Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_BOT THANOS]

Experiment Title: DEAD LOCKS

Aim/Objective:

Students should be able to understand and apply the concept of deadlocks.

Description:

Deadlock is a situation where more than one process is blocked because it is holding a resource and also requires some resource that is acquired by some other process. There are Four necessary conditions in a deadlock situation to occur mutual execution, hold and wait, no-pre-emption, and circular wait.

Prerequisite:

- **Basic functionality of Deadlocks.**
- **Complete idea of Deadlock avoidance and Prevention**

Pre-Lab Task:

Deadlock Conditions	FUNCTIONALITY
Mutual Exclusion	Only one process can access resource at a time.
Hold and Wait	Process holding resources waits for additional resources.

Deadlock Conditions	**FUNCTIONALITY**
No Preemption	Resources cannot be forcibly taken from a process.
Circular Wait	A set of processes form a circular chain of waits.
Dead Lock	A situation where processes cannot proceed due to resource locking.

In Lab Task:

1. Write a C program to simulate the Bankers Algorithm for Deadlock Avoidance.

```
#include <stdio.h>

int main() {
    int processes, resources;

    printf("Enter the number of processes: ");
    scanf("%d", &processes);

    printf("Enter the number of resources: ");
    scanf("%d", &resources);

    int allocation[processes][resources];
    int maximum[processes][resources];
    int need[processes][resources];
    int available[resources];
    int finish[processes];

    printf("Enter the allocation matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }

    printf("Enter the maximum matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            scanf("%d", &maximum[i][j]);
            need[i][j] = maximum[i][j] - allocation[i][j];
        }
    }

    printf("Enter the available resources: ");
    for (int i = 0; i < resources; i++) {
        scanf("%d", &available[i]);
    }
```

```
for (int i = 0; i < processes; i++) {
        finish[i] = 0;
    }

    int safeSeq[processes];
    int safeSeqIdx = 0;
    int work[resources];

    for (int i = 0; i < resources; i++) {
        work[i] = available[i];
    }

    int count = 0;

    while (count < processes) {
        int found = 0;

        for (int p = 0; p < processes; p++) {
            if (finish[p] == 0) {
                int canAllocate = 1;

                for (int r = 0; r < resources; r++) {
                    if (need[p][r] > work[r]) {
                        canAllocate = 0;
                        break;
                    }
                }

                if (canAllocate) {
                    for (int r = 0; r < resources; r++) {
                        work[r] += allocation[p][r];
                    }

                    safeSeq[safeSeqIdx] = p;
                    safeSeqIdx++;
                    finish[p] = 1;
                    found = 1;
                }
            }
        }
```

```
        if (found == 0) {
        printf("The system is not in a safe state.\n");
        break;
      }

      count++;
    }

    if (safeSeqIdx == processes) {
      printf("Safe sequence: ");
      for (int i = 0; i < processes; i++) {
        printf("%d", safeSeq[i]);
        if (i < processes - 1) {
          printf(" -> ");
        }
      }
      printf("\n");
    }

    return 0;
}
```

Data and Results

DATA:

- Processes: 4
- Resources: 3
- Allocation matrix, Maximum matrix, and Available resources provided.

RESULT:

Safe sequence: P0 -> P2 -> P3 -> P1

Analysis and Inferences

ANALYSIS:

The system followed Banker's Algorithm, ensuring resource allocation avoids deadlocks effectively.

INFERENCES:

System is in a safe state with valid sequence.

2. Write a C program to simulate Bankers Algorithm for Deadlock Prevention.

```
#include <stdio.h>

int main() {

    int processes, resources;

    printf("Enter the number of processes: ");
    scanf("%d", &processes);

    printf("Enter the number of resources: ");
    scanf("%d", &resources);

    int allocation[processes][resources];
    int max[processes][resources];
    int need[processes][resources];
    int available[resources];
    int work[resources];
    int finish[processes];
    int safeSeq[processes];
    int safeSeqIdx = 0;

    printf("Enter the allocation matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }

    printf("Enter the maximum matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            scanf("%d", &max[i][j]);
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }

    printf("Enter the available resources:\n");
    for (int i = 0; i < resources; i++) {
```

```
     scanf("%d", &available[i]);
    work[i] = available[i];
}

for (int i = 0; i < processes; i++) {
    finish[i] = 0;
}

int count = 0;

while (count < processes) {
    int found = 0;

    for (int i = 0; i < processes; i++) {
        if (finish[i] == 0) {
            int canAllocate = 1;

            for (int j = 0; j < resources; j++) {
                if (need[i][j] > work[j]) {
                    canAllocate = 0;
                    break;
                }
            }

            if (canAllocate) {
                for (int j = 0; j < resources; j++) {
                    work[j] += allocation[i][j];
                }
                safeSeq[safeSeqIdx] = i;
                safeSeqIdx++;
                finish[i] = 1;
                found = 1;
            }
        }
    }

    if (found == 0) {
        printf("The system is not in a safe state. Deadlock detected.\n");
        break;
    }

    count++;
```

```
 }

  if (safeSeqIdx == processes) {
    printf("Safe sequence: ");
    for (int i = 0; i < processes; i++) {
      printf("%d", safeSeq[i]);
      if (i < processes - 1) {
        printf(" -> ");
      }
    }
    printf("\n");
  }

  return 0;
}
```

