Unit 5 Process Synchronization

Structure:

5.1 Introduction
Objectives
5.2 Interprocess Communication
5.3 The Critical-Section Problem
5.4 Semaphores
5.5 Monitors
5.6 Hardware Assistance
5.7 Summary
5.8 Terminal Questions
5.9 Answers

5.1 Introduction

In the last unit you have studied about CPU scheduling algorithms. In this unit you are going to learn about process synchronization.

A co-operating process is one that can affect or be affected by the other processes executing in the system. These processes may either directly share a logical address space (that is, both code and data), or be allowed to share data only through files. Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanism to ensure the orderly execution of co-operating processes. In this unit we will discuss how co-operating processes can communicate and various mechanisms to ensure the orderly execution of co-operating processes that share a logical address space, so that data consistency is maintained.

Objectives:

After studying this unit, you should be able to:

- explain Interprocess communication
- discuss the critical section problem
- describe semaphores and monitors
- explain the significance of hardware assistance

5.2 Interprocess Communication

Communication of co-operating processes' shared-memory environment requires that these processes share a common buffer pool, and that the code for implementing the buffer be explicitly written by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for co-operating processes to communicate with each other via an inter-process-communication (IPC) facility.

IPC provides a mechanism to allow processes to communicate and to synchronize their actions. Inter-process-communication is best provided by a message system. Message systems can be defined in many different ways. Message-passing systems also have other advantages. Note that the shared-memory and message system communication schemes are not mutually exclusive, and could be used simultaneously within a single operating system or even a single process.

Basic Structure

The function of a message system is to allow processes to communicate with each other without the need to resort to shared variables. An IPC facility provides at least the two operations: send (message) and receive (message).

Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the physical implementation is straight forward. This restriction, however, makes the task of programming more difficult. On the other hand, variable-sized messages require a more complex physical implementation, but the programming task becomes simpler.

If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network), but rather with the issues of its logical implementation, such as its logical properties. Some basic implementation questions are as follows:

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of processes?

- What is the capacity of a link? That is, does the link have some buffer space? If it does, how much?
- What is the size of messages? Can the link accommodate variable-sized or only fixed-sized messages?
- Is a link unidirectional or bi-directional? That is, if a link exists between P and Q, can messages flow in only one direction (such as only from P to Q) or in both directions?

The definition of unidirectional must be stated more carefully, since a link may be associated with more than two processes. Thus, we say that a link is unidirectional only if each process connected to the link can either send or receive, but not both and each link has at least one receiver process connected to it.

In addition, there are several methods for logically implementing a link and the send/receive operations:

- Direct or indirect communication
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed-sized or variable-sized messages

For the remainder of this section, we elaborate on some of these types of message systems.

Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct communication or indirect communication, as we shall discuss under the next two subheadings.

Direct Communication:

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

Send (P, message). Send a message to process P.

Receive (Q, message). Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.
- The link may be unidirectional, but is usually bi-directional.

To illustrate, let us present a solution to the producer-consumer problem. To allow the producer and consumer processes to run concurrently, we allow the producer to produce one item while the consumer is consuming another item. When the producer finishes generating an item, it sends that item to the consumer. The consumer gets that item via the receive operation. If an item has not been produced yet, the consumer process must wait until an item is produced. The producer process is defined as:

```
repeat
.....
        produce an item in nextp
        .....
        send ( consumer, nextp );
until false;

The consumer process is defined as
repeat
        receive ( producer ,nextc) ;
        ......
        consume the item in nextc
        .....
until  false;
```

This scheme exhibits symmetry in addressing; that is, both the sender and the receiver processes have to name each other to communicate. A variant of this scheme employs asymmetry in addressing. Only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send and receive primitives are defined as follows:

- **send** (p, message). Send a message to process P.
- **receive** (id, message). Receive a message from any process; the variable *id* is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the name of a process may necessitate examining all other process definitions. All references to the old name must be found, so that they can be modified to the new name. This situation is not desirable from the viewpoint of separate compilation.

Indirect Communication:

With indirect communication, the messages are sent to and received from mailboxes (also referred to as ports). A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if the processes have a shared mailbox. The send and receive primitives are defined as follows:

send (A, message). Send a message to mailbox A.

receive (A, message). Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if they have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, each link corresponding to one mailbox.
- A link may be either unidirectional or bi-directional.

