Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

# Operating Systems Project Report

# Applications of Threads and Semaphores


**Prepared By:**

| | |
|---|---|
| R. Akshatha | 19BCE0807 |
| Reva Munish | 19BCE0810 |
| Vishnu Nair | 19BCE2467 |


**Submitted To:**

Prof. Jothi K. R.

**Slot:**

L5 + L6

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

## School of Computer Science and Engineering

## DECLARATION

I hereby declare that the project entitled **“Applications of Threads and Semaphores”** submitted by me to the School of Computer Science and Engineering, Vellore Institute of Technology, Vellore-14 towards the partial fulfilment of the requirements for the course CSE2005 – Operating Systems is a record of bonafide work carried out by me under the supervision of **Prof. Jothi K R.** I further declare that the work reported in this project has not been submitted and will not be submitted, either in part or in full, for any other course or purpose of this institute or of any other institute or university.

Signature

**Name: Vishnu Nair**

**Reg. No: 19BCE2467**

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

## School of Computer Science and Engineering

## CERTIFICATE

The project report entitled "**Applications of Threads and Semaphores**" is prepared and submitted by **Vishnu Nair (Register No: 19BCE2467),** has been found satisfactory in terms of scope, quality and presentation as partial fulfilment of the course CSE2005 – Operating Systems in Vellore Institute of Technology, Vellore-14, India.

**Guide: Jothi K R**

**(Signature)**

## ACKNOWLEDGEMENT

The project "**Applications of Threads and Semaphores"** was made possible because of inestimable inputs from everyone involved, directly or indirectly. I would first like to thank my guide, **Prof. Jothi K. R.,** who was highly instrumental in providing not only a required and innovative base for the project but also crucial and constructive inputs that helped make my final product.

I would also like to acknowledge the role of the HOD, **Dr. Santhi V.,** who was instrumental in keeping me updated with all necessary formalities and posting all the required formats and document templates through the mail, which I was glad to have had.

It would be no exaggeration to say that the Dean of SCOPE, **Dr. Saravanan R.**, was always available to clarify any queries and clear the doubts I had during the course of my project.

Finally, I would like to thank **Vellore Institute of Technology**, for providing me with a flexible choice and execution of the project and for supporting my research and execution related to the project.

# TABLE OF CONTENT

# 1. ABSTRACT

Through this project, we try to understand the various applications of threads and semaphores. To study how these are used and to be able to use these methods ourselves, we have studied some already existing problems related to threads and semaphores. Then we have come up with 3 new problem statements and solved them using the concepts mentioned above.

The problems which we chose to study are :

- Reader/Writer Problem
- Dining Philosopher Problem
- Producer-Consumer Problem
- Doctor-Patient Problem
- Cigarette-Smokers Problem
- Sleeping Barber Problem

We have used C language to solve the problem statements.

## 2. INTRODUCTION

The concept of multiple threads can be used for a variety of reasons: to build responsive servers that interact with multiple clients, to run computations in parallel on a multiprocessor for performance, and as a structuring mechanism for implementing rich user interfaces. In general, threads are useful whenever the software needs to manage a set of tasks with varying interaction latencies, exploit multiple physical resources, or execute largely independent tasks in response to multiple external events.

The only way to significantly improve performance is to enhance the processor's computational capabilities. In general, this means increasing parallelism — in all its available forms. At present only certain forms of parallelism are being exploited.

### A Brief Study of Existing Problems:

- Reader/Writer Problem:

  When one writer is writing into the data area, another writer or reader can't access the data area at the same time. When one reader is reading from the data area, other readers can also read but the writer can't write.

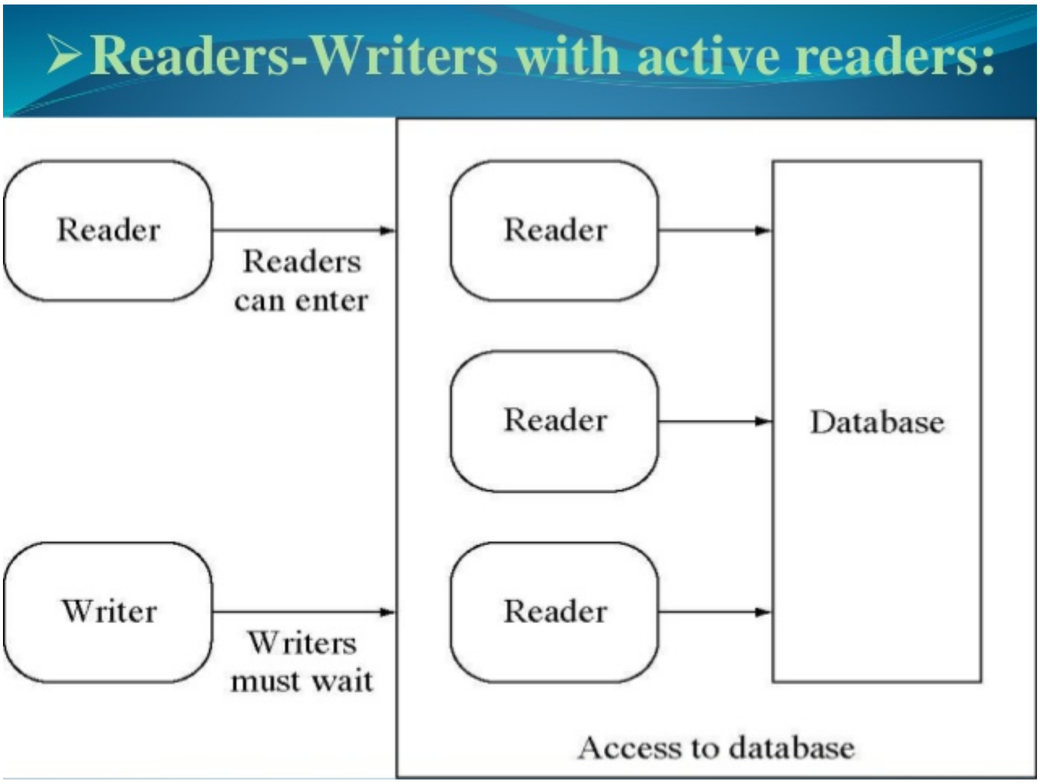


Figure 2.1

- Dining Philosopher Problem:

  There are 5 philosophers, who share a round table, which has one chair for each person. There are 5 chopsticks placed in between them and a bowl of rice in the middle of the table. When a philosopher gets hungry, he grabs the nearest two chopsticks and starts eating, if the person beside him is already eating, then he has to wait until he gets the chopstick from that person.

