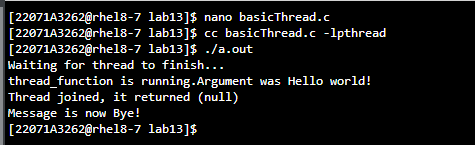
**Week – 13**

Implement the classic problem of synchronization using semaphore and monitor

**Code: (Basic Thread)**

#include<stdio.h>  
#include<unistd.h>  
#include<stdlib.h>  
#include<string.h>  
#include<pthread.h>  
  
void \*thread\_function(void \*arg);  
char message[] = "Hello world!";  
  
int main(){  
    int res;  
    pthread\_t a\_thread;  
    void \*thread\_result;  
    res = pthread\_create(&a\_thread,NULL,thread\_function,(void \*)message);  
    if(res != 0){  
        perror("Thread creation failed");  
        exit(EXIT\_FAILURE);  
    }  
    printf("Waiting for thread to finish...\n");  
    res = pthread\_join(a\_thread,&thread\_result);  
    if(res != 0){  
        perror("Thread join failed");  
        exit(EXIT\_FAILURE);  
    }  
    printf("Thread joined, it returned %S\n",(char \*)thread\_result);  
    printf("Message is now %s\n",message);  
    exit(EXIT\_SUCCESS);  
    }  
void \*thread\_function(void \*arg){  
        printf("thread\_function is running.Argument was %s\n",(char \*)arg);  
        sleep(3);  
        strcpy(message,"Bye!");  
}

**Output:**



**Code (Producer & Consumer - 1)**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define SHARED 1

void \*Producer();

void \*Consumer();

sem\_t empty, full, sm;

int data;

int main() {

pthread\_t ptid, ctid;

printf("\nMain Started");

sem\_init(&empty, SHARED, 1);

sem\_init(&full, SHARED, 0);

sem\_init(&sm, SHARED, 1);

pthread\_create(&ptid, NULL, Producer, NULL);

pthread\_create(&ctid, NULL, Consumer, NULL);

pthread\_join(ptid, NULL);

pthread\_join(ctid, NULL);

printf("\nMain done\n");

return 0;

}

void \*Producer() {

int produced;

printf("\nProducer created");

for (produced = 0; produced < 100; produced++) {

sem\_wait(&empty);

sem\_wait(&sm);

data = produced;

sem\_post(&sm);

sem\_post(&full);

printf("\nProducer: %d", data);

}

return NULL;

}

void \*Consumer() {

int consumed, total = 0;

printf("\nConsumer created");

for (consumed = 0; consumed < 100; consumed++) {

sem\_wait(&full);

sem\_wait(&sm);

total += data;

sem\_post(&sm);

sem\_post(&empty);

printf("\nConsumed: %d", data);

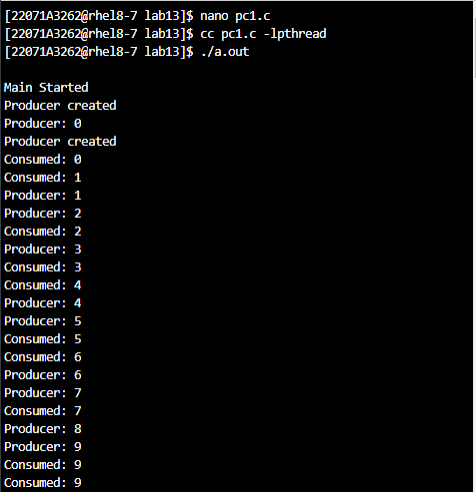
}

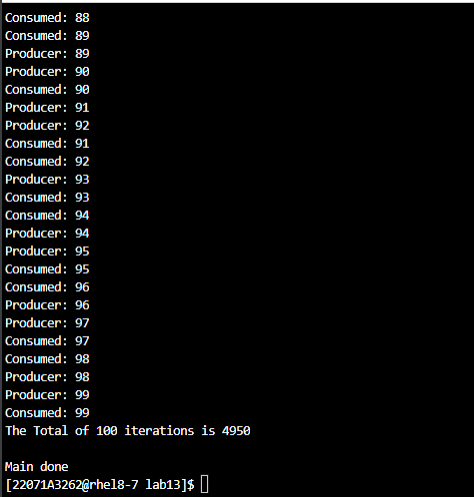
printf("\nThe Total of 100 iterations is %d\n", total);

return NULL;

