**LOVELY PROFESSIONAL UNIVERSITY**

**CA-2**

**Course Title: Operating System**

**Term: Max. Marks:30** **Date of Submission: 30 March,2020**

**By**

**Kotra Pavan Kumar**

**Registration No: 11704910**

**Roll No: A-27**

**Section: EE032**

**GitHub Link:** <https://github.com/pavankotra>

**GitHub Repository :** <https://github.com/pavankotra/pavankotra.git>

**Email Address:**[pavankotra98@gmail.com](mailto:pavankotra98@gmail.com)

**Submitted to: Dr.Aarthi Madam**



**School of Elecronics Communication and Engineering**

**Lovely Professional University**

**Phagwara,Punjab**

EXPLANATION

Solution to Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions:

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

Synchronization Hardware

Many systems provide hardware support for critical section code. The critical section problem could be solved easily in a single-processor environment if we could disallow interrupts to occur while a shared variable or resource is being modified.

In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. Unfortunately, this solution is not feasible in a multiprocessor environment.

Disabling interrupt on a multiprocessor environment can be time consuming as the message is passed to all the processors.

This message transmission lag, delays entry of threads into critical section and the system efficiency decreases.

Mutex Locks

As the synchronization hardware solution is not easy to implement for everyone, a strict software approach called Mutex Locks was introduced. In this approach, in the entry section of code, a LOCK is acquired over the critical resources modified and used inside critical section, and in the exit section that LOCK is released.

As the resource is locked while a process executes its critical section hence no other process can access it.

The classical definitions of wait and signal are:

Wait: Decrements the value of its argument S, as soon as it would become non-negative(greater than or equal to 1).

Signal: Increments the value of its argument S, as there is no more process blocked on the queue.

Properties of Semaphores

It's simple and always have a non-negative Integer value.Works with many processes.Can have many different critical sections with different semaphores.Each critical section has unique access semaphores.Can permit multiple processes into the critical section at once, if desirable.

Types of Semaphores

Semaphores are mainly of two types:

*Binary Semaphore:*

It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a Mutex. A binary semaphore is initialized to 1 and only takes the values 0 and 1during execution of a program.

*Counting Semaphores:*

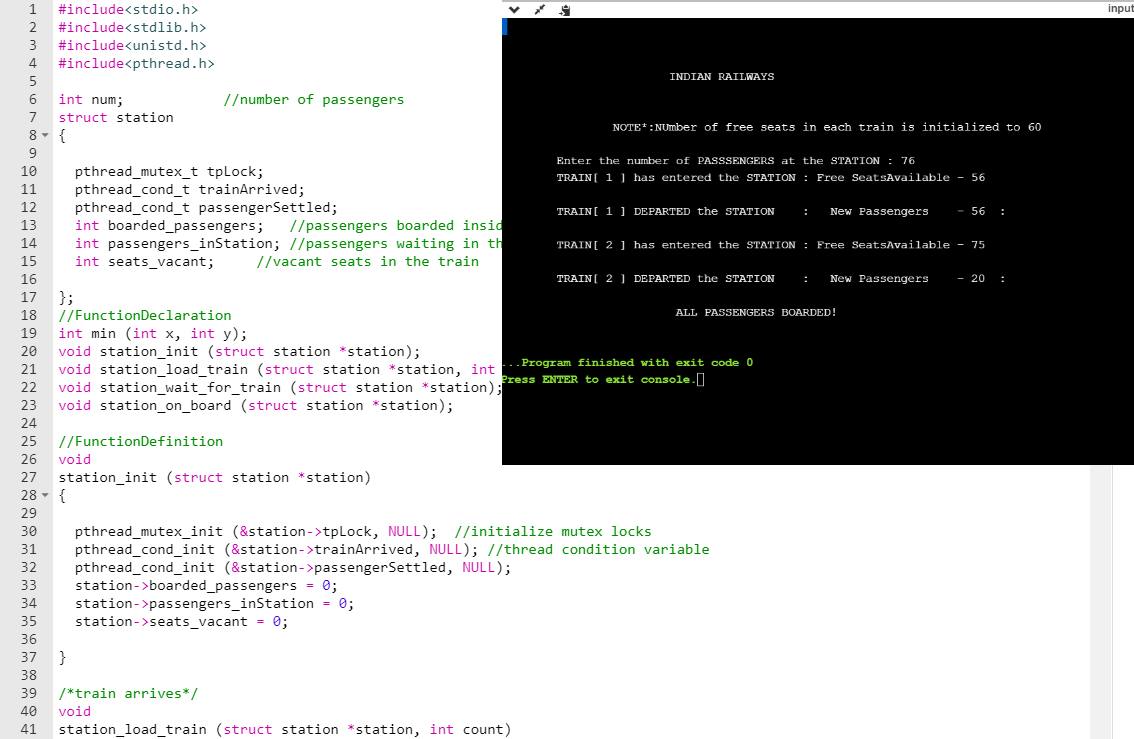
These are used to implement bounded concurrency.

