Write a C program to simulate the concept of Dining-philosopher problem

```
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#define N 5
#define HUNGRY 1
#define EATING 0
#define THINKING 2
#define LEFT (phnum+4)%N
#define RIGHT (phnum+1)%N

int state[N];
int phil[N] = {0,1,2,3,4};

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
    if (state[phnum] == HUNGRY
    && state[LEFT] != EATING
    && state[RIGHT] != EATING)
    {
    state[phnum] = EATING;

    sleep(2);
    printf("Philosopher %d takes fork %d and
    %d\n", phnum+1, LEFT+1, phnum+1);
    printf("Philosopher %d is Eating\n", phnum+1
    sem_post(&S[phnum]); }}
void take_fork(int phnum)
{
```

```
sem_wait (& mutex);
state [phnum] = HUNGRY;
printf ("philosopher %d is HUNGRY \n", phnum+1);
test (phnum);
sem_post (& mutex);
sem_wait (& s[phnum]);
sleep[1];
}
void put_fork (int phnum)
{
sem_wait (& mutex);
state [phnum] = THINKING;
printf ("philosopher %d putting fork %d and
%d down \n", phnum+1, LEFT+1, phnum+1);
printf ("philosopher %d is THINKING \n", phnum+1);
test (LEFT);
test (RIGHT);
sem_post (& mutex);
}
void* philospher (void* num
{
while (1) {
int *i = num;
sleep(1);
take_fork (*i);
sleep (0);
put_fork (*i);
} }
```

```
int main()
{ int i;
pthread_t thread_id[N];
sem_init(&mutex, 0, 1)
for(i=0; i<n; i++)
sem_init(&s[i], 0, 0);
for(i=0; i<N; i++){
  pthread_Create(&thread_id[i], NULL, philospher,
  &phil[i]);
  printf("philosopher %d is thinking\n", i+1);
}
for(i=0; i<N; i++)
pthread_Join(thread_id[i], NULL);
}
```

Output

philosopher 1 is thinking
philosopher 2 is thinking
philosopher 3 is thinking
philosopher 4 is thinking
philosopher 2 is HUNGRY
philosopher 1 is HUNGRY
philosopher 4 is HUNGRY
philosopher 5 is HUNGRY
philosopher 5 takes fork 4 and 5
philosopher 5 is eating EATING
philosopher 3 is HUNGRY
philosopher 3 takes fork 2 and 3
philosopher 3 is EATING
philosopher 5 putting fork 4 and 5 down
philosopher 5 is THINKING.

10/10

23/8/23

Lab - 7

1). Write a C program to stimulate deadlock detection

```
#include <stdio.h>
#define MAX process 10
#define MAX- Resources 10

int process, resources;
int allocation [MAX- process] [MAX- Resources];
int max_need [MAX - process] [MAX - Resources];
int available [MAX- Resources];
int marked [MAX - Proces];
int finished [MAX - process]

Void initalise () {
printf ("Enter the no of process: ");
scanf ("%d", & procas);
printf ("Enter the no of resources: ");
Scanf ("%d", & resource);
printf ("Enter the allocation matrix \n");
  for (i=0; i < process; i++)
  {
    for (j=0; j < resources; j++){
      scanf ("%d", & allocate [i][j]);
  }
}
printf ("Enter the max need matrix \n");
  for (i=0; i < process; i++)
  {
    for (j=0; j < resources; j++)
      scanf ("%d", max [i][j]);
  }
}
```

```
printf ("Enter the available resources : \n");
for (i=0; i < resources ; i++)
{ scanf ("%d", & available [i]);
}
}
void detect deadlock()
{ for (i=0 ; i < process ; i++)
{ marked[i] = 0;
finished[i] = 0;
}
int marked_count = 0;
while (marked_count < process)
{ if (! finished[i] && ! marked[i])
{
int can_allocate = 1;
for (j=0; j < resources; j++)
{ if (max_need[i][j] - allocation[i][j] > available[j]
{ can_allocate = 0;
break;
}
if (can_allocate)
{ marked[i] = 1;
marked_count++;
found = 1;
for (j=0; j < resource ; j++)
{ available[j] += allocation[i][j];
}
break;
}
}
```

```
if (!found) {
    printf("Deadlock detected! :\n");
    for (i=0; i<process; i++)
    {
        if (!finished[i] && !marked[i]){
            printf("process %d \n", i);
        }
    }
    return;
  }
}
printf("no deadlock detected \n");
}
int main()
{ initialize();
  detect deadlock();
  return 0;
}
```

Output:

Enter the no of process: 5
Enter the no of resource: 3
Enter the allocation matrix:

```
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
```

Enter the max need matrix:

```
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
```

Enter the available resources: 3 3 2

no deadlock detected

10/10