}

**Output:**





**Code (Producer & Consumer - 2)**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

void \*producer();

void \*consumer();

sem\_t empty, sm, full;

int data, in = 0, out = 0;

int buffer[5];

int main() {

pthread\_t ptid[5], ctid[5];

printf("\nMain started");

sem\_init(&empty, 0, 5);

sem\_init(&full, 0, 0);

sem\_init(&sm, 0, 1);

for (int i = 0; i < 5; i++) {

pthread\_create(&ctid[i], NULL, consumer, NULL);

pthread\_create(&ptid[i], NULL, producer, NULL);

}

for (int i = 0; i < 5; i++) {

pthread\_join(ctid[i], NULL);

pthread\_join(ptid[i], NULL);

}

printf("\nMain done\n");

return 0;

}

void \*producer() {

int produced;

printf("\nProducer created");

sem\_wait(&empty);

sem\_wait(&sm);

printf("Enter a number: ");

scanf("%d", &produced);

buffer[in] = produced;

sem\_post(&sm);

sem\_post(&full);

printf("\nProducer: %d", buffer[in]);

in = (in + 1) % 5;

return NULL;

}

void \*consumer() {

int total = 0;

printf("\nConsumer created");

sem\_wait(&full);

sem\_wait(&sm);

total += buffer[out];

sem\_post(&sm);

sem\_post(&empty);

printf("\nConsumed: %d", buffer[out]);

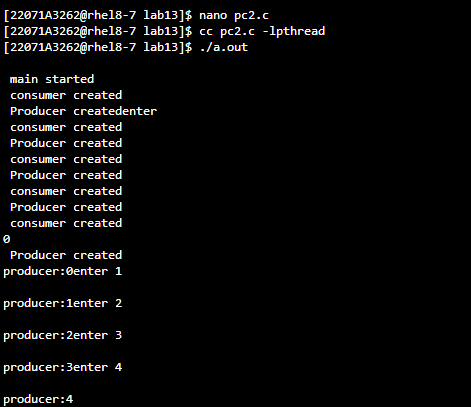
out = (out + 1) % 5;

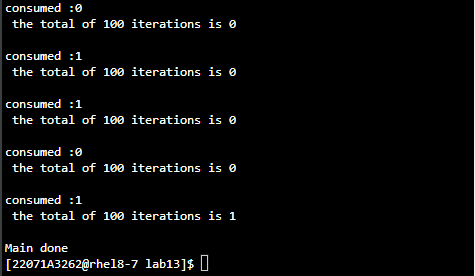
printf("\nThe total consumed value is %d\n", total);

return NULL;

}

**Output:**



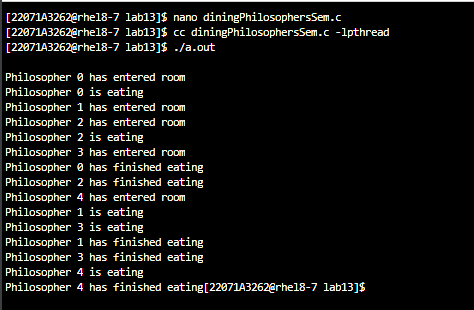


**Code for Dining Philosophers using semaphores:**

#include<stdio.h>  
#include<unistd.h>  
#include<semaphore.h>  
#include<pthread.h>  
  
sem\_t room;  
sem\_t chopstick[5];  
void \*philosopher(void \*);  
void eat(int);  
  
int main(){  
    sem\_init(&room,0,4);  
    for(int i=0;i<5;i++){  
        sem\_init(&chopstick[i],0,1);  
    }

int a[5];  
    pthread\_t tid[5];  
    for(int i=0;i<5;i++){  
        a[i] = i;  
        pthread\_create(&tid[i],NULL,philosopher,(void \*)&a[i]);  
    }  
    for(int i=0;i<5;i++)  
        pthread\_join(tid[i],NULL);  
    return 0;  
}  
  
void \*philosopher(void \*num){  
    int phil = \*(int \*)num;  
    sem\_wait(&room);  
    printf("\nPhilosopher %d has entered room",phil);  
    sem\_wait(&chopstick[phil]);  
    sem\_wait(&chopstick[(phil+1)%5]);  
    eat(phil);  
    sleep(2);  
    printf("\nPhilosopher %d has finished eating",phil);  
    sem\_post(&chopstick[(phil+1)%5]);  
    sem\_post(&chopstick[phil]);  
    sem\_post(&room);  
}  
  
void eat(int phil){  
    printf("\nPhilosopher %d is eating",phil);  
}

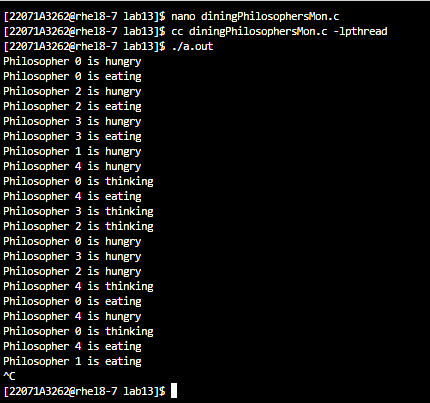
**Output:**



**Code for Dining Philosophers using monitors:**

#include<stdio.h>  
#include<pthread.h>  
#include<unistd.h>  
#define N 5  
pthread\_mutex\_t mutex;  
pthread\_cond\_t cond\_var[N];  
enum {THINKING, HUNGRY, EATING} state[N];  
  
int left(int i){  
    return (i+N-1)&N;  
}  
  
int right(int i){  
    return (i+1)%N;  
}  
  
void test(int i){  
    if(state[i] == HUNGRY && state[left(i)] != EATING && state[right(i)] != EATING){  
        state[i] = EATING;  
        printf("Philosopher %d is eating\n",i);  
        pthread\_cond\_signal(&cond\_var[i]);  
    }  
}  
  
void pickup\_forks(int i){  
    pthread\_mutex\_lock(&mutex);  
    state[i] = HUNGRY;  
    printf("Philosopher %d is hungry\n",i);  
    test(i);  
    if(state[i] != EATING)  
        pthread\_cond\_wait(&cond\_var[i],&mutex);  
    pthread\_mutex\_unlock(&mutex);  
}  
  
void return\_forks(int i){  
    pthread\_mutex\_lock(&mutex);  
    state[i] =  THINKING;  
    printf("Philosopher %d is thinking\n",i);  
    test(left(i));  
    test(right(i));  
    pthread\_mutex\_unlock(&mutex);  
}  
  
void \*philosopher(void \*arg){  
    int \*id = (int \*)arg;  
    while(1){  
        usleep(1000000);  
        pickup\_forks(\*id);  
        usleep(1000000);  
        return\_forks(\*id);  
    }  
    return NULL;  
}  
  
int main(){  
    pthread\_t tid[N];  
    int ids[N];  
    pthread\_mutex\_init(&mutex,NULL);  
    for(int i=0;i<N;i++){  
        pthread\_cond\_init(&cond\_var[i],NULL);  
        ids[i] = i;  
        pthread\_create(&tid[i],NULL,philosopher,&ids[i]);  
    }  
    for(int i=0;i<N;i++){  
        pthread\_join(tid[i],NULL);  
    }  
    pthread\_mutex\_destroy(&mutex);  
    for(int i=0;i<N;i++){  
        pthread\_cond\_destroy(&cond\_var[i]);  
    }  
    return 0;  
}

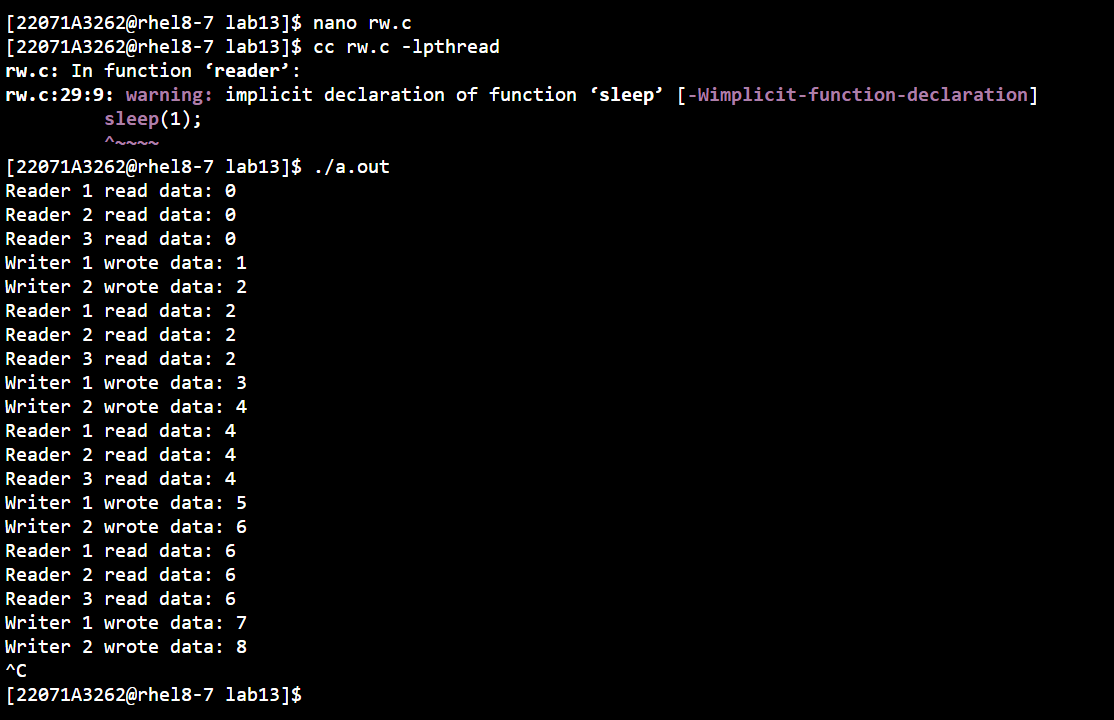
**Output:**



**Code for readers writers’ problem:**

#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include <semaphore.h>  
  
#define NUM\_READERS 3  
#define NUM\_WRITERS 2  
  
sem\_t mutex, writeblock;  
int data = 0, readers\_count = 0;  
  
void \*reader(void \*arg) {  
    int reader\_id = \*((int\*)arg);  
    while (1) {  
        sem\_wait(&mutex);  
        readers\_count++;  
        if (readers\_count == 1)  
            sem\_wait(&writeblock);  
        sem\_post(&mutex);  
  
        printf("Reader %d read data: %d\n", reader\_id, data);  
  
        sem\_wait(&mutex);  
        readers\_count--;  
        if (readers\_count == 0)  
            sem\_post(&writeblock);  
        sem\_post(&mutex);  
  
        sleep(1);  
    }  
}  
  
void \*writer(void \*arg) {  
    int writer\_id = \*((int\*)arg);  
    while (1) {  
        sem\_wait(&writeblock);  
  
        data++;  
        printf("Writer %d wrote data: %d\n", writer\_id, data);  
  
        sem\_post(&writeblock);  
  
        sleep(1);  
    }  
}  
  
int main() {  
    pthread\_t readers[NUM\_READERS], writers[NUM\_WRITERS];  
    int reader\_ids[NUM\_READERS], writer\_ids[NUM\_WRITERS];  
     
    sem\_init(&mutex, 0, 1);  
    sem\_init(&writeblock, 0, 1);  
     
    for (int i = 0; i < NUM\_READERS; i++) {  
        reader\_ids[i] = i + 1;  
        pthread\_create(&readers[i], NULL, reader, &reader\_ids[i]);  
    }  
     
    for (int i = 0; i < NUM\_WRITERS; i++) {  
        writer\_ids[i] = i + 1;  
        pthread\_create(&writers[i], NULL, writer, &writer\_ids[i]);  
    }  
     
    for (int i = 0; i < NUM\_READERS; i++)  
        pthread\_join(readers[i], NULL);  
    for (int i = 0; i < NUM\_WRITERS; i++)  
        pthread\_join(writers[i], NULL);  
     
    sem\_destroy(&mutex);  
    sem\_destroy(&writeblock);  
     
    return 0;  
}

**Output:**

****