THREADS AND SYNCHRONIZATION CODE

Question 6:

|  |
| --- |
|  |
|  | #include<stdio.h>  #include<stdlib.h> | |
|  | #include<unistd.h> | |
|  | #include<pthread.h> | |
|  |  | |
|  | int num; //number of passengers | |
|  | struct station | |
|  | { | |
|  |  | |
|  | pthread\_mutex\_t tpLock; | |
|  | pthread\_cond\_t trainArrived; | |
|  | pthread\_cond\_t passengerSettled; | |
|  | int boarded\_passengers; //passengers boarded inside the train | |
|  | int passengers\_inStation; //passengers waiting in the station | |
|  | int seats\_vacant; //vacant seats in the train | |
|  |  | |
|  | }; | |
|  | //FunctionDeclaration | |
|  | int min (int x, int y); | |
|  | void station\_init (struct station \*station); | |
|  | void station\_load\_train (struct station \*station, int count); | |
|  | void station\_wait\_for\_train (struct station \*station); | |
|  | void station\_on\_board (struct station \*station); | |
|  |  | |
|  | //FunctionDefinition | |
|  | void | |
|  | station\_init (struct station \*station) | |
|  | { | |
|  |  | |
|  | pthread\_mutex\_init (&station->tpLock, NULL); //initialize mutex locks | |
|  | pthread\_cond\_init (&station->trainArrived, NULL); //thread condition variable | |
|  | pthread\_cond\_init (&station->passengerSettled, NULL); | |
|  | station->boarded\_passengers = 0; | |
|  | station->passengers\_inStation = 0; | |
|  | station->seats\_vacant = 0; | |
|  |  | |
|  | } | |
|  |  | |
|  | /\*train arrives\*/ | |
|  | void | |
|  | station\_load\_train (struct station \*station, int count) | |
|  | { | |
|  | //returns when there are no passengers or train is full | |
|  |  | |
|  | pthread\_mutex\_lock (&station->tpLock); | |
|  | station->seats\_vacant = count; | |
|  | while (station->seats\_vacant > 0 && station->passengers\_inStation > 0) | |
|  | { | |
|  |  | |
|  | pthread\_cond\_broadcast (&station->trainArrived); //similar to used signal and used to inform several threads which are waiting | |
|  | pthread\_cond\_wait (&station->passengerSettled, &station->tpLock); | |
|  |  | |
|  | } | |
|  |  | |
|  | station->seats\_vacant = 0; | |
|  | pthread\_mutex\_unlock (&station->tpLock); | |
|  | } | |
|  |  | |
|  |  | |
|  | //passenger arrives | |
|  |  | |
|  | void | |
|  | station\_wait\_for\_train (struct station \*station) | |
|  | { | |
|  | //return when there are enough available seats and train is in the station | |
|  | pthread\_mutex\_lock (&station->tpLock); | |
|  | station->passengers\_inStation++; | |
|  |  | |
|  | while (station->boarded\_passengers == station->seats\_vacant) | |
|  | { | |
|  | pthread\_cond\_wait (&station->trainArrived, &station->tpLock); | |
|  | } | |
|  |  | |
|  | station->boarded\_passengers++; | |
|  | station->passengers\_inStation--; | |
|  | pthread\_mutex\_unlock (&station->tpLock); | |
|  | } | |
|  |  | |
|  |  | |
|  |  | |
|  | //passenger boarded | |
|  |  | |
|  | void | |
|  | station\_on\_board (struct station \*station) | |
|  | { | |
|  | //to inform the train that it is on board | |
