Unit 3 : Process Coordination

Basic Concepts of Concurrency:

A concurrent program specifies two or more sequential programs (a sequential program specifies sequential execution of a list of statements) that may be executed concurrently as parallel processes. For example, an airline reservation system that involves processing transactions from many terminals has a natural specifications 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.

The operating system consists of a collection of such processes which are basically two types:

- **Operating system processes :** Those that execute system code.
- **User processes :** Those that execute user's code.

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. Multiprogramming is a more general concept in operating system that supports memory management and file management features, in addition to supporting concurrent execution of programs.

Basic Concepts of Interprocess Communication and 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.

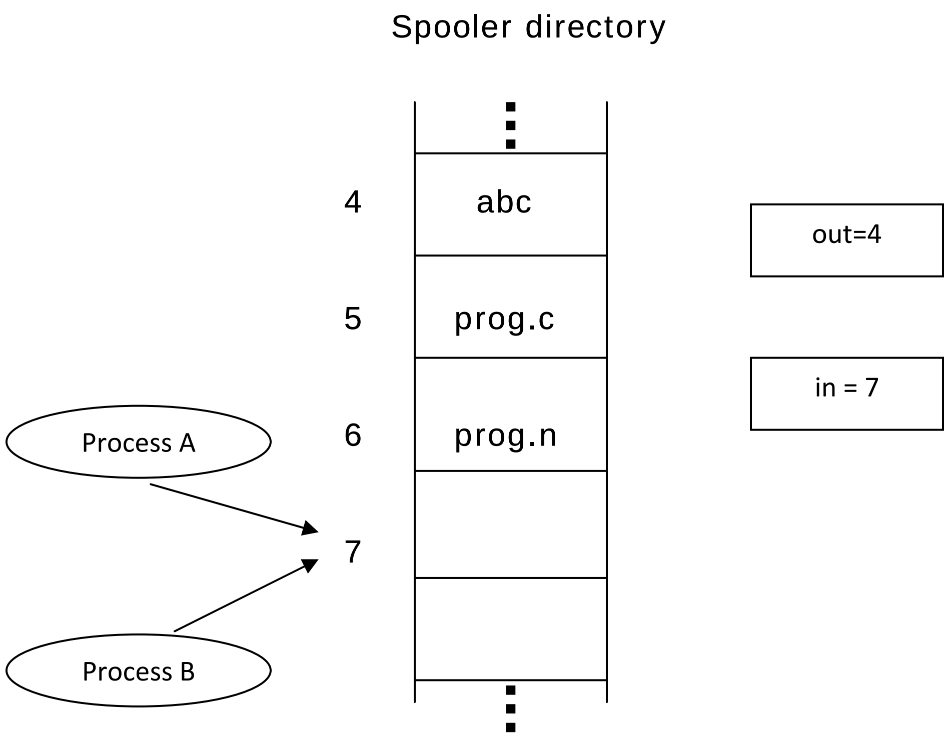
To make this concept more clear, consider the batch operating system again. A shared buffer is used for communication between the reader process and the executor process. These processes must be synchronized so that, for example, the executor process never attempts to read data from the input if the buffer is empty.

Race Condition:

Processes that are working together often share some common storage that one can read and write. The shared storage may be in main memory or it may be a shared file. Processes frequently need to communicate with other processes. When a user wants to read from a file, it must tell the file process what it wants, then the file process has to inform the disk process to read the required block.

When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer process**, periodically checks to see if there are any files to be printed, and if there are it prints them and then removes their names from the directory.

Imagine that the spooler directory has a large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables, **out**, which points to the next file to be printed, and **in**, which points to the next free slot in the directory and these are available to all processes. At a certain instant, slot 0 to 3 are empty and slots 4 to 6 are full. More or less simultaneously, process A and B decided to queue a file for printing.



Process A reads **in** and stores the value 7 in a local variable called **next_free_slot**. Just then the CPU decides that process A has run long enough, so it switches to process B. Process B also reads **in** and also gets a 7 so stores the name of its file in slot 7 and updates **in** to be an 8. When process A runs again, starting from the place it left off, it finds a 7 in **next_free_slot** and writes its file name in slot 7, by erasing the name that process B just put there and then sets **in** to 8. Situations like this, where 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**.

Serialization (To avoid concurrency related problem)

Make an operating system not to perform several tasks in parallel.