Now suppose that processes P_1, P_2, and P_3 all share mailbox A. Process P_1 sends a message to A, while P_2 and P_3 each execute a receive from A. Which process will receive the message sent by P_1? This question can be resolved in a variety of ways:

- Allow a link to be associated with at most two processes.
- Allow at most one process at a time to execute a receive operation.

- Allow the system to select arbitrarily which process will receive the message (that is, either P_2 or P_3, but not both, will receive the message). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the system. If the mailbox is owned by a process (that is, the mailbox is attached to or defined as part of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user of the mailbox (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists (via exception handling, described in Section 5.6 of this unit).

There are various ways to designate the owner and users of a particular mailbox. One possibility is to allow a process to declare variables of type mailbox. The process that declares a mailbox is that mailbox's owner. Any other process that knows the name of this mailbox can use this mailbox.

On the other hand, a mailbox that is owned by the operating system has an existence of its own. It is independent, and is not attached to any particular process. The operating system provides a mechanism that allows a process:

- To create a new mailbox
- To send and receive messages through the mailbox
- To destroy a mailbox

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receive privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox. Processes may also share a mailbox through the process-creation facility. For example, if process P created mailbox A, and then created a new process Q, P and Q may share mailbox A. Since all processes with access rights to a mailbox may ultimately terminate, after some time a mailbox may no longer be accessible by any process. In this case, the operating system should reclaim whatever space was used for the mailbox. This task may require some form of garbage collection, in which a separate operation occurs to search for and de-allocate memory that is no longer in use.

Buffering

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

- **Zero capacity:** The queue has maximum length 0; thus, the link cannot have any messages waiting in it. In this case, the sender must wait until the recipient receives the message. The two processes must be synchronized for a message transfer to take place. This synchronization is called a rendezvous.
- **Bounded capacity:** The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the latter is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. However, the link has a finite capacity. If the link is full, the sender m_1, must be delayed until space is available in the queue.
- **Unbounded capacity:** The queue has potentially infinite length; thus, any number of messages can wait in it. The sender is never delayed.

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases provide automatic buffering.

We note that, in the non-zero-capacity cases, a process does not know whether a message has arrived at its destination after the send operation is completed. If this information is crucial for the computation, the sender must communicate explicitly with the receiver to find out whether the latter has received the message. For example, suppose process P sends a message to process Q and can continue its execution only after the message is received. Process P executes the sequence.

send (Q, message);

receive (Q, message);

Process Q executes

receive (P, message);

send (P, "acknowledgment");

Such processes are said to communicate asynchronously.

There are special cases that do not fit directly into any of the categories that we have discussed:

- The process sending a message is never delayed. However, if the receiver has not received the message before the sending process sends another message, the first message is lost. The advantage of this scheme is that large messages do not need to be copied more than once. The main disadvantage is that the programming task becomes more difficult. Processes need to synchronize explicitly, to ensure both that messages are not lost and that the sender and receiver do not manipulate the message buffer simultaneously.
- The process of sending a message is delayed until it receives a reply. This scheme was adopted in the Tooth operating system. In this system messages are of fixed size (eight words). A process P that sends a message is blocked until the receiving process has received the message and has sent back an eight-word reply by the reply (P, message) primitive. The reply message overwrites the original message buffer. The only difference between the send and reply primitives is that a send causes the sending process to be blocked, whereas the reply allows both the sending process and the receiving process to continue with their executions immediately.

This synchronous communication method can be expanded easily into full-featured remote procedure call (RPC) system. An RPC system is based on the realization that a sub-routine or procedure call in a single-process system acts exactly like a message system in which the sender blocks until it receives a reply. The message is then like a subroutine call and the return message contains the value of the sub-routine computed. The next logic step, therefore, is for concurrent processes to be able to call each other sub-routines using RPC. RPCs can be used between processes running on separate computers to allow multiple computers to work together in a mutually beneficial way.

Self Assessment Questions

1. Inter Process Communication provides a mechanism to allow processes to communicate and to synchronize their actions. (True / False)

2. RPC stands for _______________.
3. The _______________ queue has finite length n; thus, at most n messages can reside in it. (Pick right option)
   a) Bounded Capacity
   b) Zero Capacity
   c) Unbounded Capacity
   d) None of the above

5.3 The Critical-Section problem

A **critical-section** is a part of program that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one process of execution. Consider a system consisting of n processes {p0, p1 …pn-1}. Each process has a segment of code called a critical-section. The important feature of the system is that, when one process is executing in its critical-section, no other process is to be allowed to execute in its critical-section. Thus the execution of critical-sections by the processes is mutually exclusive in time. The critical-section problem is to design a protocol that the processes can use to co-operate. Each process must request permission to enter its critical-section. The section of the code implementing this request is the entry section. The critical-section may be followed by an exit section. The remaining code is the remainder section.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual Exclusion: If process p_i is executing in its critical-section, then no other processes can be executing in their critical-sections.

