OPERATING SYSTEMS
Practical 07

Name: Arya Narendra Narlawar
Roll No: 30
Batch: B2

Aim: Write C programs to implement threads and semaphores for process synchronization.

Code:

1. Matrix Multiplication

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#define M 3
#define K 2
#define N 3
int A [M][K] = { {1,4}, {2,5}, {3,6} };
int B [K][N] = { {8,7,6}, {5,4,3} };
int C [M][N];
struct v {
int i; /* row */
int j; /* column */
};
void *runner(void *param); /* the thread */
int main()
{
int i,j;
/* Now create the thread passing it data as a parameter */
pthread_t tid; //Thread ID
pthread_attr_t attr; //Set of thread attributes
//Get the default attributes
pthread_attr_init(&attr);
for(i = 0; i < M; i++)
{
 for(j = 0; j < N; j++)
 {
//Assign a row and column for each thread
struct v *data = (struct v *) malloc(sizeof(struct v));
data->i = i;
data->j = j;
//Create the thread
pthread_create(&tid,&attr,
/Make sure the parent waits for all threads to complete
pthread_join(tid, NULL);
}
```

```
}
//Print out the resulting matrix
for(i = 0; i < M; i++)
{
 for(j = 0; j < N; j++)
{
printf("%d ", C[i][j]);
}
printf("\n");
}
}

//The thread will begin control in this function void
*runner(void *param)
{
struct v *data = param; // the structure that holds our data int
n, sum = 0; //the counter and sum
//Row multiplied by column
for(n = 0; n< K; n++)
{
sum += A[data->i][n] * B[n][data->j];
}
//assign the sum to its coordinate
C[data->i][data->j] = sum;
//Exit the thread
pthread_exit(0);
}
```

Output:

2. Producer Consumer

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


#define BUFFER_SIZE 5

int buf[BUFFER_SIZE], f = -1, r = -1;
```

```
sem_t mutex, full, empty;
void *produce(void *arg) {
int i;
for (i = 0; i < 10; i++) {
 sem_wait(&empty);
 sem_wait(&mutex);
 printf("Produced item is %d\n", i);
 buf[(++r) % BUFFER_SIZE] = i;
 sleep(1);
 sem_post(&mutex);
 sem_post(&full);
}
}

void *consume(void *arg) {
int item, i;
for (i = 0; i < 10; i++) {
 sem_wait(&full);
 sem_wait(&mutex);
 item = buf[(++f) % BUFFER_SIZE];
printf("Consumed item is %d\n", item);
sleep(1);
 sem_post(&mutex);
 sem_post(&empty);
}
}

int main() {
pthread_t tid1, tid2;
sem_init(&mutex, 0, 1);
sem_init(&full, 0, 0);
sem_init(&empty, 0, BUFFER_SIZE);

pthread_create(&tid1, NULL, produce, NULL);
pthread_create(&tid2, NULL, consume, NULL);
pthread_join(tid1, NULL);
pthread_join(tid2, NULL);

sem_destroy(&mutex);
sem_destroy(&full);
sem_destroy(&empty);

return 0;
}
```

Output:

3. Reader- Writer

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

#define NUM_READERS 5
#define NUM_WRITERS 2
#define MAX_ITERATIONS 10

int shared_data = 0;
int active_readers = 0;
int waiting_readers = 0;
int active_writers = 0;
int waiting_writers = 0;
int iterations = 0;

sem_t mutex, rw_mutex, iteration_mutex;

void *reader(void *arg) {
   int reader_id = *((int *) arg);
   while (1) {
      // Entry section
      sem_wait(&mutex);
      waiting_readers++;
      if (active_writers > 0 || waiting_writers > 0) {
         sem_post(&mutex);
          sem_wait(&rw_mutex);
          sem_post(&mutex);
      } else {
```

```
            sem_post(&mutex);
        }

        // Critical section
        printf("Reader %d reads shared data: %d\n", reader_id, shared_data);

        // Exit section
        sem_wait(&mutex);
        waiting_readers--;
        active_readers++;
        sem_post(&mutex);

        // Other processing
        // ...

        // Remainder section
        sem_wait(&mutex);
        active_readers--;
        if (active_readers == 0 && waiting_writers > 0) {
            sem_post(&rw_mutex);
        }
        sem_post(&mutex);

        // Other processing
        // ...

        // Sleep for a while
        usleep(rand() % 500000);

        // Check termination condition
        sem_wait(&iteration_mutex);
        iterations++;
        int current_iterations = iterations;
        sem_post(&iteration_mutex);

        if (current_iterations >= MAX_ITERATIONS) {
            break;
        }
    }
    pthread_exit(NULL);
}

void *writer(void *arg) {
    int writer_id = *((int *) arg);
    while (1) {
        // Entry section
        sem_wait(&mutex);
        waiting_writers++;
        if (active_readers > 0 || active_writers > 0) {
            sem_post(&mutex);
            sem_wait(&rw_mutex);
            sem_post(&mutex);
```

```
        } else {
            sem_post(&mutex);
        }

        // Critical section
        shared_data++;
        printf("Writer %d writes shared data: %d\n", writer_id, shared_data);

        // Exit section
        sem_wait(&mutex);
        waiting_writers--;
        active_writers++;
        sem_post(&mutex);

        // Other processing
        // ...

        // Remainder section
        sem_wait(&mutex);
        active_writers--;
        if (waiting_writers > 0) {
            sem_post(&rw_mutex);
        } else if (waiting_readers > 0) {
            int readers_to_release = (waiting_readers < NUM_READERS) ? waiting_readers :
NUM_READERS;
            sem_post(&rw_mutex);
            sem_post(&rw_mutex);
            sem_post(&rw_mutex);
            sem_post(&rw_mutex);
        }
        sem_post(&mutex);

        // Other processing
        // ...

        // Sleep for a while
        usleep(rand() % 500000);

        // Check termination condition
        sem_wait(&iteration_mutex);
        iterations++;
        int current_iterations = iterations;
        sem_post(&iteration_mutex);

        if (current_iterations >= MAX_ITERATIONS) {
            break;
        }
    }
    pthread_exit(NULL);
}
int main() {
    srand(time(NULL));
```

```
    sem_init(&mutex, 0, 1);
    sem_init(&rw_mutex, 0, 1);
    sem_init(&iteration_mutex, 0, 1);

    pthread_t readers[NUM_READERS];
    pthread_t writers[NUM_WRITERS];

    int i;
    for (i = 0; i < NUM_READERS; i++) {
        int *reader_id = malloc(sizeof(int));
        *reader_id = i + 1;
        pthread_create(&readers[i], NULL, reader, (void *) reader_id);  }

    for (i = 0; i < NUM_WRITERS; i++) {
        int *writer_id = malloc(sizeof(int));
        *writer_id = i + 1;
        pthread_create(&writers[i], NULL, writer, (void *) writer_id);  }

    for (i = 0; i < NUM_READERS; i++) {
        pthread_join(readers[i], NULL);
    }

    for (i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writers[i], NULL);
    }

    sem_destroy(&mutex);
    sem_destroy(&rw_mutex);
    sem_destroy(&iteration_mutex);

    return 0;
}
```

Output: