CHAPTER 2
MUTUAL EXCLUSION AND SYNCHRONISATION

2.1 INTRODUCTION

Mutual exclusion is one of the first problems encountered in designing distributed operating systems. Operations on shared data structures from the processes of a distributed system which must be performed atomically are termed as critical section. Mutual exclusion is a property which ensures that only one process is in the critical section at any given instant. Several solutions have been proposed and implemented to ensure mutual exclusion. A popular cross-section of such solutions are discussed in this chapter.

2.2 PROGRAMMING LEVEL SOLUTIONS

Some of the popular algorithms are mentioned below (Raynal 1986).

* Dekker's Algorithm for two processes

* Dijkstra's Algorithm for n processes: May lead to starvation i.e., a process may wait indefinitely to gain access (Dijkstra 1965).

* Knuth's Algorithm: Maximum waiting time for any process is $2^{n-1} - 1$ turns (Knuth 1966).

* De Bruijn's Algorithm: Maximum waiting time for any process is $n(n-1)/2$ turns (De Bruijn 1967).
* Eisenberg and MacGuire's Algorithm: Maximum waiting time for any process is $n-1$ turns (Eisenberg 1972).

* Peterson's Algorithm: Peterson presented a simple algorithm for $n$ processes in 1981. The maximum waiting time for any process is $\frac{n(n-1)}{2}$ turns (Peterson 1981).

In 1980 Burns and Lynch proved that every algorithm for mutual exclusion avoiding deadlock must use at least 'n' binary shared variables. Later in 1981 Burns presented an algorithm with a minimum number of variables and a minimum number of values assigned to these variables (Raynal 1986).

In all the above algorithms the common feature was the use of global variables that are shared by all the processes. A different approach is to avoid using global variables and restrict the processes only to the use of local variables or specific variables. Local variables of a process are those variables which can be accessed only by that process and specific variables are those that can only be read by all other processes. This approach is often referred to as state variable approach. The first algorithm using this method was presented by Lamport in 1974 and is popularly known as the Bakery Algorithm.

2.3 HARDWARE SOLUTIONS

In addition to the software solutions discussed above several hardware solutions have been proposed and implemented to achieve mutual exclusion in the critical region. Some of the popular hardware solutions are discussed below (Raynal 1986).

a) Uniprocessor Systems

Concurrent processes running on a uniprocessor system have to ensure that they are not interrupted during the execution of the critical section. This can be achieved by masking the interrupts before entering the critical section and unmasking them after completion of the critical region. The mutual exclusion protocol for any process is given below.

<program segment>
mask
<CRITICAL SECTION>
unmask
<program section>

The mask and unmask instructions are normally provided in the instruction set of many machines.

b) Multiprocessor Systems

To ensure mutual exclusion in multiprocessor systems with centralised control, special instructions have been incorporated into the instruction set. All these solutions are based on two fundamental concepts.
i) Once access is granted to any memory location, no other access is possible for the same word.

ii) Within one instruction fetch cycle two actions are performed namely ‘reading and writing’ or ‘reading and testing’.

Any one of the following instructions may be provided to ensure mutual exclusion in the critical region.

* Swap Instruction

* Test and Set Instruction

* Lock Instruction

* Increment Instruction

* Replace-add Instruction

All the above hardware solutions have active wait as their major disadvantage. Moreover, it is difficult to generalise these solutions for complex problems.

To circumvent these difficulties Dijkstra proposed the **Semaphore Concept**. Semaphores are part of the instruction set for several machines and are also used as synchronisation tools.
2.3.1 Semaphores

A semaphore is a shared data entity comprising of two parts (Peterson 1983a)

a) an integer counter to count the number of requests received for the critical section.

b) a queue of processes waiting for access to the critical section.

A data object $S$ acting as a semaphore has two atomic operations defined on it ($S.\text{counter}$ is the counter variable in the semaphore $S$)

i) $P(S)$ : $S.\text{counter} := S.\text{counter} - 1$
   
   if $S.\text{counter} < 0$
   
   then begin
   
   add this process to the queue block;
   
   end;

ii) $V(S)$ : $S.\text{counter} := S.\text{counter} + 1$
   
   if $S.\text{counter} \leq 0$
   
   then begin
   
   remove a process $P$ from queue wakeup($P$);
   
   end;

The block operation suspends the execution of the process invoking it and the wakeup($P$) operation makes the process $P$ to resume execution. The active wait time is greatly reduced.
The use of semaphores does not guarantee a deadlock free implementation. Semaphores gained a great deal of popularity and have been implemented directly at the operating system level.

2.4 LANGUAGE LEVEL SOLUTIONS

Brinch Hansen and Hoare introduced a language construct to solve the critical region problem at the user program level (Peterson 1983a).

A shared variable of a user-defined type \( T \) may be declared as

\[
\text{var } v : \text{shared } T;
\]

The variable \( v \) may be used only inside a region statement of the form

\[
\text{region } v \text{ do } S;
\]

This implies that while the statement sequence \( S \) is being executed no other process is given access to the shared variable. The number of time dependent errors that may occur is now reduced owing to the inherent sequential constraint on this construct. The compiler generates a semaphore for the variable \( v \) and the following code is produced

\[
P(v_{\text{sem}}) \\
S \\
V(v_{\text{sem}})
\]

The region statement may also be nested. The power of the region construct was further enhanced by the concept of conditional critical region which is of the form

\[
\text{region } v \text{ when } C \text{ do } S;
\]
If the condition C is false the process relinquishes its access to variable v without executing the statement sequence S and is delayed until the condition is true. It is to be noted that the region constructs are limited to shared variables.

2.4.1 Monitors

Several process being executed concurrently demand an operating system that permits these processes to share the available resources despite unpredictable demands. The problem is trivial if we have an unlimited number of each resource at our disposal. However, in practice the available resources are limited and it is the task of the operating system designer to construct efficient and reliable schedulers for various resources. There are several algorithms discussed earlier to cater to the problems posed by shared variables and resources. However, all of them require an explicit control strategy to be programmed for synchronisation. Instead, if the programming language can provide with constructs for synchronisation then the problem of critical region stands solved right at the user program level. Brinch Hansen’s monitor is a synchronisation technique that can be directly adopted into programming languages to solve the problems posed by shared variables and shared resources (Hoare 1974), (Pratt 1983).

Monitor is a collection of associated data procedures and synchronisation mechanisms. A calling procedure has to explicitly specify the monitor name along with the procedure call. A monitor may also be specified as a user defined type thus facilitating the use of multiple resources of the same kind. The synchronising mechanisms that are usually provided are semaphores. The code of the monitor can be accessed only by one process at any given time. This automatically ensures mutual exclusion.

2.4.2 Tasks

Another technique of implementing concurrent programs is to use Task as the basic unit of concurrency. This philosophy is adopted in the design of the programming
language Ada to be discussed later in this section. A program A may initiate a task B and once this initiation is done both A and B run concurrently. A task is dependent on the task that has initiated it. Before termination, a task waits until all the tasks initiated by it have terminated. The declaration of a task consists of two distinct parts (Barnes 1989)

a) **Task Definition**: It consists of special declarations which allow synchronisation and communication with other tasks running concurrently.

b) **Task Body**: Structurally it is similar to any normal subprogram. It consists of local variables, procedures and a sequence of statements.

The declaration of the task is included in a larger program. The tasks may automatically be initiated when the control enters the main program or they may be initiated using explicit call statements. The fork-join constructs and the concurrent statement can also be used to initiate tasks.

In many real life problems, several instances of the same task need to be running concurrently. For example, use of several terminals etc. To permit this facility, declaration of tasks as user defined types is often provided. Several tasks executing concurrently are essentially asynchronous i.e, each task has a different performing speed oblivious of the other tasks. Some of the popular techniques for synchronisation are discussed below (Pratt 1983).

2.4.2.1 Interrupts

A task X interrupts a task Y to signal the occurrence of a particular event. An interrupt handler routine within the task Y takes care of the consequences of the interrupt. On completion of the interrupt service task Y resumes execution from where it has left.
The disadvantages of this technique are

a) Separate interrupt handlers are required

b) The task waiting for an interrupt does so in an unending do nothing loop

c) Shared data objects between the task body and the interrupt handling
   segment warrant special ways of protection.

