Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.

Name: Adwait Purao

UID: 2021300101

Batch: B2

Experiment no.: 6

Aim: To design solution for dining philosophers' problem using binary semaphores

Theory:

Introduction of Process Synchronization

Introduction:

Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.

The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other, and to prevent the possibility of inconsistent data due to concurrent access. To achieve this, various synchronization techniques such as semaphores, monitors, and critical sections are used.

In a multi-process system, synchronization is necessary to ensure data consistency and integrity, and to avoid the risk of deadlocks and other synchronization problems. Process synchronization is an important aspect of modern operating systems, and it plays a crucial role in ensuring the correct and efficient functioning of multi-process systems.

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process**: The execution of one process does not affect the execution of other processes.
- **Cooperative Process**: A process that can affect or be affected by other processes executing in the system.

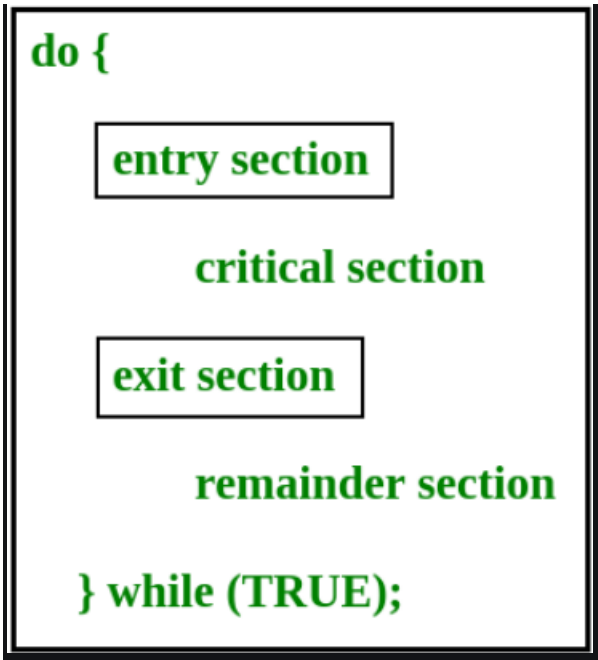
Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Race Condition:

When more than one process is executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition. Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place. A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Critical Section Problem:

A critical section is a code segment that can be accessed by only one process at a time. The critical section contains shared variables that need to be synchronized to maintain the consistency of data variables. So the critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.



In the entry section, the process requests for entry in the **Critical Section.** Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress**: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can not be postponed indefinitely.
- **Bounded Waiting**: A bound must exist 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.

Peterson's Solution:

Peterson's Solution is a classical software-based solution to the critical section problem. In Peterson's solution, we have two shared variables:

- boolean flag[i]: Initialized to FALSE, initially no one is interested in entering the critical section
- int turn: The process whose turn is to enter the critical section.

```
do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;

        critial section

    flag[i] = FALSE ;

        remainder section

  }  while (TRUE) ;
```

Peterson's Solution preserves all three conditions:

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's solution:

- It involves busy waiting.(In the Peterson's solution, the code statement- "while(flag[j] && turn == j);" is responsible for this. Busy waiting is not favored because it wastes CPU cycles that could be used to perform other tasks.)
- It is limited to 2 processes.
- Peterson's solution cannot be used in modern CPU architectures.

Semaphores:

A semaphore is a signaling mechanism and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that is called the wait function.

A semaphore uses two atomic operations, wait and signal for process synchronization.
A Semaphore is an integer variable, which can be accessed only through two operations wait() and signal().

There are two types of semaphores: Binary Semaphores and Counting Semaphores.

- **Binary Semaphores:** They can only be either 0 or 1. They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section. When it completes its critical section, it can reset the value of the mutex semaphore to 0 and some other process can enter its critical section.
- **Counting Semaphores:** They can have any value and are not restricted over a certain domain. They can be used to control access to a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

Advantages and Disadvantages:
Advantages of Process Synchronization:

- Ensures data consistency and integrity
- Avoids race conditions
- Prevents inconsistent data due to concurrent access
- Supports efficient and effective use of shared resources

Disadvantages of Process Synchronization:

- Adds overhead to the system
- Can lead to performance degradation
- Increases the complexity of the system
- Can cause deadlocks if not implemented properly.

Dining Philosophers Problem

The dining philosophers problem is a classic problem in computer science and distributed computing. It illustrates the challenges of resource allocation and concurrency in a distributed system.

The problem is as follows:

Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti and a fork to eat with. The spaghetti is divided into individual strands, and each philosopher needs to pick up two forks to eat the spaghetti. However, there are only five forks available, with one fork placed between each pair of adjacent philosophers.

