OS EXPERIMENT 7

Name: Mohammed Arhaan Ali
Div: S11
Roll No: 02

Aim: a) Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm.
b) Write a program to demonstrate the concept of the Dining Philosopser's Problem.

Theory:

A) Banker's Algorithm in Operating System:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Why is Banker's Algorithm Named So?

The banker's algorithm is named so because it is used in the banking system to check whether a loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that the bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money then the bank can easily do it.
It also helps the OS to successfully share the resources between all the processes. It is called the banker's algorithm because bankers need a similar algorithm- they admit loans that collectively exceed the bank's funds and then release each borrower's loan in installments. The banker's algorithm uses the notation of a safe allocation state to ensure that granting a resource request cannot lead to a deadlock either immediately or in the future.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in a safe state always.
The following Data structures are used to implement the Banker's Algorithm:
Let 'n' be the number of processes in the system and 'm' be the number of resource types.

Available

- It is a 1-d array of size 'm' indicating the number of available resources of each type.
- Available[j] = k means there are 'k' instances of resource type Rj

Max

- It is a 2-d array of size 'n*m' that defines the maximum demand of each process in a system.
- Max[i, j] = k means process Pi may request at most 'k' instances of resource type Rj.

Allocation

- It is a 2-d array of size 'n*m' that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process Pi is currently allocated 'k' instances of resource type Rj

Need

- It is a 2-d array of size 'n*m' that indicates the remaining resource need of each process.
- Need [i,   j] = k means process Pi currently needs 'k' instances of resource type Rj
- Need [i,   j] = Max [i,   j] – Allocation [i,   j]

Allocation specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.
Banker's algorithm consists of a Safety algorithm and a Resource request algorithm.

Banker's Algorithm

1. Active:= Running U Blocked;
for k=1…r
New_ request[k]:= Requested_ resources[requesting_ process, k];
2. Simulated_ allocation:= Allocated_ resources;
for k=1…..r //Compute projected allocation state
Simulated_ allocation [requesting _process, k]:= Simulated_ allocation [requesting _process, k] + New_ request[k];
3. feasible:= true;
for k=1….r // Check whether projected allocation state is feasible
if Total_ resources[k]< Simulated_ total_ alloc [k] then feasible:= false;
4. if feasible= true
then // Check whether projected allocation state is a safe allocation state
while set Active contains a process P1 such that
For all k, Total _resources[k] – Simulated_ total_ alloc[k]>= Max_ need [l ,k]-Simulated_ allocation[l, k]
Delete Pl from Active;
for k=1…..r
Simulated_ total_ alloc[k]:= Simulated_ total_ alloc[k]- Simulated_ allocation[l, k];
5. If set Active is empty
then // Projected allocation state is a safe allocation state
for k=1….r // Delete the request from pending requests
Requested_ resources[requesting_ process, k]:=0;
for k=1….r // Grant the request
Allocated_ resources[requesting_ process, k]:= Allocated_ resources[requesting_ process, k] + New_ request[k];
Total_ alloc[k]:= Total_ alloc[k] + New_ request[k];

Program:

A) Banker's:

```
#include <stdio.h>
int main()
{
   // P0, P1, P2, P3, P4 are the Process names here

   int n, m, i, j, k;
```

```
n = 5; // Number of processes
m = 3; // Number of resources
int alloc[5][3] = { { 0, 1, 0 }, // P0    // Allocation Matrix
                    { 2, 0, 0 }, // P1
                    { 3, 0, 2 }, // P2
                    { 2, 1, 1 }, // P3
                    { 0, 0, 2 } }; // P4

int max[5][3] = { { 7, 5, 3 }, // P0    // MAX Matrix
                  { 3, 2, 2 }, // P1
                  { 9, 0, 2 }, // P2
                  { 2, 2, 2 }, // P3
                  { 4, 3, 3 } }; // P4

int avail[3] = { 3, 3, 2 }; // Available Resources

int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++) {
    f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++)
        need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
    for (i = 0; i < n; i++) {
        if (f[i] == 0) {

            int flag = 0;
            for (j = 0; j < m; j++) {
                if (need[i][j] > avail[j]){
                    flag = 1;
                     break;
                }
            }
```

```
            if (flag == 0) {
                ans[ind++] = i;
                for (y = 0; y < m; y++)
                    avail[y] += alloc[i][y];
                f[i] = 1;
            }
        }
    }
}

  int flag = 1;

  for(int i=0;i<n;i++)
{
  if(f[i]==0)
  {
    flag=0;
     printf("The following system is not safe");
    break;
  }
}

  if(flag==1)
{
  printf("Following is the SAFE Sequence\n");
  for (i = 0; i < n - 1; i++)
    printf(" P%d ->", ans[i]);
  printf(" P%d", ans[n - 1]);
}


return (0);


}
```