2.4.2.2 Semaphores

A detailed discussion on semaphores is presented earlier. Though this
technique is simple to implement there are certain inherent disadvantages such as

a) improper coding may lead to deadlock

b) programs involving several tasks and semaphores are difficult to debug
   and understand

2.4.2.3 Guarded Commands

A guarded command is a statement level control that can be used as a part of
a variety of synchronisation techniques.

Guarded Command
when <C1> => S1
when <C2> => S2   C1 .. Cn : Conditions
when <C3> => S3   S1 .. Sn : Statements
......
when \(<C_n> \Rightarrow S_n\) 
else 
S 
end Guarded Command

The conditions or guards are evaluated and from those that return true, one corresponding statement is arbitrarily chosen and executed. If none of them return true then the statement in the else clause is executed.

2.4.2.4 Rendezvous

The act of synchronisation between any two concurrent tasks for a brief period is termed as rendezvous in Ada. The caller task is called client and the called task is called server. A rendezvous point in a task is termed as an entry and the receiving task executes an accept statement.

\[
\text{accept } \text{<entry variable> do} \\
\text{---- statements} \\
\text{end}
\]

The sending task executes an entry call. Several accept statements may be clubbed in a select command thus mimicking a guarded command either with or without explicit guards. When more than a single accept command is ready for execution the selection is arbitrarily made. Execution of the client is suspended until the server reaches the specified accept statement. The execution of the server is suspended upon reaching an accept statement until a call is initiated for that accept statement. This technique is more
or less similar to the **Remote Procedure Call** mechanism. A server cannot initiate a call to itself.

As discussed earlier, scheduling a shared resource among a family of concurrently executing tasks warrants mutual exclusion in the code segment of the shared resource. The implementation of mutual exclusion is not so straightforward in this case as the called task is unaware of the caller to register the request and release the resource. A user task may place a request together with an unforgettable identification number. The resource may be released only by the execution of the accept command corresponding to the specified identification number.

All the tasks declared as an array are initialised as soon as the control enters the program segment in which they are declared. If each of the concurrent tasks requires a unique initialising information this can be supplied only through entry calls to each task. The execution of this mandatory rendezvous at the time of initialising may prove to be costly while using a large array of concurrent tasks (Jim 1981).

### 2.5 COMPARISON OF LANGUAGE LEVEL PRIMITIVES

The three primitives of concurrent programming are processes, tasks and monitors. Processes and tasks are by definition basic units of concurrency providing two distinct implementation strategies for concurrent programming. Monitor is a programming entity introduced to solve the critical region problem at the programming language level. Thus, processes and tasks signal the dynamic qualities of the concurrent system whereas monitors represent the static portion of the system. It is to be noted that tasks are also employed to solve the critical region problems thus giving a new perspective to the concept of tasks. The primitives are compared in the Table 2.1 and Table 2.2 (Gopal 1990b), (Judy 1986), (Wegner 1983).
<table>
<thead>
<tr>
<th>Table 2.1: Processes Vs. Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROCESSES</strong></td>
</tr>
<tr>
<td>A process structurally</td>
</tr>
<tr>
<td>resembles a sub-program.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>The process name, parameters</td>
</tr>
<tr>
<td>and statements have to be</td>
</tr>
<tr>
<td>together.</td>
</tr>
<tr>
<td>Process are initiated by</td>
</tr>
<tr>
<td>a) The Fork-Join Construct</td>
</tr>
<tr>
<td>b) The Concurrent Statement</td>
</tr>
<tr>
<td>c) Explicit initialising</td>
</tr>
<tr>
<td>statements</td>
</tr>
<tr>
<td>All the parallel processes</td>
</tr>
<tr>
<td>have to be known at the</td>
</tr>
<tr>
<td>compile time.</td>
</tr>
<tr>
<td>Communication between</td>
</tr>
<tr>
<td>processes occurs in two steps</td>
</tr>
<tr>
<td>a) Synchronisation.</td>
</tr>
<tr>
<td>b) Transmit the message</td>
</tr>
<tr>
<td>The call specifies the mode of message passing send/receive</td>
</tr>
<tr>
<td>d) Transmit o/p parameters</td>
</tr>
<tr>
<td>The calling task does not specify the send/receive mode of communication</td>
</tr>
</tbody>
</table>
Table 2.1: Processes Vs. Tasks (Contd.)

