*; the number of barbers (0 or 1)
        - *mutex*; used for mutual exclusion
        - also need a variable *waiting*; also counts the waiting customers; (reason; no way to read the current value of semaphore)
    - The barber executes the procedure *barber*, causing him to block on the semaphore *customers* (initially 0);
    - The barber then goes to sleep;
    - When a customer arrives, he executes *customer*, starting by acquiring *mutex* to enter a critical region;
    - If another customer enters, shortly thereafter, the second one will not be able to do anything until the first one has released *mutex;*
    - The customer then checks to see if the number of waiting customers is less than the number of chairs;
    - If not, he releases *mutex* and leaves without a haircut;
    - If there is an available chair, the customer increments the integer variable, *waiting;*
    - Then he does an **up** on the semaphore *customers;*
    - When the customer releases *mutex*, the barber begins the haircut.

3.6 LOCKS

Locks are another synchronization mechanism. A lock has got two atomic operations (similar to semaphore) to provide mutual exclusion. These two operations are Acquire and Release. A process will acquire a lock before accessing a shared variable, and later it will be released. A process locking a variable will run the following code:

Lock-Acquire();

critical section

Lock-Release();

The difference between a lock and a semaphore is that a lock is released only by the process that have acquired it earlier. As we discussed above any process can increment the value of the semaphore. To implement locks, here are some things you should keep in mind:

- To make Acquire () and Release () atomic
- Build a wait mechanism.
- Making sure that only the process that acquires the lock will release the lock.

3.7 MONITORS AND CONDITION VARIABLES

When you are using semaphores and locks you must be very careful, because a simple misspelling may lead that the system ends up in a deadlock. Monitors are written to make synchronization easier and correctly. Monitors are some procedures, variables, and data structures grouped together in a package.

An early proposal for organising the operations required to establish mutual exclusion is the explicit critical section statement. In such a statement, usually proposed in the form “critical x do y”, where “x” is the name of a semaphore and “y” is a statement, the actual wait and signal operations used to ensure mutual exclusion were implicit and automatically balanced. This allowed the compiler to trivially check for the most obvious errors in concurrent programming, those where a wait or signal operation was accidentally forgotten. The problem with this statement is that it is not adequate for many critical sections.

A common observation about critical sections is that many of the procedures for manipulating shared abstract data types such as files have critical sections making up their entire bodies. Such abstract data types have come to be known as monitors, a term coined by C. A. R. Hoare. Hoare proposed a programming notation where the critical sections and semaphores implicit in the use of a monitor were all implicit. All that this notation requires is that the programmer encloses the declarations of the procedures and the representation of the data type in a monitor block; the compiler supplies the semaphores and the wait and signal operations that this implies. Using Hoare's suggested notation, shared counters might be implemented as shown below:

```
var value: integer;
procedure increment;
begin
        value := value + 1;
end { increment };
end { counter };

var i, j: counter;
```

Calls to procedures within the body of a monitor are done using record notation; thus, to increment one of the counters declared in above example, one would call “i.increment”. This call would implicitly do a wait operation on the semaphore implicitly associated with “i”, then execute the body of the “increment” procedure

before doing a signal operation on the semaphore. Note that the call to "i.increment" implicitly passes a specific instance of the monitor as a parameter to the "increment" procedure, and that fields of this instance become global variables to the body of the procedure, as if there was an implicit "with" statement.

There are a number of problems with monitors which have been ignored in the above example. For example, consider the problem of assigning a meaning to a call from within one monitor procedure to a procedure within another monitor. This can easily lead to a deadlock. For example, when procedures within two different monitors each calling the other. It has sometimes been proposed that such calls should never be allowed, but they are sometimes useful! We will study more on deadlocks in the next units of this course.

The most important problem with monitors is that of waiting for resources when they are not available. For example, consider implementing a queue monitor with internal procedures for the enqueue and dequeue operations. When the queue empties, a call to dequeue must wait, but this wait must not block further entries to the monitor through the enqueue procedure. In effect, there must be a way for a process to temporarily step outside of the monitor, releasing mutual exclusion while it waits for some other process to enter the monitor and do some needed action.

Hoare's suggested solution to this problem involves the introduction of condition variables which may be local to a monitor, along with the operations wait and signal. Essentially, if s is the monitor semaphore, and c is a semaphore representing a condition variable, "wait c" is equivalent to "signal(s); wait(c); wait(s)" and "signal c" is equivalent to "signal(c)". The details of Hoare's wait and signal operations were somewhat more complex than is shown here because the waiting process was given priority over other processes trying to enter the monitor, and condition variables had no memory; repeated signalling of a condition had no effect and signaling a condition on which no process was waiting had no effect. Following is an example monitor:

```
monitor synch
      integer i;
      condition c;
      procedure producer(x);
      .
      .
      end;
      procedure consumer(x);
      .
      .
      end;
   end monitor;
```

There is only one process that can enter a monitor, therefore every monitor has its own waiting list with process waiting to enter the monitor.