Output:
A) Banker's:

```
PS C:\Users\arhaa\OneDrive\Desktop\OS> cd "c:\Users\arhaa\OneDrive\
 .\bankers }
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
PS C:\Users\arhaa\OneDrive\Desktop\OS>
```

Theory:

B) Dining Philosophers Problem:

Dining Philosophers Problem States that there are 5 Philosophers who are engaged in two activities: Thinking and Eating. Meals are taken communally in a table with five plates and five forks in a cyclic manner as shown in the figure.

Constraints and Condition for the problem :

1. Every Philosopher needs two forks in order to eat.
2. Every Philosopher may pick up the forks on the left or right but only one fork at once.
3. Philosophers only eat when they had two forks. We have to design such a protocol i.e. pre and post protocol which ensures that a philosopher only eats if he or she had two forks.
4. Each fork is either clean or dirty.
5.

Solution :

Correctness properties it needs to satisfy are :Mutual Exclusion Principle –

No two Philosophers can have the two forks simultaneously.

1. Free from Deadlock –
   Each philosopher can get the chance to eat in a certain finite time.
2. Free from Starvation –When few Philosophers are waiting then one gets a chance to eat in a while.
3. No strict Alternation.
4. Proper utilization of time.

Algorithm(outline) :

loop forever

   p1: think

p2: preprotocol
p3: eat
p4: postprotocol

First Attempt :

We assume that each philosopher is initialized with its index I and that addition is implicitly modulo 5. Each fork is modeled as a semaphore where wait corresponds to taking a fork and signal corresponds to putting down a fork.

Algorithm –

```
semaphore array[0..4] fork ← [1, 1, 1, 1, 1]
loop forever
    p1 : think
    p2 : wait(fork[i])
    p3 : wait(fork[i + 1])
    p4 : eat
    p5 : signal(fork[i])
    p6 : signal(fork[i + 1])
```

Problem with this solution :

This solution may lead to a deadlock under an interleaving that has all the philosophers pick up their left forks before any of them tries to pick up a right fork. In this case, all the Philosophers are waiting for the right fork but no one will execute a single instruction.

Second Attempt :

One way to tackle the above situation is to limit the number of philosophers entering the room to four. By doing this one of the philosophers will eventually get both the fork and execute all the instruction leading to no deadlock.

Algorithm –

```
semaphore array[0..4] fork ← [1, 1, 1, 1, 1]

  semaphore room ← 4
  loop forever
    p1 : think
    p2 : wait(room)
    p3 : wait(fork[i])
    p4 : wait(fork[i + 1])
    p5 : eat
```

```
p6 : signal(fork[i])
p7 : signal(fork[i + 1])
p8 : signal(room)
```

In this solution, we somehow interfere with the given problem as we allow only four philosophers.

Third Attempt :

We use the asymmetric algorithm in the attempt where the first four philosophers execute the original solution but the fifth philosopher waits for the right fork and then for the left fork.

Algorithm –

semaphore array [0..4] fork ← [1,1,1,1,1]

For the first four philosophers –

```
loop forever
  p1 : think
  p2 : wait(fork[i])
  p3 : wait(fork[i + 1])
  p4 : eat
  p5 : signal(fork[i])
  p6: signal(fork[i + 1])
```

For the fifth philosopher –

```
loop forever
  p1 : think
  p2 : wait(fork[0])
  p3 : wait(fork[4])
  p4 : eat
  p5 : signal(fork[0])
  p6 : signal(fork[4])
```

Note –

This solution is also known as Chandy/Mishra Solution.