Two strategies to serializing processes in a multitasking environment:

- The Scheduler can be disabled
- A Protocol can be introduced

The Scheduler can be disabled

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 will be 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

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.

The Critical-Section Problem:

Consider a system consisting of n processes. Each process has a segment of code called **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. When one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The section of code in a process that request permission to enter into its critical section is called **entry section** and the critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

When one process is executing in its critical section, no other process is to be executed in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time.

```
Do
{
        Entry Section       -   Section of code that request permission to
                                enter its critical section.
        Critical Section    -   It is a part of code in which it is necessary to
                                have exclusive access to shared data.
        Exit Section        -   Code for tasks just after exiting from the
                                critical section.
        Remainder Section   -   The remaining code.

} while (TRUE);
```

Fig: General structure of a typical process

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

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

- **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection can not be postponed indefinitely.

- **Bounded Waiting:** There exists 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.

Mutual Exclusion:

Processes frequently need to communicate with other processes. When a user wants to read from a file, it must tell the file process what it wants, then the file process has to inform the disk process to read the required block.

Processes that are working together often share some common storage that one can read and write. The shared storage may be in main memory or it may be a shared file. Each process has segment of code, called a critical section, which accesses shared memory or files. The key issue involving shared memory or shared files is to find way to prohibit more than one process from reading and writing the shared data at the same time. What we need is mutual exclusion **-** some way of making sure that if one process is executing in its critical section, the other processes will be excluded from doing the same thing.

An algorithm to support mutual exclusion. This is applicable for two processes only.

```
Module Mutex
var
        P1busy, P2busy : boolean;
Process P1
begin
        while true do
        begin
            P1busy :=true;
            While P2busy do {keep testing};
            critical.-,section;
            P1busy:=false;
            Other_P1busy_Processing
        end {while}
end; {P1}

Process P2
```

```
begin
        while true do
        begin
            P2busy :=true;
            While P1busy do {keep testing};
            critical.-,section;
            P2busy:=false;
            Other_P2busy_Processing
        end {while}
end; {P2}

{Parent process}
begin (mutex)
        P1busy:=false;
        P2busy:=false;
        Initiate P1, P2
End (mutex)
```

Program : Mutual Exclusion Algorithm

P1 first sets P1busy and then tests P2busy to determine what to do next. When it finds P2busy to be false, process P1 may safely proceed to the Critical section knowing that no matter how the two processes may be interleaved, process P2 is certain to find P2busy set and to stay away from the critical section. The single change ensures mutual exclusion.

But consider a case where P1 wishes to enter the critical section and sets P1busy to indicate the fact. If process P2 wishes to enter the critical section at the same time and pre-empts process P1 just before P1 tests P2busy.

Process P2 may set P2busy and start looping while waiting for P1busy to become false. When control is eventually returned to Process P1, it finds P2busy set and starts looping while it waits for P2busy to become false. And so both processes are looping forever, each awaiting the other one to clear the way.

In order to remove this kind of behavior, we must add another requirement to occur in our algorithm. When more than one process wishes to enter the critical section, the decision to grant entrance to one of them must be made in finite time.

Synchronization Hardware:

The critical-section problem can be solved in a uni-processor environment if we can forbid interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

This solution is not feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time-consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

Many machines provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words, atomically - as one uninterruptible unit. These special instructions can be used to solve the critical-section problem.

The TestAndSet instruction can be defined as follows:

```
Boolean TestAndSet (Boolean &target)
{
        Boolean  rv = target;
        Target = true;
        Return  rv;
}
```

Fig : Definition of the TestAndSet instruction.

The important characteristic is that this instruction is executed atomically. Thus, if two TestAndSet instructions are executed simultaneously, each on a different CPU, they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet instruction, then we can implement mutual exclusion by declaring, a Boolean variable **lock**, initialized to **false**. The structure of the process is as follows:

```
do {
        while (TestAndSet(lock));
        critical section
        lock = false;
        remainder section
}while(1)
```

Fig : Mutual-exclusion implementation with TestAndSet.

The Swap instruction can be defined as follows:

```
void  Swap(Boolean &a, Boolean &b)
{
        Boolean temp = a;
        a = b;
        b = temp;
}
```

Fig : Definition of the Swap instruction.

