LAB 3: Concurrent Programming & Synchronization Problems

OBJECTIVE:

The primary objective of this report is to comprehensively explore and analyze three classic synchronization problems in concurrent programming: the Producer-Consumer Problem, Sleeping Barber Problem, and Dining Philosophers Problem, through a detailed examination of each scenario.

THEORY:

1. **Producer-Consumer Problem:**
   * **Scenario:** In a multi-process or multi-threaded environment, there are two types of processes - producers and consumers. Producers produce items and place them in a shared buffer, while consumers consume items from the buffer.
   * **Challenge:** Synchronization is required to ensure that the buffer is not accessed concurrently by a producer and a consumer. The buffer must be appropriately managed to avoid issues like overflow or underflow.
2. **Sleeping Barber Problem:**
   * **Scenario:** In a barbershop, there is one barber and a certain number of waiting chairs for customers. If a customer arrives and the barber is busy, the customer takes a seat in one of the waiting chairs. If all chairs are occupied, the customer leaves.
   * **Challenge:** Synchronization is needed to prevent race conditions where multiple customers try to access the barber at the same time, or the barber tries to serve multiple customers simultaneously.
3. **Dining Philosophers Problem:**
   * **Scenario:** Five philosophers are sitting at a dining table, and each philosopher thinks and eats. Between each pair of philosophers, there is a single chopstick. To eat, a philosopher needs both the chopsticks on either side of them.
   * **Challenge:** The problem is to design a synchronization solution that prevents deadlock (where no philosopher can finish eating) and avoids contention for resources (chopsticks) between philosophers.

Program 1: Producer Consumer Problem

Solution:

#include<stdio.h>

int main(){

int bufsize, in, out, produce = 0, consume = 0 , counter = 0;

in = 0; out = 0;

printf("Please enter the size of buffer: ");

scanf("%d", &bufsize);

int buffer[bufsize] ;

char choice ;

printf ("Enter p to produce an item in buffer:\n") ;

printf ("Enter c to consume an item from buffer:\n") ;

printf ("Enter q to Quit:\n") ;

do{

scanf("%c", &choice);

switch(choice){

case 'p':

if(counter == bufsize)

printf("Buffer is Full\n");

else { printf("Enter the value: ");

scanf("%d", &produce);

buffer[in] = produce;

in = (in+1)%bufsize;

counter++ ;

}break ;

case 'c':

if(counter == 0)

printf("Buffer is Empty\n");

else {

consume = buffer[out];

printf("The consumed value is %d\n", consume);

out = (out+1)%bufsize;

counter-- ;

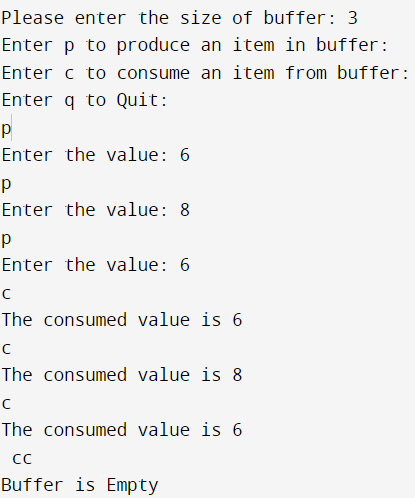
}break ;

}

}while( choice != 'q') ;

}

Output:



Program 2: Dining Philosopher problem

Solution:

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#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++) {

// create philosopher processes

pthread\_create(&thread\_id[i], NULL, philospher, &phil[i]);

printf("Philosopher %d is thinking\n", i + 1);}

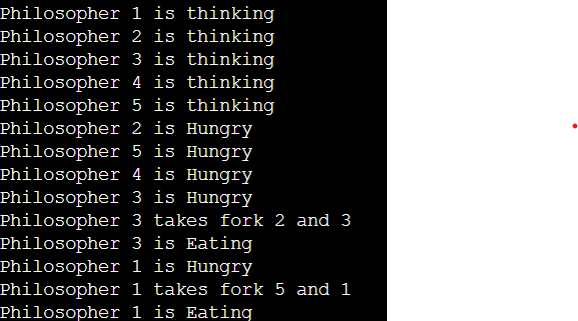
for (i = 0; i < N; i++)

pthread\_join(thread\_id[i], NULL);