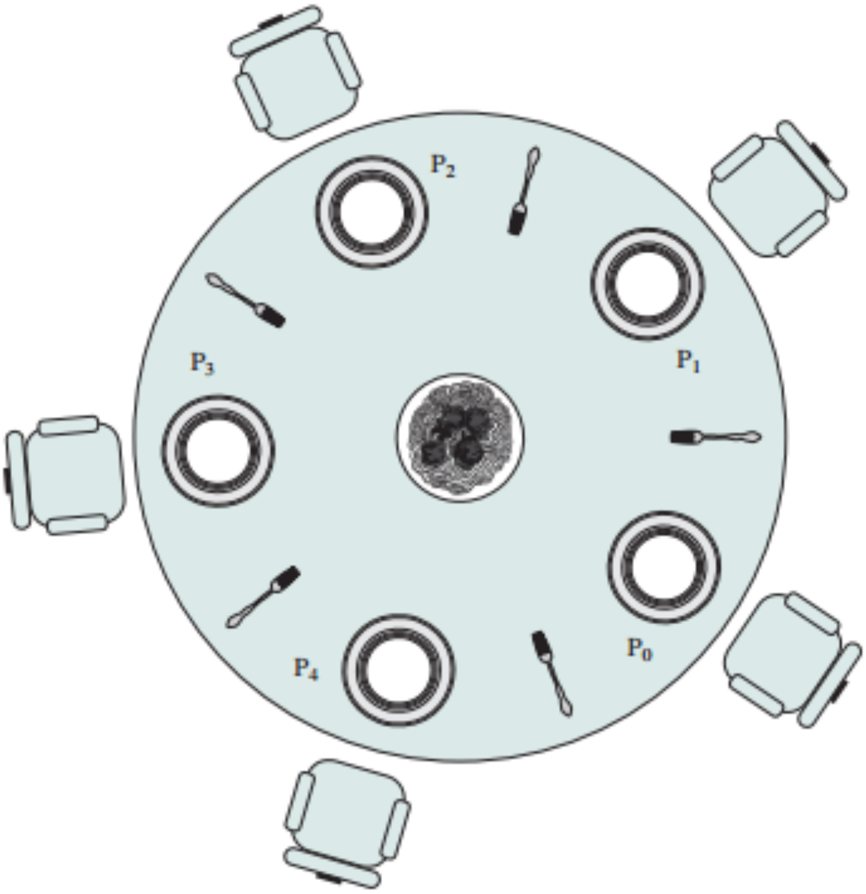


Figure 2.2

- Producer-Consumer Problem:

A producer produces the item which the consumer consumes. When the producer has not yet produced any item, the consumer cannot consume it and when the consumer does not consume at least one item in the buffer, the producer has to wait until he gets one empty place in the buffer in order to produce an item.

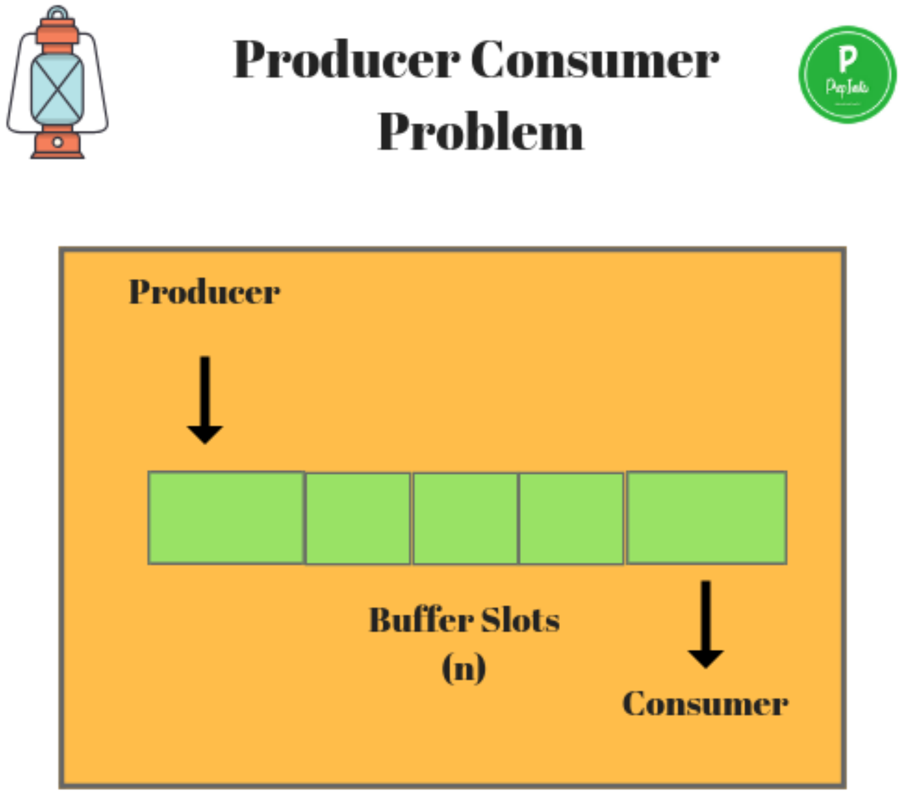


Figure 2.3

- Doctor-Patient Problem:

  In clinics, patients are given chairs where they can wait until they meet the doctor. If the number of patients is more, then some patients have to stand and wait until a patient comes out of the doctor's room and a patient enters the doctor's room. Later, when a chair is left empty, the patient who is waiting can take the seat.

