ASSIGNMENT NO :- 6

TITLE: Deadlock Avoidance Using Semaphores

OBJECTIVE:

1. To understand the deadlock and starvation problem using programming.
2. To study dining philosophers problem of operating system.

Problem Statement:

Implement the deadlock-free solution to Dining Philosophers problem to illustrate the problem of deadlock and/or starvation that can occur when many synchronized threads are competing for limited resources.

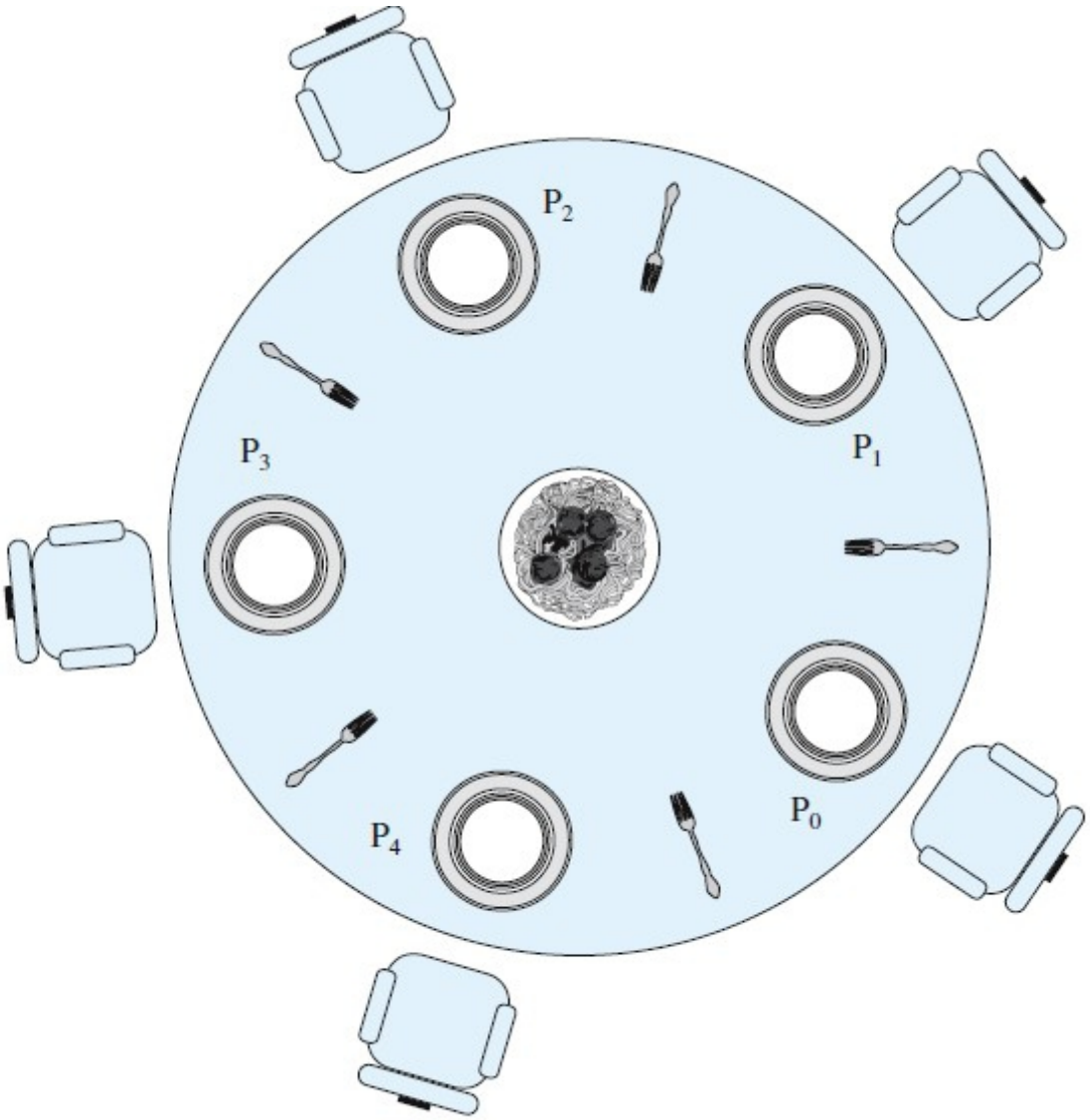
Theory:-

The Dining Philosophers Problem

The dining philosophers problem was introduced by Dijkstra. Five philosophers live in a house, where a table is laid for them. The life of each philosopher consists principally of thinking and eating, and through years of thought, all of the philosophers had agreed that the only food that contributed to their thinking efforts was spaghetti. Due to a lack of manual skill, each philosopher requires two forks to eat spaghetti.

The eating arrangements are simple: a round table on which is set a large serving bowl of spaghetti, five plates, one for each philosopher, and five forks. A philosopher wishing to eat goes to his or her assigned place at the table and, using the two forks on either side of the plate, takes and eats some spaghetti.

The problem: devise a ritual (algorithm) that will allow the philosophers to eat. The algorithm must satisfy mutual exclusion (no two philosophers can use the same fork at the same time) while avoiding deadlock and starvation. This problem may not seem important or relevant in itself. However, it does illustrate basic problems in deadlock and starvation. Furthermore,
attempts to develop solutions reveal many of the difficulties in concurrent programming. In addition, the dining philosophers problem can be seen as representative of problems dealing with the coordination of shared resources, which may occur when an application includes concurrent threads of execution. Accordingly, this problem is a standard test case for evaluating approaches to synchronization.



Dining Arrangement for Philosophers

Solution Using Semaphores

Each philosopher picks up first the fork on the left and then the fork on the right. After the philosopher is finished eating, the two forks are replaced on the table.

```
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
while (true)
{
think();
wait (fork[i]);
wait (fork [(i+1) mod 5]);
eat();
```

```
signal(fork [(i+1) mod 5]);
signal(fork[i]);
}
}
void main()
{
parbegin (philosopher (0), philosopher (1), philosopher (2), philosopher (3),