2. Progress: If no process is executing in its critical-section and there exist some processes that are not executing in their remainder section can participate in the decision of which will enter in its critical-section next, and this selection cannot be postponed indefinitely.

3. Bounded Waiting: There exist a bound on the number of times that other processes are allowed to enter their critical-sections after a process has made a request to enter its critical-section and before that request is granted. When presenting an algorithm, we define only the variables used for synchronization purposes, and describe only a typical process p_i whose general structure is shown in Figure 5.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

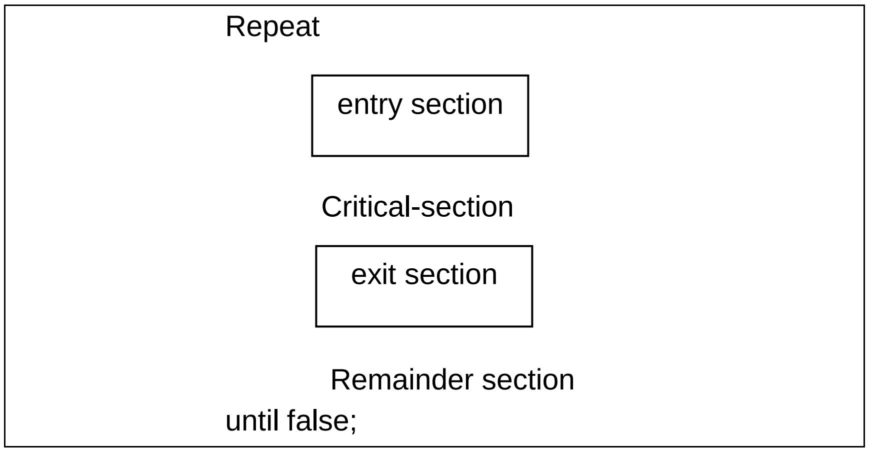

Fig. 5.1: General structure of a typical process p_i

Two-process solutions

The following two algorithms are applicable to only two processes at a time. The processes are numbered p0 and p1. For convenience, when presenting pi, we use pj to denote the other process; that is j=1-i.

Algorithm 1:

Let the processes share a common variable *turn* initialized to 0 (or 1). If *turn* =i, then process pi is allowed to execute in its critical-section. The structure of process pi is shown in Figure 5.2.

This solution ensures that only one process at a time can be in its critical-section. But it does not satisfy the progress requirement, since it requires strict alternation of processes in the execution of the critical-section. For example, if turn=0 and p1 is ready to enter its critical-section, p1 cannot do so, even though p0 may be in its remainder section.

```
Repeat
        While turn ≠ i do no-op;
            Critical-section
                turn= j;
            Remainder section
until false;
```

Fig. 5.2: The structure of process pi in algorithm 1

Algorithm 2:

The problem with algorithm 1 is that it does not retain sufficient information about the state of each process; it remembers only which process is allowed to enter that process critical-section. To remedy this problem, we can replace the variable *turn* with the following array:

var flag: array[0..1] of Boolean;

the elements of the array are initialized to false. If flag[i] is true, this value indicates that pi is ready to enter the critical-section. The structure of process pi is shown in Figure 5.3.

In this algorithm process pi first sets flag[i] to be true, signaling that it is ready to enter its critical-section. Then pi checks to verify that process pj is not also ready to enter its critical-section. If pj were ready, then pi would wait until pj had indicated that it no longer needed to be in the critical-section. At this point, pi would enter the critical-section. On exiting the critical section, pi would set its flag to be false, allowing the other process to enter its critical-section.