- Cigarette-Smokers Problem:

  There are three requirements to make the cigarette, and each smoker will carry any one ingredient of his choice. Later, two random requirements out of three are kept on the table and the lucky person who has the other ingredient can make the cigarette and smoke. Within that time, another two ingredients can be kept on the table and the process continues on.

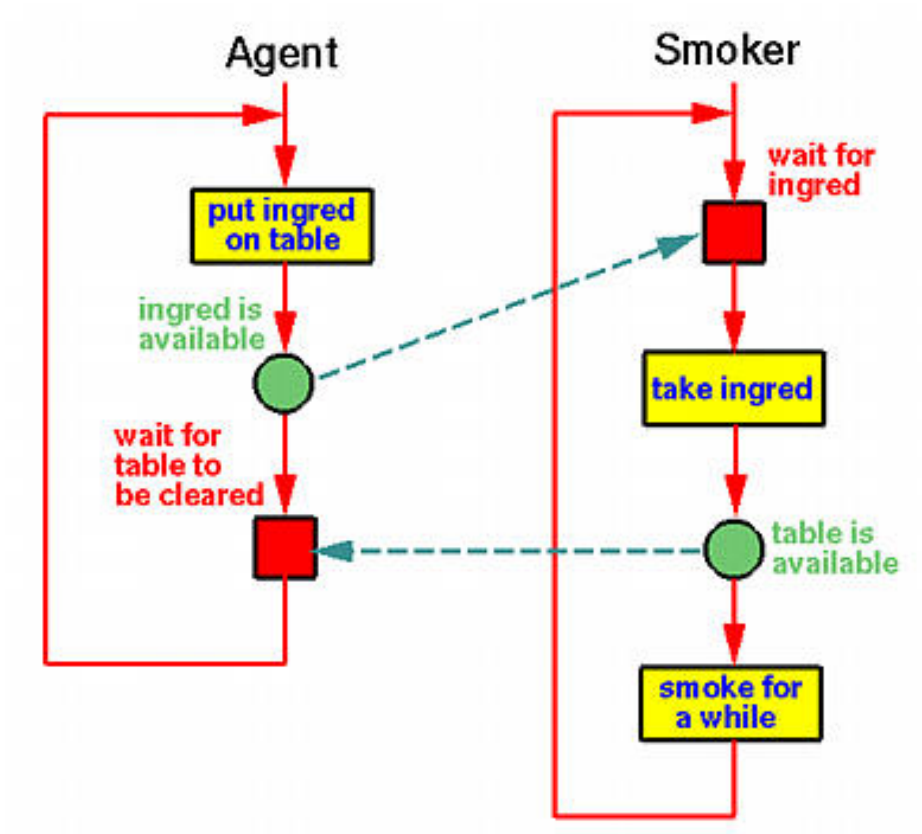


Figure 2.4

- Sleeping Barber Problem:

  There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair. If there is no customer, then the barber sleeps in his own chair. When a customer arrives, he has to wake up the barber. If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty

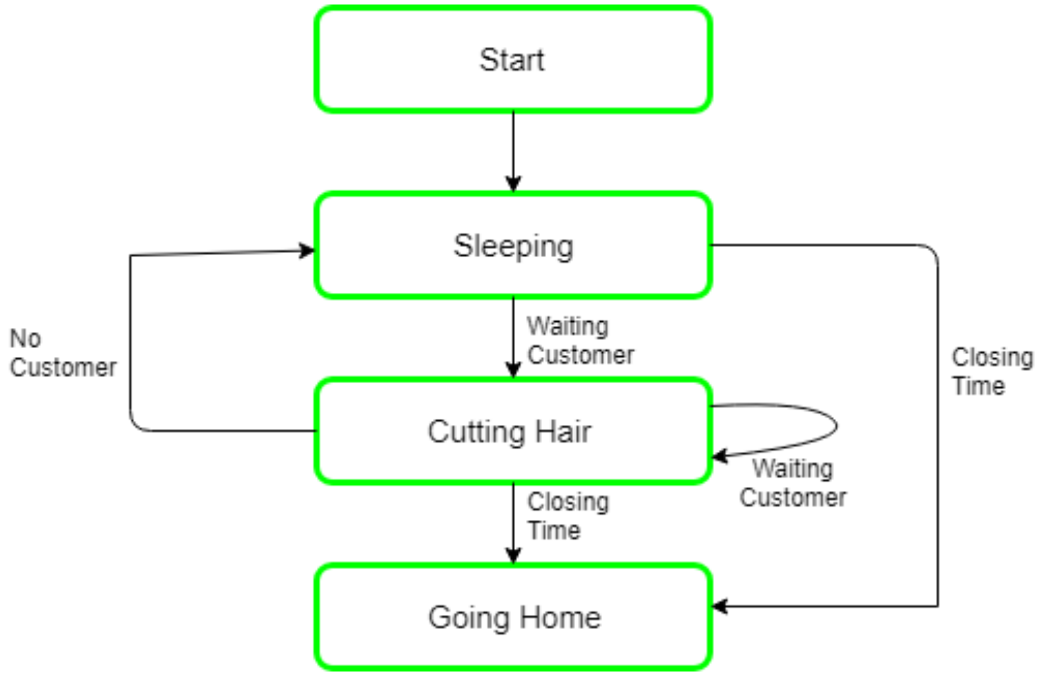


Figure 2.5

## Problem Statements:

- Parts of a Car:

  Let's assume a situation where the headquarters of the company can accommodate only 2 of the 3 resources for manufacturing a car at a time. We assume that the three resources are Engine, Chassis and Music System. The 2 resources are supplied to the headquarters and sent to a workshop where the third resource is available. After all the resources are received at the workshop, the car is assembled and is ready to be sold in the market.