It operates on the contents of two words and it is executed atomically. If the machine supports the Swap instruction, then the mutual exclusion can be provided by declaring a variable **lock** and is initialized to **false**. In addition, each process also has a local Boolean variable **key**. The structure of the process is as follows:

```
do {
        key = true;
        while (key == true)
                Swap(lock, key);
        Critical section
        lock = false;
        remainder section
}while(1);
```

These algorithms do not satisfy the bounded-waiting requirement. The following is an algorithm that uses the TestAndSet instruction satisfies all the critical-section requirements. The common data structures are

```
Boolean waiting[n];
Boolean lock;
```

These data structures are initialized to false.

```
do {
    waiting[i] = true;
    key = ture;
    while (waiting[i] &&  key)
        key = TestAndSet(lock);
    waiting[i] = flase;

    critical section

    j = (i+1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    remainder section
} while(1);
```

Fig : Bounded-waiting mutual exclusion with TestAndSet.

To prove that the mutual exclusion requirement is met, note that process Pi can enter its critical section only if either waiting[i] == flase or key == flase. The value of key can become false only if the TestAndSet is executed. The first process to execute the TestAndSet will find key == flase; all others must wait. The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual exclusion requirement.

To prove the progress requirement is met, note that a process exiting the critical section either sets lock to false, or sets waiting[i] to false. Both allow a process that is waiting to enter its critical section to proceed.

To prove the bounded-waiting requirement is met, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i+1, i+2, ... n-1, 0, 1, ..., i-1). It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n-1 turns.

Semaphores:

To overcome the mutual exclusion problem, a synchronization tool called semaphore was proposed by Dijkstra which gained wide acceptance and implemented in several commercial operating system through system calls or as built-in functions.

A semaphore is a variable which accepts non-negative integer values and except for initialization may be accessed and manipulated through two primitive operations - wait and signal (originally defined as P and V respectively). These names come from the Dutch words Problem (to test) and Verogen (to increment). The two primitives take only argument as the semaphore variable, and may be defined as follows.

a. Wait(s):
   ```
   while S <= 0 do (keep testing);
   S: = S-1;
   ```
   wait operation decrements the value of semaphore variable as soon as it would become non-negative.

b. Signal(s) S:= S+1;

   Signal operation increments the value of semaphore variable.

Modifications to the integer value of the semaphore in the wait and signal operations are executed indivisibly. That is, when one process modifies the semaphore no other process can simultaneously modify the same semaphore value. In addition in the case of wait(s), the testing of the integer value of S (S <= 0) and its possible modification (S :=S-1) must also be executed without any interruption.

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual exclusion.

We can use binary semaphores to deal with the critical-section problem for multiple processes. Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource perform a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it perform a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Program 2 demonstrates the functioning of semaphores. In this program, there are 3 processes to trying to share a common resource which is being protected by a binary semaphore (bsem). (A binary semaphore is a variable which contains only values of 0 and 1) by enforcing its use in mutually exclusive fashion. Each process ensures the integrity of its critical section by opening it with a WAIT operation and closing with a SIGNAL operation on the related semaphore, bsem in our example.

This way any number of concurrent processor might share the resource provided each of these process use wait and signal operation.

The parent process in the program first initializes binary semaphore variable **bsem** to 1 indicating that the source is available. As shown in the table (figure 8) at time T1 no process is active to share the resource. But at time T2 all the three processes become active and want to enter their critical sections to share the resource by running the wait operation. At T2, the bsem is decremented to 0 which indicates that some processes has been given permission to enter the critical section. At time T3, we find that it is P1 which has been given some permission. One important thing is to be noted that only one process is allowed by semaphore at a time to the critical section.

Once P1 is given the permission, it prevents other processes P2 & P3 to read the value of bsem as 1 till the wait operation of P1 decrements bsem to 0. This is why wait operation is executed without interruption.
After grabbing the control from semaphore P1 starts sharing the resource which is depicted at time T3. At T4, P1 executes signal operation to release the resource and comes out of its critical section. As shown in the table that the value of bsem becomes 1 since the resource is now free.
The two remaining processes P2 and P3 have an equal chance to compete the resource. In our example, process P3 become the next to enter the critical section and to use the shared resource. At time T7, process P3 releases the resource and semaphore variable bsem again becomes 1. At this time, the two other processes P1 and P2 will attempt to compete for the resource and they have equal chance to get access.

In our example, it is P2 which gets the chance but it might happens one of the three processes could have never got the chance.