1. Write a C program to simulate to implement of the Shared memory and IPC.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/types.h>

#define SHM_KEY 12345
#define SEM_KEY 54321
#define MAX_COUNT 10

void sem_wait(int sem_id) {
   struct sembuf sb;
   sb.sem_num = 0;
   sb.sem_op = -1;
   sb.sem_flg = 0;
   semop(sem_id, &sb, 1);
}

void sem_signal(int sem_id) {
   struct sembuf sb;
   sb.sem_num = 0;
   sb.sem_op = 1;
   sb.sem_flg = 0;
   semop(sem_id, &sb, 1);
}

int main() {

   int shmid, semid;
   int *shared_data;
   int count = 0;

   if ((shmid = shmget(SHM_KEY, sizeof(int), IPC_CREAT | 0666)) == -1) {
      perror("shmget");
      exit(1);
   }
```

```
    shared_data = (int *)shmat(shmid, NULL, 0);

    semid = semget(SEM_KEY, 1, IPC_CREAT | 0666);
    semctl(semid, 0, SETVAL, 1);

    while (count < MAX_COUNT) {
        sem_wait(semid);
        (*shared_data)++;
        printf("Process %d writes: %d\n", getpid(), *shared_data);
        sem_signal(semid);
        count++;
        sleep(1);
    }

    shmdt(shared_data);
    shmctl(shmid, IPC_RMID, NULL);
    semctl(semid, 0, IPC_RMID);

    return 0;
}
```

2. Write a program to simulate Threading and Synchronization Applications. in C

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

#define NUM_THREADS 3
#define NUM_ITERATIONS 5

int shared_counter = 0;
pthread_mutex_t mutex;

void* thread_function(void* arg) {
   int thread_id = *((int*)arg);
   for (int i = 0; i < NUM_ITERATIONS; i++) {
      pthread_mutex_lock(&mutex);
      shared_counter++;
      printf("Thread %d: Incremented counter to %d\n", thread_id, shared_counter);
      pthread_mutex_unlock(&mutex);
   }
   pthread_exit(NULL);
}

int main() {
   pthread_t threads[NUM_THREADS];
   int thread_ids[NUM_THREADS];
   pthread_mutex_init(&mutex, NULL);

   for (int i = 0; i < NUM_THREADS; i++) {
      thread_ids[i] = i;
      int result = pthread_create(&threads[i], NULL, thread_function, &thread_ids[i]);
      if (result != 0) {
         perror("pthread_create");
         return 1;
      }
   }

   for (int i = 0; i < NUM_THREADS; i++) {
      pthread_join(threads[i], NULL);
   }
```

```
    pthread_mutex_destroy(&mutex);
    printf("Final shared counter value: %d\n", shared_counter);
    return 0;
}
```

Sample VIVA-VOCE Questions (In-Lab):

1. What is the use of Dead Locks?

   Deadlocks prevent resource conflicts and ensure system stability.

2. Differentiate between Deadlock Prevention and Deadlock Avoidance.

   - **Prevention**: Ensures deadlock can't occur.
   - **Avoidance**: Allows but avoids unsafe resource allocation.

3. Explain in detail Bankers Algorithm.

   Allocates resources based on safety to prevent deadlock.

4. Explain in detail Mutual Exclusion.

Prevents simultaneous access to shared resources to avoid conflicts.

5. Explain in detail the Methods of Deadlock Handling.

Prevention, avoidance, detection, and recovery methods ensure system stability.

Evaluator Remark (if any):	**Marks Secured_____ out of 50**
	Signature of the Evaluator with Date

Note: Evaluator MUST ask Viva-voce before signing and posting marks for each experiment.