  Constraints:

  - The resources are allocated to the headquarters randomly.
  - Only 2 resources are accessed at a time in the headquarters and the third resource is made available in the workshop.

- Gas Station:

Consider a gas station with three gas pumps, three attenders and a waiting lane that can accommodate four cars. Only 7 cars are allowed inside the station; 3 at the gas pump and 4 in the waiting line. A car will not enter the gas station if there are already 7 cars in the station. Once inside, the car enters the waiting lane (a queue). When an attender is free, the car at the head of the queue drives up to that gas pump and gets served. When a car's filling is done, any attender can accept the payment, but because there is only one ATM machine, payment is accepted for one car customer at a time. The attenders divide their time among serving cars, accepting payment whilst waiting to do the same.

Constraints:

- The Cars invoke the following functions in order: enterStation, waitInLine, goToPump, pay, exitStation.
- Attenders invoke serveCar and acceptPayment.
- Cars cannot invoke enterStation if the gas station is at capacity.
- If all the three pumps are busy, a car in the wait lane cannot invoke goToPump until one of the cars being served invokes pay.
- The car customer has to pay before the attender can acceptPayment.
- The attender must acceptPayment before the car can exitStation.

- Cafeteria Problem:

Assume a cafeteria with only one cook during lunch times that puts the food in the trays and puts the trays on a conveyor belt so students can fetch them and eat. The conveyor can take at most 8 trays. If the conveyor is full, the cook sleeps until a tray is fetched. Students come in and fetch their trays. If there are no trays available, they wait until the cook fills a tray and puts it on the conveyor.

Constraints:

- If the conveyor is full, cook sleeps; if not, he fills trays.
- Each student has to be a different thread.
- One student can fetch only one tray at a time.
- Order of the students while fetching a tray is not important.

# 3.LITERATURE SURVEY

## 3.1. Concurrent programming

Concurrent programming is essential in order to reduce the execution time in several application domains, such as image processing and simulations. A concurrent program is a group of processes or threads which execute simultaneously and work together to perform a task. These threads access a common addressing space and interact through memory using shared variables. The most common method to develop multithreaded programs is to use thread libraries, like PThreads (POSIX Threads). [1]

## 3.2. Synchronization

Synchronization is the mechanism used to guarantee mutual exclusion among processes when accessing a critical section and to achieve inter-process communication. Processes involved in synchronization become indirectly aware of each other by waiting on a condition that is set by other processes. [2]

In concurrent programming a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in a fixed amount of time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Therefore, a synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore. [3]

## 3.3. Communication Between Processes

Processes communicate with each other using inter-process communication (IPC) mechanisms. Pipes, files and shared memory are some examples of the methods which are used for IPC. Files are the most obvious means of passing information and communicating between processes. One process writes to a file and the other process reads from that file. Despite the fact that files are not interactive, they are commonly used for IPC. Another method of connecting the output data stream of one process to the input of another process is referred to as a pipe. A pipe may be of two types: unidirectional and bidirectional. In unidirectional pipes, the second process cannot talk back to the first process. In bi-directional pipes, two unidirectional pipes connect the two processes, so that both of the processes can communicate with each other. Pipes can hold only a finite amount of data. Deadlock can arise when using pipes for IPC. For example, while using a bi-directional pipe between two processes, if both unidirectional pipes get filled up, then if both processes are blocked writing to their pipes, neither can read any information from their own unidirectional pipe because they haven't finished writing into the other pipe. [2]

## 3.4. Synchronization Mechanisms

Operating system based solutions can be achieved by adding process synchronization support to an operating system. The use of semaphores is one example of this type of support. Semaphores can be utilized to solve most of the synchronization problems. Dijkstra originally defined the semaphore concept. A semaphore is a non-negative integer variable that has an implicit queue associated with it. The value of the variable can be handled only by the following two primitive operations: Wait and Signal.[2]

The Wait or P operation is used by a process which would like to enter a critical section. If the value of the semaphore variable is greater than zero, it is decremented by one and the process is executed. If the value is less than or equal to zero, then the process is added to  the queue that is associated with the semaphore. [2]

The Signal or V operation is used by a process which is leaving a critical section. It checks the queue to see if there is a process waiting. The processes in the queue are in a passive waiting state. If there is a process, it is activated. If no process is waiting, the semaphore is incremented by one. There are many extensions to the basic definition and implementation of the concept of a semaphore, intended to suit various synchronization requirements, runtime environments, and implementation platforms. [2]

Semaphores are also particularly designed to support an effective waiting mechanism. If a thread cannot continue until some change occurs, it is undesirable for that thread to be looping and repeatedly checking the state until it changes. In this case semaphore can be utilized to represent the right of a thread to proceed. A non-zero value implies that the thread should proceed, whereas zero implies to hold off. When a thread attempts to decrement an unavailable semaphore (with a zero value), it efficiently waits until another thread increments the semaphore to signal the state change that will permit it to proceed.[4]

# 4. IMPLEMENTATION

## 1. Parts of a Car

```
#include<stdio.h>
#include<sys/types.h>
#include<semaphore.h>
#include<pthread.h>
#include <unistd.h>

int clear_cycle = 1;
int part[2], gen = 0;
char *parts[] = { "Engine", "Chassis", "Music System" };

sem_t ready;
int place[4] = { 0, 1, 2, 3 };

void * branch (void *arg)
{
  int i, j, k = 0;
  while (1)
    {
      sleep (1);
      sem_wait (&ready);
      if (clear_cycle == 1)
  {
    i = k;
    j = i + 1;
    if (j == 3)
      j = 0;
    k = j;
    part[0] = i;
    part[1] = j;
    printf ("Headquarters can take %s and %s\n", parts[i], parts[j]);
    gen = 1;
    clear_cycle = 0;
  }
      sem_post (&ready);
      printf ("Parts are being moved\n");
    }
}
```