}

Output:

Infinite loop in which philosopher switch between thinking, hungry and eating



Program 3: Sleeping Barber’s Problem

Solution:

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <time.h>

#include <pthread.h>

#include <semaphore.h>

#define MAX\_CUSTOMERS 25

void \*customer(void \*num);

void \*barber(void \*);

void randwait(int secs);

sem\_t waitingRoom;

sem\_t barberChair;

sem\_t barberPillow;

sem\_t seatBelt;

int allDone = 0;

int main(int argc, char \*argv[]) {

pthread\_t btid;

pthread\_t tid[MAX\_CUSTOMERS];

long RandSeed;

int i, numCustomers, numChairs;

int Number[MAX\_CUSTOMERS];

printf("Enter the number of Customers : ");

scanf("%d",&numCustomers) ;

printf("Enter the number of Chairs : ");

scanf("%d",&numChairs);

if (numCustomers > MAX\_CUSTOMERS) {

printf("The maximum number of Customers is %d.\n",

MAX\_CUSTOMERS);

exit(-1); }

for (i=0; i<MAX\_CUSTOMERS; i++) {

Number[i] = i; }

sem\_init(&waitingRoom, 0, numChairs);

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

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

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

pthread\_create(&btid, NULL, barber, NULL);

for (i=0; i<numCustomers; i++) {

pthread\_create(&tid[i], NULL, customer, (void \*)&Number[i]);

sleep(1); }

for (i=0; i<numCustomers; i++) {

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

sleep(1); }

allDone = 1;

sem\_post(&barberPillow);

pthread\_join(btid,NULL); }

void \*customer(void \*number) {

int num = \*(int \*)number;

printf("Customer %d leaving for barber shop.\n", num);

randwait(2);

printf("Customer %d arrived at barber shop.\n", num);

sem\_wait(&waitingRoom);

printf("Customer %d entering waiting room.\n", num);

sem\_wait(&barberChair);

sem\_post(&waitingRoom);

printf("Customer %d waking the barber.\n", num);

sem\_post(&barberPillow);

sem\_wait(&seatBelt);

sem\_post(&barberChair);

printf("Customer %d leaving barber shop.\n", num); }

void \*barber(void \*junk) {

while (!allDone) {

printf("The barber is sleeping\n");

sem\_wait(&barberPillow);

if (!allDone) {

printf("The barber is cutting hair\n");

randwait(2);

printf("The barber has finished cutting hair.\n");

sem\_post(&seatBelt);

}else {

printf("The barber is going home for the day.\n"); } }}

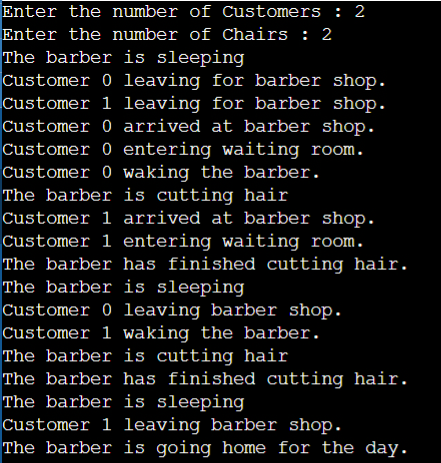
void randwait(int secs) {

int len;

len = (int) ((1 \* secs) + 1);

sleep(len); }

Output:



CONCLUSION:

In conclusion, the exploration of the Producer-Consumer Problem, Sleeping Barber Problem, and Dining Philosophers Problem has provided valuable insights into the intricacies of concurrent programming and shared resource management. These classic synchronization problems exemplify the challenges inherent in designing systems where multiple processes or threads interact.