In this solution, the mutual exclusion requirement is satisfied. Unfortunately, the progress requirement is not met.

```
Repeat
        flag[i]=true;
        while flag[j] do no-op;

            Critical-section

                flag[i]= false;

        Remainder section

until false;
```

Fig. 5.3: The structure of process pi in algorithm 2

Algorithm 3:

By combining the key ideas of algorithm 1 and algorithm 2, we obtain a correct solution to the critical-section problem, where all three requirements are met. The processes share two variables;

var flag: array[0…1] of boolean;
 turn; 0…1;

initially flag[0]=flag[1]=false, and the value of turn is immaterial (but it is either 0 or 1). The structure of process pi is shown in figure 5.4. The eventual value of turn decides which of the two processes is allowed to enter its critical-section first.

```
Repeat
        flag[i]=true;
        turn=j;
        while (flag[j] and turn=j) do no-op;

            Critical-section

                flag[i]= false;

            Remainder section
            until false;
```

Fig. 5.4: The structure of process pi in algorithm 3

This algorithm satisfies the following three properties.

1) Mutual exclusion
2) The progress requirement
3) The bounded- waiting requirement

To prove property 1, we observe that each pi enters its critical-section only if flag[j]=false or turn=i. Also note that if both processes are executing in their critical-sections at the same time, then flag[0]=flag[1]=true and value of turn can be either 0 or 1, but not both, hence only one of the two can be executing in their critical-section.

To prove property 2 and 3, we note that a process pi can be prevented from entering the critical-section only if it stuck in the while loop with the condition flag[j]=true and turn=j; this loop is the only one. If pj is not ready to enter the critical-section, then flag[j]=false and pi can enter its critical-section. If pj has set flag[j]=true and is also executing in its while statement, then either turn=I or turn=j. if turn=i, then pi will enter the critical-section. If turn=j, then pj will enter the critical-section. Since pi does not change the value of the variable turn while executing the while statement, pi will enter the critical-section (progress) after at the most one entry by pj (bounded waiting).

Multiple-process solutions

An algorithm developed for solving the critical-section problem for n processes is also called Bakery algorithm, because it is based on the scheduling algorithm commonly used in bakeries, ice-cream stores, meat markets, motor-vehicle registries and other locations where order must be made out of chaos.

In this algorithm each process, receives a number. Unfortunately, the bakery algorithm cannot guarantee that two processes do not receive the same number. In such a case (if two processes receive same number) the process with the lowest name is served first. Since the process name is unique and totally ordered, the algorithm is completely deterministic.

The common data structures used are

```
var choosing : array [0..n-1] of Boolean;
      number: array [0..n-1] of integer;
```

initially these data structures are initialized to false and 0 respectively . For convenience, we define the following notation. The structure of process pi, used in the bakery algorithm, is shown in Figure 5.5.

Consider a process pi in its critical section and pk trying to enter the its critical section. When process pk executes the second while statement for j=I, it finds that

- number[i] ≠ 0
- (number[i],i) < (number[k],k)

Thus it continues looping in the while statement until pi leaves the pi critical section.

```
Repeat
    choosing[i]=true;
    number[i]=max(number[0],number[1],….number[n-1])+1;
    choosing[i]=false;
    for j=0 to n-1
    do begin
      while choosing[j] do no-op;
      while number[j]  ≠ 0
    and (number[j],j) <(number[i],i) do no-op;
    end;

        critical section

        number[i]=0;

        remainder section

  until false;
```

Fig. 5.5: The structure of process pi in the bakery algorithm.

Self Assessment Questions

4. A critical-section is a part of program that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one process of execution. (True / False)
5. An algorithm developed for solving the critical-section problem for n processes is also called _______________ algorithm.

6. If process p_i is executing in its critical-section, then no other processes can be executing in their critical-sections. This is called _______________.
   a) Bounded Waiting
   b) Mutual Exclusion
   c) Progress
   d) None of the above

5.4 Semaphores

Semaphores are the classic method for restricting access to shared resources (e.g. storage) in a multi-processing environment. They were invented by Dijkstra and first used in T.H.E operating system.

A semaphore is a protected variable (or abstract data type) which can only be accessed using the following operations:

```
P(s)
        Semaphore s;
        {
          while (s == 0) ;/* wait until s>0 */
          s = s-1;
        }

V(s)
        Semaphore s;
        {
          s = s+1;
        Init(s, v)
        Semaphore s;
        Int v;
        {
          s = v;
            }
```

The P operation busy-waits (or may be sleeps) until a resource is available whereupon it immediately claims one. V is the inverse; it simply makes a resource available again after the process has finished using it. Init is only used to initialize the semaphore before any requests are made. The P and V operations must be indivisible, i.e. no other process can access the semaphore during their execution.