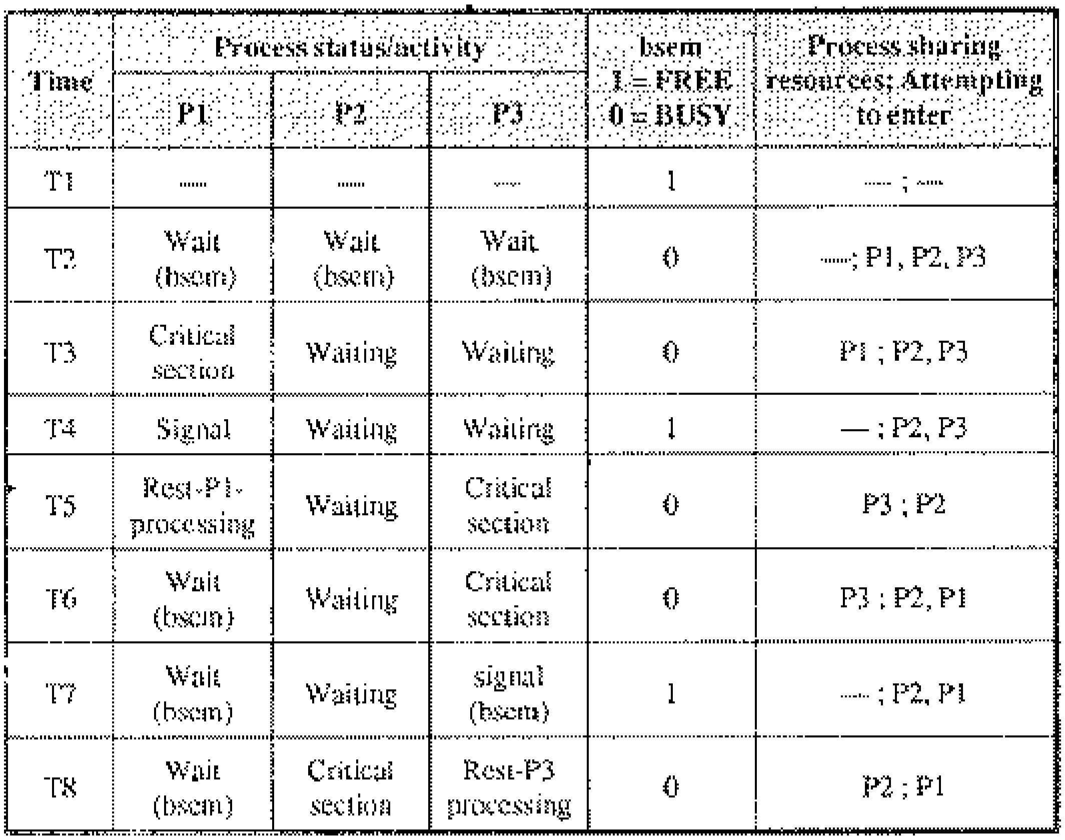
```
module Sem-mutex var bsem : semaphore; {binary
semaphore)
process P1;
Begin
   while true do
      wait (bsem);
      Critical_section
      Signal (bsem);
      The rest_of P1_Processing
end; (P1)


process P2;
Begin
   while true do
      wait (bsem);
      Critical-section;
      signal (bsem);
      The rest of P2-Processing
end; (P2)

process P3;
Begin
   while true do
      wait (bsem);
      Critical-section;
      signal (bsem);
      The rest of P3-Processing
end; (P3)

(Parent process)
begin (sem-mutex)
    bsem:= 1; (free)
    initiate P1, P2, P3;
end; (Mutux)
```

Program 2. Mutual Exclusion with Semaphore



Time	Process status/activity			bsem 1 = FREE 0 = BUSY	Process sharing resources; Attempting to enter
	P1	P2	P3		
T1	-----	-----	-----	1	----- ; -----
T2	Wait (bsem)	Wait (bsem)	Wait (bsem)	0	-----; P1, P2, P3
T3	Critical section	Waiting	Waiting	0	P1 ; P2, P3
T4	Signal	Waiting	Waiting	1	— ; P2, P3
T5	Rest-P1-processing	Waiting	Critical section	0	P3 ; P2
T6	Wait (bsem)	Waiting	Critical section	0	P3 ; P2, P1
T7	Wait (bsem)	Waiting	signal (bsem)	1	----- ; P2, P1
T8	Wait (bsem)	Critical section	Rest-P3 processing	0	P2 ; P1

Figure 8: Run time behaviour of Process

We also present a table (figure 8) showing the run time behaviour of three processes and functioning of semaphore. Each column of the table show the activity of a particular process and the value of a semaphore after certain action has been taken on this process.