<table>
<thead>
<tr>
<th>PROCESSES</th>
<th>TASKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>The non-determinism is introduced by guarded commands that may spawn more parallel processes.</td>
<td>The non-determinism is introduced by conditional entry calls and select statements with boolean guards.</td>
</tr>
<tr>
<td>As the called process knows its user mutual exclusion in the critical region is easy to implement.</td>
<td>As the called task is oblivious of its user the mutual exclusion in the critical region is difficult to accomplish.</td>
</tr>
<tr>
<td></td>
<td>Elimination of this serialisation bottleneck is a key factor in highly parallel architectures.</td>
</tr>
</tbody>
</table>

Table 2.2: Tasks Vs. Monitors

<table>
<thead>
<tr>
<th>TASKS</th>
<th>MONITORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks represent both the dynamic and static aspects of concurrent programs.</td>
<td>Monitors are the static aspects of concurrent programs.</td>
</tr>
<tr>
<td>A task has a definition and a body that can be compiled separately. The task entries are user callable. An entry call may be executed only when the control expects it. Upon completion the subsequent statements of the called task are executed.</td>
<td>A monitor has local data, initialising statements and a set of user callable procedures all compiled together. Completion of a monitor call returns the monitor to the initial state.</td>
</tr>
<tr>
<td>Waiting entry calls are placed on different queues for each entry.</td>
<td>Waiting procedure calls are all placed on the same queue.</td>
</tr>
</tbody>
</table>
Table 2.2: Tasks Vs. Monitors (Contd.)

<table>
<thead>
<tr>
<th>TASKS</th>
<th>MONITORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification of sequential execution is more natural</td>
<td>The choice of first come first served system irrespective of the called procedure is more natural.</td>
</tr>
<tr>
<td>A guarded select statement associates with callable procedures.</td>
<td>A guarded command in the monitor do not associate with procedures.</td>
</tr>
<tr>
<td>When none of the guards is true the execution of the task is suspended until an external event occurs.</td>
<td>When none of the guards is true the primary action is suspended and the monitor executes a waiting secondary action.</td>
</tr>
<tr>
<td>Tasks cannot always model monitors as the latter use coroutine protocols.</td>
<td>The monitor procedures use coroutine protocols.</td>
</tr>
</tbody>
</table>

2.6 SOLUTIONS WITH DISTRIBUTED CONTROL

From the point of view of control all the solutions discussed earlier are centralised. The algorithms that permit the distribution of control work on the basic principle that there is a single privilege that is held by one process at any point of time. This privilege can be procured by any one of the remaining n-1 processes by explicit request in the form of messages. Only the process that holds the privilege can enter the critical section. Some of the algorithms based on this approach for 'n' processes are mentioned below (Raynal 1988).

* Lamport's Algorithm: requires 3(n-1) messages (Lamport 1974), (Peterson 1983)
* Ricart and Agrawala’s Algorithm: requires 2(n-1) messages (Ricart 1981)

* Carvalho and Roucailor’s Algorithm: requires 0 to 2(n-1) messages (Carvalho 1983)

* Ricart and Agrawala’s Algorithm: requires 0 to n messages (Ricart 1983)

* Suzuki and Kasami Algorithm: requires 0 to n messages (Suzuki 1982)

Elective algorithms (Change 1979), (Dolev 1982), (Franklin 1982), (Hirshberg 1980), (Korach 1984), (Peterson 1982) determine the process which initially generates and keeps track of the privilege. It is evident that the success of these algorithms entirely depends on the well-being of the privilege. The generation and maintenance of the privilege thus demand a careful implementation of fault-tolerance within the system.

2.7 CONCLUSION

This chapter presents an overview of existing techniques to solve the problem of Mutual Exclusion in concurrent programming environments. The solutions are categorised based on the basic principle of implementation. Popular algorithms or techniques are specified in each category. The language level solutions namely monitors and tasks are of particular relevance to the work.