To avoid busy-waiting, a semaphore may have an associated queue of processes (usually a FIFO). If a process does a P on a semaphore which is zero, the process is added to the semaphore's queue. When another process increments the semaphore by doing a V and there are tasks on the queue, one is taken off and resumed.

Semaphores are not provided by hardware. But they have several attractive properties:

- Machine independent.
- Simple.
- Powerful. Embody both exclusion and waiting.
- Correctness is easy to determine.
- Work with many processes.
- Can have many different critical sections with different semaphores.
- Can acquire many resources simultaneously (multiple P's).
- Can permit multiple processes into the critical section at once, if that is desirable.

5.5 Monitors

Another high-level synchronization construct is the monitor type. A monitor is a collection of procedures, variables and data structures grouped together. Processes can call the monitor procedures but cannot access the internal data structures. Only one process at a time may be **active** in a monitor.

The syntax of monitor looks like:

```
Type monitor-name= monitor
Variable declarations
procedure entry P1 (….);
begin….  end;
.
.
.
procedure entry P2 (….);
begin….  end;
.
procedure entry Pn (….);
begin….  end;
begin
initialization code
end
```

A procedure defined within a monitor can access only those variables declared locally within the monitor and the formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.

5.6 Hardware Assistance

Checking for mutual exclusion is also possible through hardware. Special instructions called Test and Set Lock (TSL) is used for the purpose. An important feature is that the set of instructions used for this purpose is indivisible, that is, they cannot be interrupted during execution. The instruction has the format 'TSL ACC, IND' where ACC is an accumulator register and IND is a memory location used as a flag. Whenever the instruction is executed, contents of IND are copied to ACC and IND is set to 'N' which implies that a process is in its critical section. If the process comes out of its critical section IND is set to 'F'. Hence the TSL instruction is executed only if IND has a value 'F' as shown below:

```
Begin
        - - - - - - -
        Call Enter-critical-section
        Critical section instructions
        Call Exit-critical-section
        - - - - - - -
```

Call Enter-critical-section executes the TSL instruction if IND = F else waits. Call Exit-critical-section sets IND = F. Any process producer / consumer can be handled using the above algorithm. Demerits of the algorithm include the use of special hardware that restricts portability.

Self Assessment Questions

7. Semaphores are the classic method for restricting access to shared resources (e.g. storage) in a multi-processing environment. (True / False)
8. Semaphores were invented by _______________.
9. _______________is a high-level synchronization construct which is a collection of procedures, variables and data structures grouped together. (Pick right option)
   a) Semaphores
   b) Processes
   c) Monitor
   d) None of the above

5.7 Summary

Let's summarize the key points covered in the unit:

- Given a collection of co-operating processes that share data, mutual exclusion should be provided.
- Mutual exclusion is a way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing. One solution for this is by ensuring that a critical section of code is in use by only one process or thread at a time.
- There are different algorithms to solve the critical-section problem. But the main disadvantage of these algorithms is that they all need busy waiting.

- Busy waiting is a technique in which a process repeatedly checks to see if a condition is true. It will delay execution for some amount of time.
- To overcome this problem we use semaphores, which is a mechanism that prevents two or more processes from accessing a shared resource simultaneously.
- Another way to achieve mutual exclusion is to use monitors which are characterized by a set of programmer defined operators.

5.8 Terminal Questions

1. Discuss Interprocess Communication.
2. What is critical-section problem? Explain.
3. What are semaphores? Explain.
4. What are monitors? Explain.

5.9 Answers

Self Assessment Questions

1. True
2. Remote Procedure Call
3. a) Bounded Capacity
4. True
5. Bakery
6. b) Mutual Exclusion
7. True
8. Dijkstra
9. c) Monitor

Terminal Questions

1. Inter Process Communication provides a mechanism to allow processes to communicate and to synchronize their actions. Inter-process-communication is best provided by a message system. (Refer Section 5.2)
2. A critical-section is a part of program that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one process of execution. (Refer Section 5.3)

3. Semaphores are the classic method for restricting access to shared resources (e.g. storage) in a multi-processing environment. They were invented by Dijkstra and first used in T.H.E operating system. (Refer Sections 5.4)
4. A monitor is a collection of procedures, variables and data structures grouped together. Processes can call the monitor procedures but cannot access the internal data structures. Only one process at a time may be active in a monitor. (Refer Section 5.5)