Figure 4.1.1

```c
void * assembly (void *arg)
{
    int p_id = (*(int *) arg);
    int flag = 1;
    while (1)
      {
        sleep (1);
        sem_wait (&ready);
        if (clear_cycle == 0)
    {
      if (gen && part[0] != p_id && part[1] != p_id)
        {
          printf ("Branch %d has completed the assembly\n", p_id);
          clear_cycle = 1;
          gen = 0;
          flag = 0;
        }
    }
        sem_post (&ready);
        if (clear_cycle == 1 && flag == 0)
    printf ("The cycle is complete\n");
      }
}

int main ()
{
    pthread_t branch1, branch2, branch3, headquarters;
    sem_init (&ready, 0, 1);
    pthread_create (&headquarters, NULL, branch, &place[0]);
    pthread_create (&branch1, NULL, assembly, &place[1]);
    pthread_create (&branch2, NULL, assembly, &place[2]);
    pthread_create (&branch3, NULL, assembly, &place[3]);
    while (1);
    return 0;
}
```

Figure 4.1.2

## 2. Gas Station Problem

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

#define carsno 1000
#define atten 3
#define maxcars 7
#define pumpsno 3
#define len 4

pthread_t ts[1008];
pthread_t cash_counter;
sem_t maxcap;
int freepump=pumpsno;
sem_t mut2;
int freepos=len;
sem_t mutqueue;
sem_t semq;
sem_t sempump;
int pump[pumpsno];
sem_t readycar;
sem_t fin[carsno];
sem_t done[carsno];
sem_t fincounter;
sem_t leavepump[pumpsno];
sem_t mut3;
sem_t mut4;
sem_t readypayment;
sem_t receipt[carsno];
int pipe1[2],pipe2[2];

void writepipe(int writefd,int val)
{
    if(write(writefd,&val,sizeof(int)) != sizeof(int))
    {
        printf("Pipe write error");
        exit(0);
    }
}
void readpipe(int readfd,int *val,int *stat)
{
    int n;
    if((n=read(readfd,val,sizeof(int))) == -1)
    {
        printf ("Pipe read error");
        exit(0);
    }
    *stat=(char)n;
}
```

Figure 4.2.1

```c
void Attender(int number);
void PAY();
void Car(int Car_Id);
void serveCar(int myCar,int number,int myPump);
void acceptPayment();
void CarMaker();
void AttenderMaker();

int main()
{
    int is=time(NULL);
    srand(is);
    int i;
    sem_init(&maxcap,0,maxcars);
    sem_init(&mut2,0,1);
    sem_init(&mut3,0,1);
    sem_init(&mut4,0,1);
    sem_init(&mutqueue,0,1);
    sem_init(&semq,0,len);
    sem_init(&sempump,0,pumpsno);
    sem_init(&readycar,0,0);
    sem_init(&fincounter,0,0);
    sem_init(&readypayment,0,0);

    for(i=0;i<carsno;i++)
    {
        sem_init(&fin[i],0,0);
        sem_init(&receipt[i],0,0);
    }
    for(i=0;i<pumpsno;i++)
    {
        sem_init(&leavepump[i],0,0);
        pump[i]=-1;
    }

    if (pipe(pipe1) < 0)
        printf("Can't create pipe1\n");
    if (pipe(pipe2) < 0)
        printf("Can't create pipe2\n");
    AttenderMaker();
    pthread_create(&cash_counter,NULL,(void *)&PAY,NULL);
    CarMaker();
}
```

Figure 4.2.2

```c
void PAY()
{
    int Car;
    int myid=-1;
    int Mychairno = 0;
    int stat;
    while(1)
    {
        sem_wait(&readypayment);
        sem_wait(&mut4);
        readpipe(pipe2[0],&Car,&stat);
        sem_post(&mut4);
        printf("\nCASH COUNTER : Car %d has arrived with payment. Calling an attender\n",Car);
        sem_wait(&mut3);
        writepipe(pipe1[1],myid);
        writepipe(pipe1[1],Mychairno);
        sem_post(&mut3);
        sem_post(&readycar);
        printf("\nCASH COUNTER : Waiting payment confirmation from Car %d\n",Car);
        sem_wait(&fincounter);
        printf("\nPayment started\n");
        sleep(1);
        printf("\nCASH COUNTER : Car %d has paid\n",Car);
        sem_post(&receipt[Car]);
    }
}

void AttenderMaker()
{
    int i=1;
    while (i<=atten) {
        pthread_create(&ts[i],NULL,(void *)&Attender,(void *)i);
        i++;
    }
}

void CarMaker()
{
    int i=0;
    while (i <carsno)
    {
        sleep(rand()%3);
        pthread_create(&ts[i+atten],NULL,(void *)&Car,(void *)i);
        printf("\nCar %d has arrived\n",i );
        i++;
    }
}
```

Figure 4.2.3

```
void Attender(int num)
{
    int mycar, mypump, stat;
    while(1)
    {
        printf("\nAttender %d waiting for a car\n",num);
        sem_wait(&readycar);
        sem_wait(&mut3);
        readpipe(pipe1[0],&mycar,&stat);
        readpipe(pipe1[0],&mypump,&stat);
        sem_post(&mut3);
        if (mycar!=-1)
            serveCar(mycar,num,mypump);
        else
            acceptPayment();
    }
}

void acceptPayment()
{
    sem_post(&fincounter);
}

void serveCar(int mycar,int num,int mypump)
{
    printf("\nAttender %d fills gas in Car %d using Pump %d\n",num,mycar,mypump);
    sleep(3);
    printf("\nFilling completed for Car %d by Attender %d using Pump %d\n",mycar,num,mypump);
    sem_post(&fin[mycar]);
    sem_wait(&leavepump[mypump]);
    printf("\nAttender %d instructs Car %d to leave Pump %d and go to CASH COUNTER\n",num,mycar,mypump);
    sem_post(&sempump);
}
```

Figure 4.2.4