The philosophers alternate between thinking and eating. When a philosopher wants to eat, he needs to pick up the two forks adjacent to him. If both forks are available, he can eat, otherwise, he needs to wait until the necessary forks are available. After he finishes eating, he puts down the forks and starts thinking again.

The challenge is to design a protocol that allows each philosopher to eat without getting into a deadlock or starvation situation, where a philosopher is stuck waiting for a fork that is never released.

Various solutions have been proposed to solve the dining philosophers problem, including resource hierarchy, resource ordering, and arbitration. Some common approaches include using semaphores, monitors, or message passing to coordinate the access to forks.

List of solutions

There are several solutions to the dining philosophers problem, including:

Resource hierarchy: Assign a unique identifier to each fork and have the philosophers always try to acquire the lower-numbered fork first. This avoids the possibility of circular waiting.

Resource ordering: Establish a strict ordering of resources (forks) and have the philosophers always acquire them in the same order. This can be achieved by assigning a numerical identifier to each philosopher and having them only acquire the fork to their left first, and then the fork to their right.

Arbitration: Introduce a designated arbitrator, such as a waiter, to manage the allocation of resources. The arbitrator decides when a philosopher can pick up or put down a fork, ensuring that the resources are not overused and that no philosopher starves.

Chandy/Misra solution: Use a token-based protocol where a philosopher can only eat when they hold a token. The token is passed from philosopher to philosopher in a predetermined order, ensuring that only one philosopher can eat at a time.

Dijkstra's solution: Use a semaphore for each fork and introduce a protocol to ensure that the philosophers can acquire both forks at the same time. This solution uses a "test-and-set" mechanism to allow a philosopher to acquire both forks only when they are both available.

Lamport's solution: Use a logical clock to assign timestamps to the philosophers' actions and establish a partial ordering of events. This solution ensures that the philosophers do not interfere with each other's actions and avoids deadlocks.

Each solution has its advantages and disadvantages, and the choice of which solution to use will depend on the specific requirements and constraints of the system.

Problem:

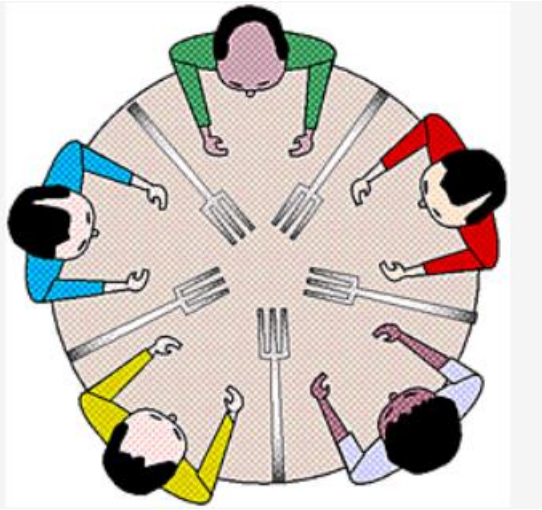
Dining Philosopher's Problem:

5 philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with 5 chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

Write a program to implement this problem using the Semaphores.

Note: Program should work for any count of philosophers

(O/P should display Philosopher no, it's state - thinking/hungry/eating, chopstick no. used)



Solution:

One solution to the dining philosophers problem using binary semaphores is as follows:

1. Initialize a binary semaphore for each fork, initially set to 1 to indicate that the fork is available.
2. When a philosopher wants to eat, they need to acquire both forks by decrementing the binary semaphore for each fork. If a fork is not available, the philosopher will be blocked until the fork is released.
3. After the philosopher has finished eating, they release both forks by incrementing the binary semaphore for each fork.
4. To prevent deadlocks, introduce an arbitrary rule that all philosophers must pick up the fork on their left first, and then the fork on their right.

Code:

```
#include <stdio.h>
```

```
#include <stdlib.h>

#include <time.h>

#include <string.h>

#include <unistd.h>

#include <pthread.h>


/*

use the pthread flag with gcc to compile this code

~$ gcc -pthread dining_philosophers.c -o dining_philosophers


On windows : To compile and run

gcc -g -pthread dining_philosophers.c -o dining_philosophers

./dining_philosophers.exe

*/


pthread_t *philosophers;        // Array of threads representing philosophers

pthread_mutex_t *forks;         // Array of mutexes representing forks

int philosophers_count;         // Number of philosophers


void eat(int i){


   printf("Philosopher %d is eating\n",i+1);
```

```
    sleep(1 + rand()%10);       // Random sleep time between 1 and 10 seconds
}

void* philosopher(void* args){
    int i = 0,first,second;

    // Get philosopher's ID
    while(!pthread_equal(*(philosophers+i),pthread_self()) && i < philosophers_count){
        i++;
    }

    while(1){

        printf("Philosopher %d is thinking\n",i+1);

        sleep(1 + rand()%10);    // Random sleep time between 1 and 10 seconds

        // Acquire forks
        first = i;
        second = (i+1)%philosophers_count;
```