Classical Problems of Synchronization

- Bounded-Buffer Problem

To avoid the occurrence of race condition, we present a solution to the bounded-buffer problem using semaphores. The biggest advantage of this solution using semaphores is that it not only avoids the occurrence of race condition but also allows to have **size** items in the buffer at the same time, thus, eliminating the shortcomings of the solutions using shared memory. The following three semaphores are used in this solution.

We assume that the pool consists of n buffers, each capable of holding one item. The *mutex* semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The *empty* and *full* semaphores count the number of empty and full buffers. The semaphore *empty* is initialized to the value n; the semaphore *full* if initialized to the value 0.

The structure of the producer process

```
while (true)  {
        //   produce an item
    wait (empty);
    wait (mutex);
        //  add the item to the  buffer
    signal (mutex);
    signal (full);
              }
```

The structure of the consumer process

```
while (true) {

    wait (full);

    wait (mutex);

        //  remove an item from  buffer

    signal (mutex);
```

```
        signal (empty);
    //  consume the removed item
                }
```

We can interpret these codes as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

- Readers and Writers Problem

Concurrently executing processes that are sharing a data object, such as a file or a variable, fall into two groups: readers and writers. The processes in the **readers group** want only to read the contents of the shared object, whereas, the processes in **writers group** want to update (read and write) the value of shared object. There is no problem if multiple readers access the shared object simultaneously. However, if a writer and some other process (either a reader or writer) access the shared object simultaneously, data may become inconsistent.

To ensure that such a problem does not arise, we must guarantee that when a writer is accessing the shared object, no reader or writer accesses that shared object. This synchronization problem is termed as **readers-writers problem**. The readers-writers problem has several variations. The simplest one referred to as the *first* reader-writer problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for others readers to finish simply because a writer is waiting. The *second* readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case writers may starve, in the second case, readers may starve.

Following is a solution to the first readers-writers problem. The reader process share the following data structures:
semaphore mutex, wrt;
int readcount;

The semaphore mutex and wrt are initialized to 1 and readcount is initialized to 0. The semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exists the critical section. It is not used by readers who enter or exit while other readers are in their critical section.

If a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n-1 readers are queued on mutex. Also, observe that, when a writer executes signal(wrt), we may resume the execution of either the waiting readers or a single waiting writer.

<u>The structure of a writer process</u>

```
while (true) {
        wait (wrt) ;

            //    writing is performed
        signal (wrt) ;
}
```

<u>The structure of a reader process</u>

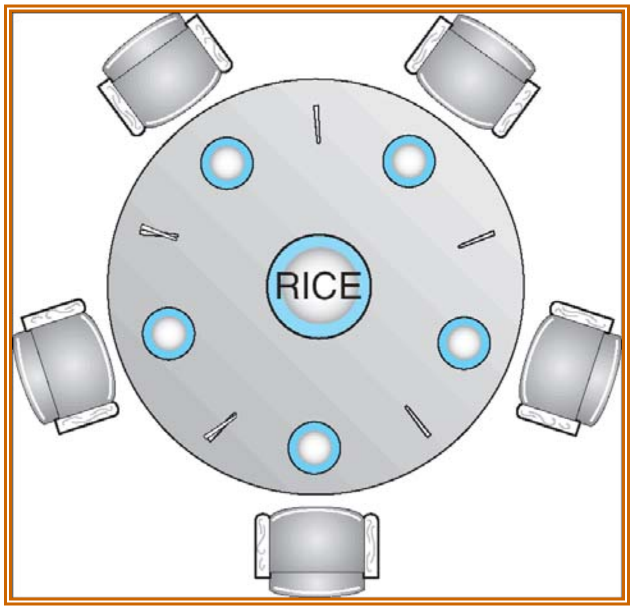
```
while (true) {
       wait (mutex) ;
       readcount ++ ;
```

```
                if (readercount == 1)  wait (wrt) ;
                signal (mutex)

                       // reading is performed
                 wait (mutex) ;
                 readcount  - - ;
                 if (redacount  == 0)  signal (wrt) ;
                 signal (mutex) ;
         }
```

- Dining-Philosophers Problem

Consider five philosophers sitting around a circular table. There is a bowl of rice in the centre of the table and five chopsticks – one in between each pair of philosophers.