```
void Car(int carid)
{
    int i,k;
    sem_wait(&maxcap);
    printf("\nCar %d enters the gas station\n",carid);
    sem_wait(&mut2);
    sem_wait(&mutqueue);
    if ((freepump==0) || (freepos<len))
    {
        sem_post(&mutqueue);
        sem_post(&mut2);
        sem_wait(&mutqueue);
        if (freepos<=0)
            printf("\nCar %d is waiting\n",carid);
        freepos--;
        sem_post(&mutqueue);
        sem_wait(&semq);
        printf("\nCar %d is in the waiting lane now\n",carid);
        sem_wait(&sempump);
        sem_wait(&mutqueue);
        freepos++;
        sem_post(&mutqueue);
        sem_post(&semq);
        printf("\nCar %d has released its queue position\n",carid);
    }
    else
    {
        sem_post(&mutqueue);
        sem_post(&mut2);
        sem_wait(&sempump);
    }
    sem_wait(&mut2);
    i = 0;
    while ((pump[i]!=-1) && (i<pumpsno))
    i++;
    if (i == pumpsno)
        exit(0);
    else
    {
        pump[i] = carid;
        freepump--;
        printf("\nCar %d occupies Pump %d. \nNumber of free pumps is %d\n",carid,i,freepump);
    }
```

Figure 4.2.5

```c
    sem_post(&mut2);
    sem_wait(&mut3);
    printf("\nFilling for Car %d started\n",carid);
    writepipe(pipe1[1],carid);
    writepipe(pipe1[1],i);
    sem_post(&mut3);
    sem_post(&readycar);
    sem_wait(&fin[carid]);
    sem_wait(&mut2);
    freepump++;
    pump[i]=-1;
    sem_post(&leavepump[i]);
    sem_post(&mut2);
    printf("\nCar %d left Pump %d to go to the CASH COUNTER. \nNumber of free pumps is %d\n",carid,i,freepump);
    sem_wait(&mut4);
    writepipe(pipe2[1],carid);
    sem_post(&mut4);
    printf("\nCar %d ready to pay\n",carid);
    sem_post(&readypayment);
    sem_wait(&receipt[carid]);
    printf("\nCar %d has paid\n",carid);
    sem_post(&done[carid]);
    sem_post(&maxcap);
    if (carid==(carsno-1))
    {
        for(k=0;k<=(carsno-1);k++)
        sem_wait(&done[k]);
        puts("\nAll cars served");
        exit(0);
    }
    else  pthread_exit(0);
}
```

Figure 4.2.6

## 3. Cafeteria Problem

```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

sem_t tray_full_sem;

pthread_mutex_t print_lock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t tray_lock = PTHREAD_MUTEX_INITIALIZER; //access the tray number
pthread_mutex_t stud_wait_lock = PTHREAD_MUTEX_INITIALIZER; //when new student arrives
pthread_mutex_t stud_tot_lock = PTHREAD_MUTEX_INITIALIZER;//to access total no of students
pthread_mutex_t cook_sleep_lock = PTHREAD_MUTEX_INITIALIZER; //for cook's sleep state

int trays = 8; //current trays

int cook_sleep = 0;
char *cook_str[] = { "Working", "Sleeping" };

int tray_tot = 8; //total no of trays
int stud_tot = 0;
int stud_fetch = 0;

static time_t START_TIME;

void *cook (void *);
void *student (void *);
void screen (void);

int main ()
{
  pthread_t thread_cook;
  pthread_t thread_student;

  if (sem_init (&tray_full_sem, 0, 0) == -1)
    {
      perror ("sem_init");
      exit (EXIT_FAILURE);
    }

  START_TIME = time (NULL);
  printf ("\n\t\t WELCOME TO THE CAFETERIA %s\n", ctime (&START_TIME));
  pthread_create (&thread_cook, NULL, &cook, NULL); //cook created

  while (1)
    {
      sleep (rand () % 3 + 1); //time for student to arrive; 1-3 secs
      pthread_create (&thread_student, NULL, &student, NULL);
    }
  void *status;
  pthread_join (thread_cook, status);
  return 0;
}
```

Figure 4.3.1

```c
void * cook (void *arg)
{
  pthread_mutex_lock (&print_lock);
  pthread_mutex_unlock (&print_lock);
  unsigned int random_time;
  time_t now;

  while (1)
    {
      pthread_mutex_lock (&tray_lock);
      if (trays == 8)
      //if there are 8 trays, the cook will sleep
    {
      pthread_mutex_lock (&cook_sleep_lock);
      cook_sleep = 1;
      pthread_mutex_unlock (&cook_sleep_lock);
      pthread_mutex_lock (&print_lock);
      printf ("[ %ld ] - Cook goes to sleep \n",time (NULL) - START_TIME); //tells the time (sec) when cook goes to sleep
      screen();
      pthread_mutex_unlock (&print_lock);
      pthread_mutex_unlock (&tray_lock); // releases tray lock before going to sleep
      sem_wait (&tray_full_sem);
      pthread_mutex_lock (&cook_sleep_lock);
      cook_sleep = 0;
      pthread_mutex_unlock (&cook_sleep_lock);
      pthread_mutex_lock (&print_lock);
      printf ("[ %ld ] - Cook is awake \n", time (NULL) - START_TIME); //tells the time (sec) when cook is awake
      screen ();
      pthread_mutex_unlock (&print_lock);
    }
      else
        pthread_mutex_unlock (&tray_lock);

      random_time = rand () % 5 + 2; //time taken to fill a tray
      now = time (0);

      pthread_mutex_lock (&print_lock);
      printf ("[ %ld ] Cook starts filling tray no. %d\n", time (NULL) - START_TIME, tray_tot + 1);
      screen ();
      pthread_mutex_unlock (&print_lock);

      while (time (0) - now < random_time); //wait till the tray is being filled.

      pthread_mutex_lock (&tray_lock);
      trays++;
      pthread_mutex_unlock (&tray_lock);
      tray_tot++;
      pthread_mutex_lock (&print_lock);
      printf ("[ %ld ] Cook finishes filling tray no. %d\n", time (NULL) - START_TIME, tray_tot);
      screen ();
      pthread_mutex_unlock (&print_lock);
    }
}
```

Figure 4.3.2