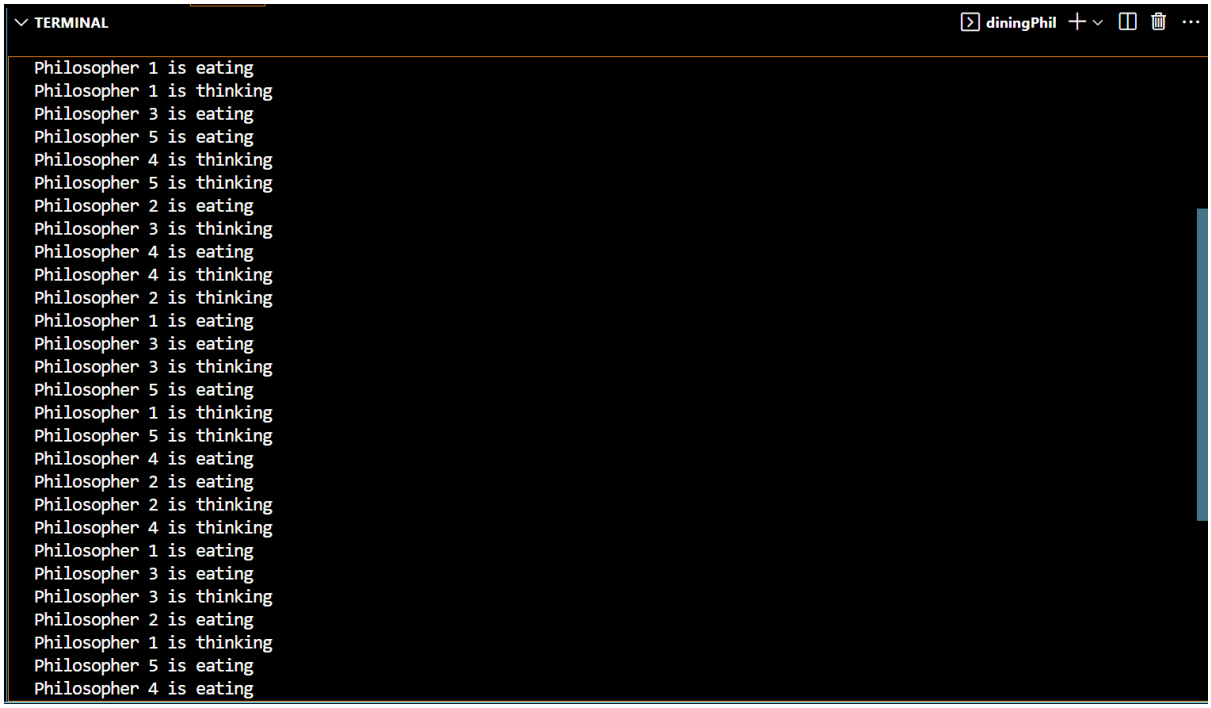
```
        pthread_mutex_lock(forks + (first>second?second:first));
        pthread_mutex_lock(forks + (first<second?second:first));

        // Eat
        eat(i);

        // Release forks
        pthread_mutex_unlock(forks+first);
        pthread_mutex_unlock(forks+second);
    }

    return NULL;
}


int main(void){
    int i,err;

    srand(time(NULL));        // Seed random number generator with current time

    // Get number of philosophers
    printf("Enter number of philosophers:");
```