Initially, all the philosophers are in the thinking phase and while thinking, they make sure that they do not interact with each other. As time passes by, philosophers might feel hungry. When a philosopher feels hungry, he attempts to pick up the two chopsticks kept in close proximity to him (that are in between him and his left and his right philosophers). If the philosophers on his left and right are not eating, he successfully gets the two chopsticks. With the two chopsticks in his hand, he starts eating. After he finishes eating, the chopsticks are positioned back on the table and the philosopher begins to think again. On the contrary, if the philosopher on his left or right is already eating, then fails to grab the two chopsticks at the same time, and thus, has to wait.

A solution to this problem is to represent each chopstick as a semaphore, and philosophers must grab or release chopsticks by executing wait operation or signal operation respectively on the appropriate semaphores. We use an array *chopstick* of size 5 where each element is initialized to 1.

<u>The structure of Philosopher *i*:</u>

```
While (true)  {
        wait ( chopstick[i] );
           wait ( chopStick[ (i + 1) % 5] );

                  //  eat
           signal ( chopstick[i] );
           signal (chopstick[ (i + 1) % 5] );

              //  think
}
```

This solution is simple and ensure that no two neighbors are eating at the same time. However, the solution is not free from deadlock. Suppose all the philosophers attempt to grab the chopsticks simultaneously and grab one chopstick successfully. In this case, all the elements of chopstick will be 0. Thus, when each philosopher attempts to grab the second chopstick, he will go in waiting state forever.

Several possible remedies to the deadlock problem are available:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available.
- Use an asymmetric solution; that is, an odd philosopher picks up first his left chopstick and then his right chopsticks, whereas an even philosopher picks up his right chopstick and then her left chopstick.

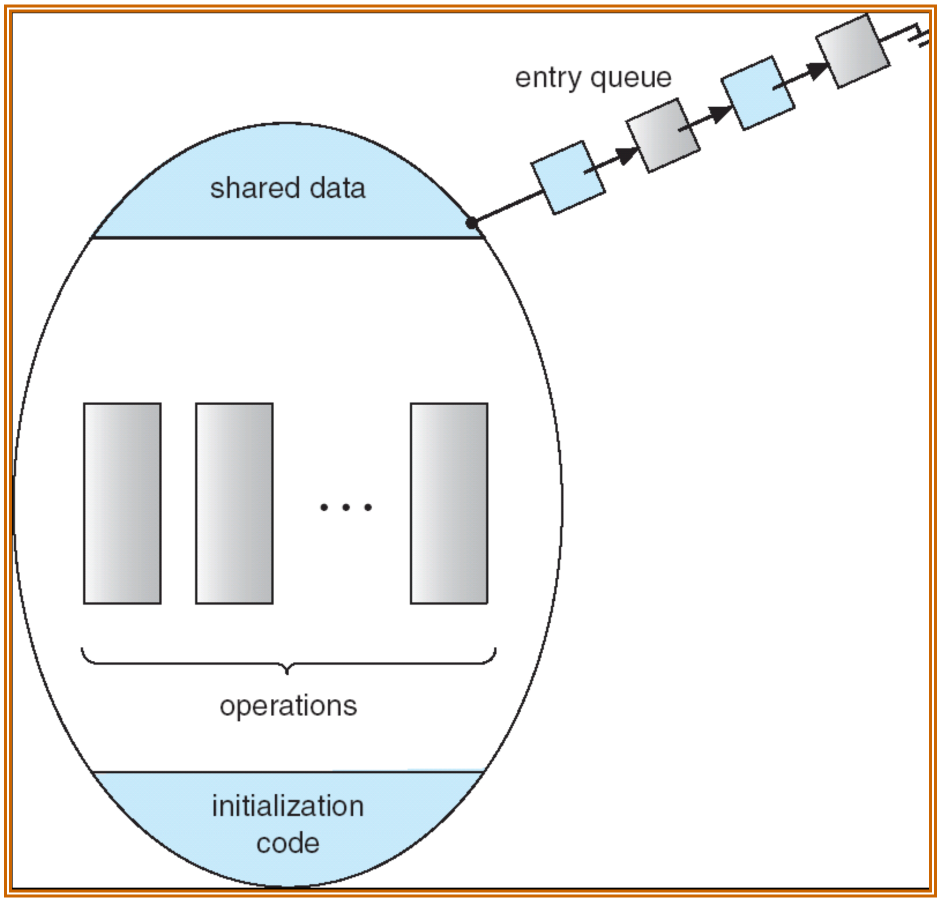
Dining-philosophers problem can be solved with the use of monitors.