Advantages of this Solution :

1. Allows a large degree of concurrency.
2. Free from Starvation.
3. Free from Deadlock.
4. More Flexible Solution.
5. Economical
6. Fairness

7. Boundedness.

The above discussed the solution for the problem using semaphore. Now with monitors, Here, Monitor maintains an array of the fork which counts the number of free forks available to each philosopher. The take Forks operation waits on a condition variable until two forks are available. It decrements the number of forks available to its neighbor before leaving the monitor. After eating, a philosopher calls release Forks which updates the array fork and checks if freeing these forks makes it possible to signal.

Algorithm –

```
monitor ForkMonitor:
integer array[0..4]
fork ← [2,2,2,2,2]
condition array[0..4]OKtoEat

operation takeForks(integer i)
if(fork[i]!=2)
waitC(OKtoEat[i])

fork[i+1]<- fork[i+1]-1
fork[i-1] <- fork[i-1]-1

operation releaseForks(integer i)
fork[i+1] <- fork[i+1]+1
fork[i-1] <- fork[i-1]

if(fork[i+1]==2)
signalC(OKtoEat[i+1])

if(fork[i-1]==2)
signalC(OKtoEat[i-1])
```

For each Philosopher –

```
loop forever :
    p1 : think
    p2 : takeForks(i)
    p3 : eat
    p4 : releaseForks(i)
```

Program:

B) Dining Philosopher:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define NUM_PHILOSOPHERS 5
#define NUM_CHOPSTICKS 5

void dine(int n);
pthread_t philosopher[NUM_PHILOSOPHERS];
pthread_mutex_t chopstick[NUM_CHOPSTICKS];

int main()
{
  // Define counter var i and status_message
  int i, status_message;
  void *msg;

  // Initialise the semaphore array
  for (i = 1; i <= NUM_CHOPSTICKS; i++)
  {
    status_message = pthread_mutex_init(&chopstick[i], NULL);
    // Check if the mutex is initialised successfully
    if (status_message == -1)
    {
      printf("\n Mutex initialization failed");
      exit(1);
    }
  }

  // Run the philosopher Threads using *dine() function
  for (i = 1; i <= NUM_PHILOSOPHERS; i++)
  {
```

```
    status_message = pthread_create(&philosopher[i], NULL, (void *)dine,
(int *)i);
    if (status_message != 0)
    {
      printf("\n Thread creation error \n");
      exit(1);
    }
  }

  // Wait for all philosophers threads to complete executing (finish dining)
before closing the program
  for (i = 1; i <= NUM_PHILOSOPHERS; i++)
  {
    status_message = pthread_join(philosopher[i], &msg);
    if (status_message != 0)
    {
      printf("\n Thread join failed \n");
      exit(1);
    }
  }

  // Destroy the chopstick Mutex array
  for (i = 1; i <= NUM_CHOPSTICKS; i++)
  {
    status_message = pthread_mutex_destroy(&chopstick[i]);
    if (status_message != 0)
    {
      printf("\n Mutex Destroyed \n");
      exit(1);
    }
  }
  return 0;
}
void dine(int n)
{
  printf("\nPhilosopher % d is thinking ", n);
```

```
  // Philosopher picks up the left chopstick (wait)
  pthread_mutex_lock(&chopstick[n]);

  // Philosopher picks up the right chopstick (wait)
  pthread_mutex_lock(&chopstick[(n + 1) % NUM_CHOPSTICKS]);

  // After picking up both the chopstick philosopher starts eating
  printf("\nPhilosopher % d is eating ", n);
  sleep(3);
  // Philosopher places down the left chopstick (signal)
  pthread_mutex_unlock(&chopstick[n]);
  // Philosopher places down the right chopstick (signal)
  pthread_mutex_unlock(&chopstick[(n + 1) % NUM_CHOPSTICKS]);
  // Philosopher finishes eating
  printf("\nPhilosopher % d Finished eating ", n);
}
```

Output:

B)Dining Philosopher:

```
Philosopher  5 is thinking
Philosopher  5 is eating
Philosopher  2 is thinking
Philosopher  2 is eating
Philosopher  1 is thinking
Philosopher  3 is thinking
Philosopher  4 is thinking
Philosopher  4 is eating
Philosopher  5 Finished eating
Philosopher  1 is eating
Philosopher  2 Finished eating
Philosopher  4 Finished eating
Philosopher  3 is eating
Philosopher  3 Finished eating
Philosopher  1 Finished eating

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: Thus we have successfully implemented Banker's algorithm for deadlock avoidance and Dining Philosophers Problem.