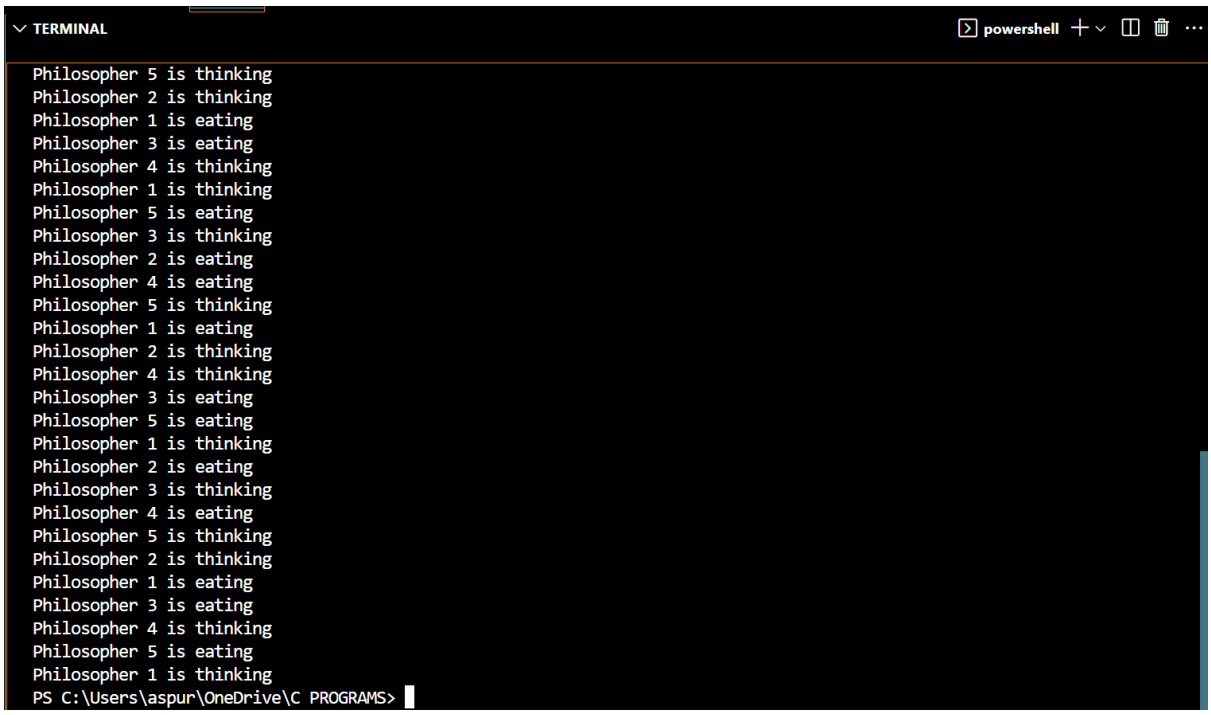
```
   scanf("%d",&philosophers_count);


   // Allocate memory for arrays
   philosophers = (pthread_t*) malloc(philosophers_count*sizeof(pthread_t));
   forks = (pthread_mutex_t*)
malloc(philosophers_count*sizeof(pthread_mutex_t));


   // Initialize forks
   for(i=0;i<philosophers_count;++i)
      if(pthread_mutex_init(forks+i,NULL) != 0){
         printf("Failed initializing fork %d\n",i+1);
         return 1;
      }


   // Create philosopher threads
   for(i=0;i<philosophers_count;++i){
      err = pthread_create(philosophers+i,NULL,&philosopher,NULL);


      if(err != 0){
         printf("Error creating philosopher: %s\n",strerror(err));
      }else{
         printf("Successfully created philosopher %d\n",i+1);
      }
```

```
}

// Wait for philosopher threads to finish
for(i=0;i<philosophers_count;++i)
    pthread_join(*(philosophers+i),NULL);

// Free allocated memory
free(philosophers);
free(forks);

return 0;
}
```

Output:

```
TERMINAL                                                    diningPhil
PS C:\Users\aspur\OneDrive\C PROGRAMS> gcc -g -pthread .\diningPhil.c -o .\diningPhil
PS C:\Users\aspur\OneDrive\C PROGRAMS> .\diningPhil.exe
Enter number of philosophers:5
Successfully created philosopher 1
Successfully created philosopher 2
Successfully created philosopher 3
Philosopher 2 is thinking
Philosopher 1 is thinking
Philosopher 3 is thinking
Successfully created philosopher 4
Philosopher 4 is thinking
Successfully created philosopher 5
Philosopher 5 is thinking
Philosopher 4 is eating
Philosopher 1 is eating
Philosopher 1 is thinking
Philosopher 3 is eating
Philosopher 5 is eating
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 2 is eating
Philosopher 3 is thinking
Philosopher 4 is eating
Philosopher 4 is thinking
Philosopher 2 is thinking
Philosopher 1 is eating
Philosopher 3 is eating
Philosopher 3 is thinking
```





To exit the program press Ctrl + C to exit

Conclusion:

We learnt from the above experiment that process synchronization is the coordination of multiple processes to ensure that they do not interfere with

each other's execution. Binary semaphores are a tool used for process synchronization. They can be used to enforce mutual exclusion, ensuring that only one process can access a shared resource at a time.

In the dining philosophers problem, each philosopher represents a process, and the forks represent the shared resources. The use of binary semaphores allows each philosopher to acquire both forks, eat, and then release the forks, ensuring that all philosophers can eat without deadlocking or starving.

The solution using binary semaphores requires each philosopher to acquire the forks in a specific order, typically the left fork first, to avoid deadlocks.