Monitors

A monitor is a programming language construct which is also used to provide mutually exclusive access to critical sections. The programmer defines monitor type which consists of declaration of shared data (or variables), procedures or functions that access these variables, and initialization code. The general syntax of declaring syntax of declaring a monitor type is as follows:

```
monitor <monitor-name>
{
// shared data ( or variable) declarations
data type <variable-name>;
...
// function (or procedure) declarations
return_type <function_name> (parameters)
{
        // body of function
}
.
.
monitor-name()
{
        // initialization
}
}
```

The variables defined inside a monitor can only be accessed by the functions defined within the monitor, and it is not feasible for any process to access these variables. Thus, if any process has to access these variables, it is only possible through the execution of the functions defined inside the monitor. Further, the monitor construct checks that only one process may be executing within the monitor at a given moment. But if a process is executing within the monitor, then other requesting processes are blocked and placed on an entry queue.



Schematic view of a monitor

However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the **condition** construct. We can define a mechanism by defining variables of **condition** type on which only two operations can be invoked: wait and signal.

Suppose, programmer defines a variable **C** of **condition** type, then execution of the operation **C.wait()** by a process **Pi**, suspends the execution of **Pi**, and places it on a queue associated with the **condition** variable **C**. On the other hand, the execution of the operation **C.signal()** by a process **Pi**, resumes the execution of exactly one suspended process **Pj**, if any. It means that the execution of the **signal** operation by **Pi** allows other suspended process **Pj** to execute within the monitor. However, only one process is allowed to execute within the monitor at one time. Thus, monitor construct prevents **Pj** from resuming until **Pi** is executing in the monitor. There are following possibilities to handle this situation.

- The process **Pi** must be suspended to allow **Pj** to resume and wait until **Pj** leaves the monitor.
- The process **Pj** must remain suspended until **Pi** leaves the monitor.
- The process **Pi** must execute the **signal** operation as its last statement in the monitor so that **Pj** can resume immediately.

The solution to the dining-philosophers problem is as follows:
The distribution of the chopsticks is controlled by the monitor **dp**. Each philosopher, before starting to eat, must invoke the operation **pickup()**. This may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the **putdown()** operation. Thus, philosopher **i** must invoke the operations **pickup()** and **putdown()** in the following sequence:

```
dp.pickup(i);
...
eat
...
dp.putdown(i);
```

```
monitor DP
   {
        enum { THINKING; HUNGRY, EATING) 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 left and right neighbors
              test((i + 4) % 5);
              test((i + 1) % 5);
       }

void test (int i) {
              if ( (state[(i + 4) % 5] != EATING) &&
              (state[i] == HUNGRY) &&
              (state[(i + 1) % 5] != EATING) )
               {
                  state[i] = EATING ;
                 self[i].signal () ;
               }
        }
      initialization_code() {
             for (int i = 0; i < 5; i++)
             state[i] = THINKING;
       }
}
```

After eating is finished, each philosopher invokes **putdown()** operation before start thinking. This operation changes the state of philosopher process to thinking and then invoke **test((i + 4) % 5)** and **test((i + 1) % 5)** operation for philosophers on his left and right side (one by one). This verifies whether the philosopher feels hungry, and if so then allows him to eat in case philosophers on his left and right side are not eating.

DEADLOCKS

In a multiprogramming environment several processes may compete for a fixed number of resources. A process requests resources and if the resources are not available at that time, it enters a wait state. It may happen that the waiting process will never gain access to the resources. Since those resources are being held by other waiting processes.

For example, take a system with one tape drive and one plotter. Process P1 request the tape drive and process P2 requests the plotter. Both requests are granted. Now P1 requests the plotter (without giving up the tape drive) and P2 requests the tape drive (without giving up the plotter). Neither request can be granted so both processes enter a deadlock situation.

A deadlock is a situation where a group of processes is permanently blocked as a result of each process having acquired a set of resources needed for its completion and having to wait for release of the remaining resources held by others thus making it impossible for any of the deadlocked processes to proceed. Deadlocks can occur in concurrent environments as a result of the uncontrolled granting of the system resources to the requesting processes.