philosopher (4));
      }
```

Semaphores:

The various hardware based solutions can be difficult for application programmers to implement. **Semaphores** are most often used to synchronize operations (to avoid race conditions) when multiple processes access a common, non-shareable resource. *Semaphores* are integer variables for which only two (atomic) operations are defined, the **wait (P)** and **signal (V)** operations, whose definitions in pseudocode are shown in the following figure.

Wait:

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

Signal:

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

P(S) or S.wait(): decrement or block if already 0
V(S) or S.signal(): increment and wake up process if any

To indicate a process has gained access to the resource, the process decrements the semaphore. Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly. When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. Access to the semaphore is provided by a series of semaphore system calls. Semaphores can be used to deal with the *n-pro*cesses critical section problem, where the *n-processes* share a semaphore **mutex** (mutual exclusion) initialized to 1. Each process Pi is organized as shown:

```
do {
   waiting(mutex);
     // critical section
    signal(mutex);
     // remainder section
}while (TRUE);
```

Mutual-exclusion implementation with semaphores.

Implementation

The big problem with semaphores described above is the busy loop in the wait call (**busy waiting)**, which consumes CPU cycles without doing any useful work. This type of lock is known as a ***spinlock***, because the lock just sits there and spins while it waits. While this is generally a bad thing, it does have the advantage of not invoking context switches, and so it is sometimes

used in multi-processing systems when the wait time is expected to be short - One thread spins on one processor while another completes their critical section on another processor. An alternative approach is to block a process when it is forced to wait for an available semaphore, and swap it out of the CPU. In this implementation each semaphore needs to maintain a list of processes that are blocked waiting for it, so that one of the processes can be woken up and swapped back in when the semaphore becomes available. (Whether it gets swapped back into the CPU immediately or whether it needs to hang out in the ready queue for a while is a scheduling problem.) The new definition of a semaphore and the corresponding wait and signal operations are shown as follows:

Semaphore Structure:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Wait Operation:

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
        add this process to S->list;
        block();
  }
}
```

Signal Operation:

```
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
  }
}
```

OS's 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. A binary semaphore must be initialized with 1 or 0, and the completion of P and V operations must alternate. If the semaphore is initialized with 1, then the first completed
operation must be P. If the semaphore is initialized with 0, then the first completed operation must be V. Both P and V operations can be blocked, if they are attempted in a consecutive manner. Binary semaphores are known as mutex locks as they are locks that provide mutual exclusion. Binary semaphores are used 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. Counting semaphores maintain a count of the number of times a resource is given. The semaphore is initialized to the number of resources

available. Each process that wishes to use a resource performs a wait() operation on the semaphore. When a process releases a resource, it performs a signal() operation.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event (execution of a signal()) that can be caused only by one of the waiting processes. When such a state is reached, these processes are said to be **deadlocked.**

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Another problem related to deadlocks is indefinite blocking or starvation, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO order.

Header files :

1)#include<stdio.h> : This ANSI header file relates to "standard" input/output functions. Files, dvices and directories are referenced using pointers to objects of the type FILE . The header file contains declarations for these functions and macros, defines the FILE type, and contains various constants related to files.

2)#include<stdlib.h>:

This ANSI header file contains declarations for many standard functions,excluding those declared in other header files discussed in this section.

3)#include<pthread.h> : when we use pthread.h library in our program, program doesn't execute using simple compile command after including pthread.h header file, program doesn't compile and return error.

To compile c program with pthread.h library, you have to put –lpthread just after the compile command will tell to the compiler to execute program with pthread.h library.

The command is : cc -pthread -o dp dp.c

cc is the compiler command.

dp**.c** is the name of c program file.

-**o** is option to make header file.

dp is the name of object file.

-**lpthread** is option to execute pthread.h library file.

4)#include <unistd.h>

The <*unistd.h*> header defines miscellaneous symbolic constants and types, and declares miscellaneous functions. The contents of this header are shown below.

Version Test Macros

The following symbolic constants are defined:

_POSIX_VERSION
Integer value indicating version of the ISO POSIX-1 standard (C language binding).
_POSIX2_VERSION
Integer value indicating version of the ISO POSIX-2 standard (Commands).
_POSIX2_C_VERSION
Integer value indicating version of the ISO POSIX-2 standard (C language binding).
_XOPEN_VERSION
Integer value indicating version of the X/Open Portability Guide to which the implementation conforms.
_POSIX_VERSION is defined in the ISO POSIX-1 standard. It changes with each new version of the ISO POSIX-1 standard.

_POSIX2_VERSION is defined to have the value of the ISO POSIX-2 standard's POSIX2_VERSION limit. It changes with each new version of the ISO POSIX-2 standard.

XOPEN_VERSION is defined as an integer value equal to 500.

_XOPEN_XCU_VERSION is defined as an integer value indicating the version of the **XCU** specification to which the implementation conforms. If the value is -1, no commands and utilities are provided on the implementation. If the value is greater than or equal to 4, the functionality associated with the following symbols is also supported (see Mandatory Symbolic Constants and Constants for Options and Feature Groups):

_POSIX2_C_BIND
_POSIX2_C_VERSION
_POSIX2_CHAR_TERM
_POSIX2_LOCALEDEF
_POSIX2_UPE
_POSIX2_VERSION

Constants for Functions

The following symbolic constant is defined:

NULL
Null pointer

The following symbolic constants are defined for the *access()* function:

R_OK
Test for read permission.
W_OK
Test for write permission.
X_OK
Test for execute (search) permission.
F_OK
Test for existence of file.

CONCLUSION:

The dining philosopher’s problem can be seen as represent ative of problems dealing withthe coordination of shared resources, which may occur when an application includes concurrent deadlock of execution.One simple solution is to represent each chopstick by a semaphore.