```c
void * student (void *arg)
{
  pthread_mutex_lock (&stud_tot_lock);
  stud_tot++;
  pthread_mutex_unlock (&stud_tot_lock);
  pthread_mutex_lock (&print_lock);
  printf ("[ %ld ] Student no. %d arrives \n", time (NULL) - START_TIME, stud_tot);
  screen ();
  pthread_mutex_unlock (&print_lock);
  pthread_mutex_lock (&stud_wait_lock);
  while (trays == 0); //there are no trays so the student waits
  sleep (1);
  pthread_mutex_lock (&tray_lock);
  trays--;

  if (trays == 7) //waking up the cook
    {
      pthread_mutex_lock (&cook_sleep_lock);
      if (cook_sleep == 1)
      sem_post (&tray_full_sem);
      pthread_mutex_unlock (&cook_sleep_lock);
    }

  stud_fetch++;
  pthread_mutex_lock (&print_lock);
  printf ("[ %ld ] Student no. %d fetches his tray \n", time (NULL) - START_TIME, stud_fetch);
  screen ();
  pthread_mutex_unlock (&print_lock);
  pthread_mutex_unlock (&tray_lock);
  pthread_mutex_unlock (&stud_wait_lock);
  pthread_exit (NULL);
}
```

Figure 4.3.3

```c
void screen (void)
{
  printf ("\n|-------------------------------------------------------|\n");
  printf ("|\t\t\t\t\t\tCONVEYOR \t\t\t\t\t\t|\n");
  printf ("|\t\t\t\t----------------------- \t\t\t\t\t|\n");
  printf ("|\t\tTRAYS READY\t:%d \t\t\t\t\t\t\t\t\t|\n", trays);
  printf ("|\t\tTRAYS AVAILABLE\t:%d \t\t\t\t\t\t\t\t|\n", (8 - trays));

  printf ("|\tCOOK\t\t\t\t\tWAITING LINE \t\t\t\t|\n");
  printf ("|  ---------\t\t       ---------------- \t\t\t|\n");
  printf ("|  %s\t\t\t\t      %d \t\t\t\t\t\t|\n", cook_str[cook_sleep],
      (stud_tot - stud_fetch));
  printf ("|\t\t\t\t\t\t    Students waiting \t\t\t|\n");
  if (cook_sleep == 0)
    printf ("|  FILLING TRAY NO. %d \t\t\t\t\t\t\t\t\t|\n", (tray_tot + 1));
  printf ("|\t\t\t\t\t\t\t\t\t\t\t\t\t\t|\n");

  printf ("|\t\t\t\t\t\t\t\t\t\t\t\t\t\t|\n");
  printf ("|\t\t    CAFETERIA STATISTICS \t\t\t\t\t\t|\n");
  printf ("|\t\t---------------------------- \t\t\t\t\t|\n");
  printf ("|\t\tNO. OF TRAYS FILLED\t\t:%d\t\t\t\t\t\t|\n", tray_tot);
  printf ("|\t\tNO. OF STUDENTS ARRIVED\t:%d\t\t\t\t\t\t|\n", stud_tot);
  printf ("|\t\tNO. OF TRAYS FETCHED\t:%d\t\t\t\t\t\t|\n", stud_fetch);
  printf ("|-------------------------------------------------------|\n\n");
}
```

Figure 4.3.4

## 5. RESULT AND DISCUSSION

### 1. Parts of a Car

```
Headquarters can take Engine and Chassis
Parts are being moved
Branch 3 has completed the assembly
The cycle is complete
Headquarters can take Chassis and Music System
Parts are being moved
Branch 3 has completed the assembly
The cycle is complete
Headquarters can take Music System and Engine
Parts are being moved
Branch 3 has completed the assembly
The cycle is complete
Headquarters can take Engine and Chassis
Parts are being moved
Branch 2 has completed the assembly
The cycle is complete
The cycle is complete
Headquarters can take Chassis and Music System
Parts are being moved
Branch 3 has completed the assembly
The cycle is complete
The cycle is complete
Headquarters can take Music System and Engine
Parts are being moved
Branch 3 has completed the assembly
The cycle is complete
```

Figure 5.1.1

```
Headquarters can take Music System and Engine
Parts are being moved
Branch 3 has completed the assembly
The cycle is complete
The cycle is complete
Headquarters can take Engine and Chassis
Parts are being moved
Branch 3 has completed the assembly
The cycle is complete
The cycle is complete
Headquarters can take Chassis and Music System
Parts are being moved
Branch 3 has completed the assembly
The cycle is complete
```

Figure 5.1.2

## 2. Gas Station Problem

```
Attender 2 waiting for a car

Attender 1 waiting for a car

Attender 3 waiting for a car

Car 0 has arrived

Car 0 enters the gas station

Car 0 occupies Pump 0.
Number of free pumps is 2

Filling for Car 0 started

Car 1 enters the gas station

Car 1 has arrived

Attender 2 fills gas in Car 0 using Pump 0

Car 1 occupies Pump 1.
Number of free pumps is 1

Filling for Car 1 started

Attender 1 fills gas in Car 1 using Pump 1

Car 2 has arrived
```

Figure 5.2.1

```
Car 2 enters the gas station

Car 2 occupies Pump 2.
Number of free pumps is 0

Filling for Car 2 started

Attender 3 fills gas in Car 2 using Pump 2

Car 3 has arrived

Car 3 enters the gas station

Car 3 is in the waiting lane now

Filling completed for Car 0 by Attender 2 using Pump 0

Attender 2 instructs Car 0 to leave Pump 0 and go to CASH COUNTER

Attender 2 waiting for a car

Car 0 left Pump 0 to go to the CASH COUNTER.
Number of free pumps is 1

Car 0 ready to pay

Car 3 has released its queue position

CASH COUNTER : Car 0 has arrived with payment. Calling an attender
```

Figure 5.2.2