System Model:

Deadlocks can occur when processes have been granted exclusive access to devices, files and so forth. A system consists of a finite number of resources to be distributed among a number of competing processes. The resources can be divided into several types, each of which consists of some number of identical instances. CPU cycles, memory space, files and I/O devices (such as printers and tape drives) are examples of resource types. If a system has two tape drives then the resource type tape drive has two instances.

If a process requests an instance of a resource type, any type of that resource of class may satisfy the request. If this is not the case, then the instances are not identical and the resource type classes have not been properly defined. For example a system may have two printers These two printers may be defined into same printer class, if one is not concerned about type of printers (Dot Matrix or Laser Printer).

Whenever a process wants to utilize any resource, it must make a request for it. It may request as many resources as it wants but it should not exceed the total number of resources available with the system. Once the process has utilized the resource it must release it. Therefore, a sequence of events to use a resource is:

i. Request the resource:
ii. Use the resource:
iii. Release the resource:

Request and release of resources can be accomplished through the wait and signal operations on semaphores. A system table records whether each resource is free or allocated, and, if a resource is allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

Deadlock Characterization:

Deadlocks are undesirable features. In the most of deadlock situation process is waiting for the release of some resource concurrently possessed by some deadlocked process. A deadlock situation can arise if the following four conditions hold simultaneously in a system:

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Mutual exclusion:

At least one resource must be held in a non-sharable mode; that is only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

Hold and wait:

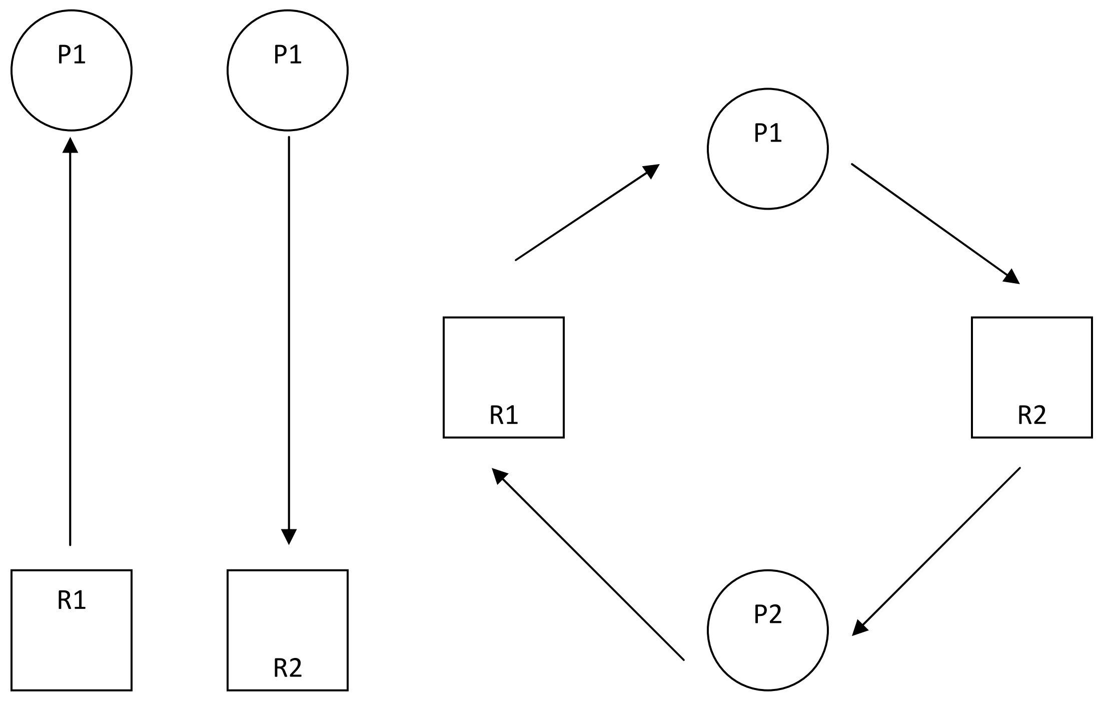
A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

No preemption:

Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait:

A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, ..., Pn-1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.



Holding a resource

Requesting a resource

Deadlock situation

Graphic Representation of Resource Allocation