Let us see the dining philosopher's which was explained in the above section with semaphores, can be re-written using the monitors as:

Example: Solution to the Dining Philosophers Problem using Monitors

```
monitor dining-philosophers
{
  enum state {thinking, hungry, eating};
  state state[5];
  condition self[5];

  void pickup (int i)
  {
    state[i] = hungry;
    test(i);
    if (state[i] != eating)
```

```
    self[i].wait;
  }

    void putdown (int i)
  {
   state[i] = thinking;
   test(i+4 % 5);
   test(i+1 % 5);
  }

  void test (int k)
  {
   if ((state[k+4 % 5] != eating) && (state[k]==hungry)
                        && state[k+1 % 5] != eating))
   {
    state[k] = eating;
    self[k].signal;
   }
  }

  init
  {
   for (int i = 0; i< 5; i++)
    state[i] = thinking;
  }
 }
```

Condition Variables

If a process cannot enter the monitor it must block itself. This operation is provided by the condition variables. Like locks and semaphores, the condition has got a wait and a signal function. But it also has the broadcast signal. Implementation of condition variables is part of a synch.h; it is your job to implement it. Wait (), Signal () and Broadcast () have the following semantics:

- Wait() releases the lock, gives up the CPU until signaled and then re-acquire the lock.
- Signal() wakes up a thread if there are any waiting on the condition variable.
- Broadcast() wakes up all threads waiting on the condition.

When you implement the condition variable, you must have a queue for the processes waiting on the condition variable.

☞ Check Your Progress 1

1) What are race conditions? How race conditions occur in Operating Systems?

……

……

2) What is a critical section? Explain.

……

……

3) Provide the solution to a classical synchronization problem namely "cigarette smoker's problem". The problem is defined as follows:

There are four processes in this problem: three smoker processes and an agent process. Each of the smoker processes will make a cigarette and smoke it. To make a cigarette requires tobacco, paper, and matches. Each smoker process

has one of the three items. i.e., one process has tobacco, another has paper, and a third has matches. The agent has an infinite supply of all three. The agent places two of the three items on the table, and the smoker that has the third item makes the cigarette.

……

……

3.8 SUMMARY

Interprocess communication provides a mechanism to allow process to communicate with other processes. Interprocess communication system is best provided by a message passing system. Message systems can be defined in many different ways. If there are a collection of cooperating sequential processes that share some data, mutual exclusion must be provided. There are different methods for achieving the mutual exclusion. Different algorithms are available for solving the critical section problem which we have discussion in this unit. The bakery algorithm is used for solving the *n* process critical section problem.

Interprocess synchronization provides the processes to synchronize their activities. Semaphores can be used to solve synchronization problems. Semaphore can only be accessed through two atomic operations and can be implemented efficiently. The two operations are *wait* and *signal*.

There are a number of classical synchronization problems which we have discussed in this unit (such as producer- consumer problem, readers – writers problem and d dining – philosophers problem). These problems are important mainly because they are examples for a large class of concurrency-control problems. In the next unit we will study an important aspect called as “Deadlocks”

3.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Processes that are working together share some common storage (main memory, file etc.) that each process can read and write. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Concurrently executing threads that share data need to synchronize their operations and processing in order to avoid race condition on shared data. Only one “customer” thread at a time should be allowed to examine and update the shared variable.

   Race conditions are also possible in Operating Systems. If the ready queue is implemented as a linked list and if the ready queue is being manipulated during the handling of an interrupt, then interrupts must be disabled to prevent another interrupt before the first one completes. If interrupts are not disabled than the linked list could become corrupt.

2) The most synchronization problem confronting cooperating processes, is to control the access between the shared resources. Suppose two processes share access to a file and at least one of these processes can modify the data in this shared area of memory. That part of the code of each program, where one process is reading from or writing to a shared memory area, is a *critical section* of code; because we must ensure that only one process execute a critical section of code at a time. The critical section problem is to design a protocol that the processes can use to coordinate their activities when one wants to enter its critical section of code.

3) This seems like a fairly easy solution. The three smoker processes will make a cigarette and smoke it. If they can’t make a cigarette, then they will go to sleep.

The agent process will place two items on the table, and wake up the appropriate smoker, and then go to sleep. All semaphores except lock are initialised to 0. Lock is initialised to 1, and is a mutex variable.

Here's the code for the ***agent process***.

```
do forever {
    P( lock );
    randNum = rand(1,3); // Pick a random number from 1-3
    if (randNum == 1) {
            // Put tobacco on table
            // Put paper on table
            V(smoker_match);  // Wake up smoker with match
            }
    else if (randNum == 2) {
            // Put tobacco on table
            // Put match on table
    V(smoker_paper);  // Wake up smoker with paper
            }
            else {
                    // Put match on table
                    // Put paper on table
                    V(smoker_tobacco);
            } // Wake up smoker with tobacco

    V(lock);
    P(agent);  //  Agent sleeps

    }  // end forever loop
```

The following is the code for one of the smokers. The others are analogous.

```
do forever {
        P(smoker_tobacco);  // Sleep right away
        P(lock);
        // Pick up match
        // Pick up paper
        V(agent);
        V(lock);
        // Smoke
        }
```

3.10 FURTHER READINGS

1) Milenkovic, Milan, "*Operating Systems Concepts and Design"*, McGraw-Hill, 2nd Edition.

2) Tanenbaum, Andrew, *Modern Operating Systems*, Prentice-Hall.

3) Silberschatz, Abraham and Galvin Peter, "*Operating System Concepts"*, Addison-Wesley.

4) D.M. Dhamdhere, "*Operating Systems A concept-based Approach"*, Tata McGraw-Hill.

5) William Stalling, "*Operating System*", Prentice Hall.

6) Deitel, Harvey M. , "*An introduction to Operating System*", 4th ed., Addison-Wesley.