```
Car 3 occupies Pump 0.
Number of free pumps is 0

Filling for Car 3 started

Attender 2 waiting for a car

CASH COUNTER : Waiting payment confirmation from Car 0

Payment started

Filling completed for Car 2 by Attender 3 using Pump 2

Car 2 left Pump 2 to go to the CASH COUNTER.
Number of free pumps is 1

Car 2 ready to pay

Attender 3 instructs Car 2 to leave Pump 2 and go to CASH COUNTER

Attender 3 waiting for a car

Attender 3 fills gas in Car 3 using Pump 0

Filling completed for Car 1 by Attender 1 using Pump 1

Car 1 left Pump 1 to go to the CASH COUNTER.
Number of free pumps is 2
```

Figure 5.2.3

### 3. Cafeteria Problem

```
        WELCOME TO THE CAFETERIA Thu Nov  5 16:45:29 2020

[ 0 ] - Cook goes to sleep

|-------------------------------------------------------|
|                       CONVEYOR                        |
|               ----------------------                  |
|       TRAYS READY :8                                  |
|       TRAYS AVAILABLE :0                              |
|   COOK                    WAITING LINE                |
|  ---------               ----------------            |
|  Sleeping                   0                        |
|                          Students waiting             |
|                                                       |
|                                                       |
|           CAFETERIA STATISTICS                        |
|       ----------------------------                    |
|       NO. OF TRAYS FILLED      :8                     |
|       NO. OF STUDENTS ARRIVED :0                      |
|       NO. OF TRAYS FETCHED     :0                     |
|-------------------------------------------------------|

[ 2 ] Student no. 1 arrives
```

Figure 5.3.1

```
|-------------------------------------------------------|
|                       CONVEYOR                        |
|               ----------------------                  |
|       TRAYS READY :8                                  |
|       TRAYS AVAILABLE :0                              |
|   COOK                    WAITING LINE                |
|  ---------               ----------------            |
|  Sleeping                   1                        |
|                          Students waiting             |
|                                                       |
|                                                       |
|           CAFETERIA STATISTICS                        |
|       ----------------------------                    |
|       NO. OF TRAYS FILLED      :8                     |
|       NO. OF STUDENTS ARRIVED :1                      |
|       NO. OF TRAYS FETCHED     :0                     |
|-------------------------------------------------------|

[ 3 ] Student no. 1 fetches his tray
```

Figure 5.3.2

```
|-----------------------------------------------------------|
|                         CONVEYOR                          |
|                 -----------------------                   |
|       TRAYS READY :7                                      |
|       TRAYS AVAILABLE :1                                  |
|   COOK                     WAITING LINE                   |
|  ---------                ----------------                |
|  Working                    0                             |
|                           Students waiting                |
|  FILLING TRAY NO. 9                                       |
|                                                           |
|                                                           |
|            CAFETERIA STATISTICS                           |
|       -----------------------------                       |
|       NO. OF TRAYS FILLED      :8                         |
|       NO. OF STUDENTS ARRIVED :1                          |
|       NO. OF TRAYS FETCHED     :1                         |
|-----------------------------------------------------------|

[ 3 ] - Cook is awake
```

Figure 5.3.3

```
|-----------------------------------------------------------|
|                         CONVEYOR                          |
|                 -----------------------                   |
|       TRAYS READY :7                                      |
|       TRAYS AVAILABLE :1                                  |
|   COOK                     WAITING LINE                   |
|  ---------                ----------------                |
|  Working                    0                             |
|                           Students waiting                |
|  FILLING TRAY NO. 9                                       |
|                                                           |
|                                                           |
|            CAFETERIA STATISTICS                           |
|       -----------------------------                       |
|       NO. OF TRAYS FILLED      :8                         |
|       NO. OF STUDENTS ARRIVED :1                          |
|       NO. OF TRAYS FETCHED     :1                         |
|-----------------------------------------------------------|

[ 3 ] Cook starts filling tray no. 9
```

Figure 5.3.4

```
|----------------------------------------------------|
|                       CONVEYOR                      |
|               ----------------------                |
|       TRAYS READY :7                                |
|       TRAYS AVAILABLE :1                            |
|  COOK                     WAITING LINE              |
|  ---------               ----------------           |
|  Working                    0                        |
|                           Students waiting          |
|  FILLING TRAY NO. 9                                 |
|                                                     |
|                                                     |
|           CAFETERIA STATISTICS                      |
|       ----------------------------                  |
|       NO. OF TRAYS FILLED      :8                   |
|       NO. OF STUDENTS ARRIVED :1                    |
|       NO. OF TRAYS FETCHED     :1                   |
|----------------------------------------------------|

[ 4 ] Student no. 2 arrives
```

Figure 5.3.5

## 6. CONCLUSION

Through this project, we have thoroughly understood threads as well as their applications. We have also learnt how to apply threads and synchronization mechanisms such as semaphores to solve real world problems. We have thoroughly studied existing problems such as the Reader/Writer problem and Dining Philosophers problem and gained enough knowledge to successfully solve our own problem statements.

## 7. REFERENCES

[1] Felipe S. Sarmanho, Paulo S.L. Souza, Simone R.S. Souza, and Adenilso S. Sim˜ao, “Structural Testing for Semaphore-Based Multithreaded Programs”, Springer-Verlag, 2008

[2] Ramasamy Satishkumar, “A Study of Synchronization Mechanisms in Unix, windows NT, and Mac OS”

[3] Er.Ankit Gupta, Er. Arpit Gupta, Er. Amit Mishra, “RESEARCH PAPER ON SOFTWARE SOLUTION OF CRITICAL SECTION PROBLEM”, International Journal of Advance Technology & Engineering Research (IJATER), Vol. 1, Issue 1, November 2011

[4] Julie Zelenski, Nick Parlante, “Thread and Semaphore Examples”, Spring, May 2008