|  | pthread\_mutex\_lock (&station->tpLock); | |
|  | station->boarded\_passengers--; | |
|  | station->seats\_vacant--; | |
|  |  | |
|  | if ((station->seats\_vacant == 0) || (station->boarded\_passengers == 0)) | |
|  | { | |
|  | pthread\_cond\_signal (&station->passengerSettled); | |
|  | } | |
|  |  | |
|  | pthread\_mutex\_unlock (&station->tpLock); | |
|  |  | |
|  | } | |
|  |  | |
|  | volatile int threads\_completed = 0; | |
|  | void \* | |
|  | passenger\_thread (void \*arg) | |
|  | { | |
|  | struct station \*station = (struct station \*) arg; | |
|  | station\_wait\_for\_train (station); | |
|  | threads\_completed++; | |
|  | return NULL; | |
|  | } | |
|  |  | |
|  | struct TrainLoaded\_Para | |
|  | { | |
|  | struct station \*station; | |
|  | int free\_seats; | |
|  | }; | |
|  |  | |
|  | volatile int return\_LoadTrain = 0; | |
|  |  | |
|  | void \* | |
|  | load\_train\_thread (void \*args) | |
|  | { | |
|  | struct TrainLoaded\_Para \*temp = (struct TrainLoaded\_Para \*) args; | |
|  | station\_load\_train (temp->station, temp->free\_seats); | |
|  | return\_LoadTrain = 1; | |
|  | return NULL; | |
|  | } | |
|  |  | |
|  | //finds the minimum value among x and y | |
|  | #ifndef MIN | |
|  | #define MIN(\_x,\_y) ((\_x) < (\_y)) ? (\_x) : (\_y) | |
|  | #endif | |
|  | //main function starts from here | |
|  | int | |
|  | main () | |
|  | { | |
|  | struct station station; | |
|  | station\_init (&station); | |
|  |  | |
|  | srandom (getpid () ^ time (NULL)); //generates random numbers | |
|  | int i; | |
|  | printf ("\n\n\n\t\t\tINDIAN RAILWAYS\n\n"); | |
|  | printf | |
|  | ("\n\t\tNOTE\*:NUmber of free seats in each train is initialized to 60"); | |
|  | printf ("\n\n\tEnter the number of PASSSENGERS at the STATION : "); | |
|  | scanf ("%d", &num); | |
|  | if (num < 0) | |
|  | { | |
|  | printf | |
|  | (" \tYou have entered number of passengers as %d which is not possible.\n", | |
|  | num); | |
|  | printf (" \tPlease enter a valid number!!\n"); | |
|  | scanf ("%d", &num); | |
|  | } | |
|  | if (num == 0) | |
|  | { | |
|  | printf (" \t NO PASSENGERS in the STATION!!\n\n"); | |
|  | return 0; | |
|  | } | |
|  | const int total\_Passngrs = num; | |
|  | int remaining\_Passngrs = total\_Passngrs; | |
|  | for (i = 0; i < total\_Passngrs; i++) | |
|  | { | |
|  | pthread\_t tid; | |
|  | int ret = pthread\_create (&tid, NULL, passenger\_thread, &station); | |
|  | } | |
|  |  | |
|  | int total\_Passngrs\_boarded = 0; | |
|  | const int tot\_FreeSeats\_PerTrain = 100; | |
|  | int pass = 0; | |
|  | int j = 1, p = 1; | |
|  | while (remaining\_Passngrs > 0) | |
|  | { | |
|  |  | |
|  | int free\_seats = random () % tot\_FreeSeats\_PerTrain; | |
|  |  | |
|  | printf | |
|  | (" \tTRAIN[ %d ] has entered the STATION : Free SeatsAvailable - %d\n\n", | |
|  | j, free\_seats); | |
|  | j++; | |
|  | return\_LoadTrain = 0; | |
|  | struct TrainLoaded\_Para args = { &station, free\_seats }; | |
|  | pthread\_t lt\_tid; | |
|  | int ret = pthread\_create (&lt\_tid, NULL, load\_train\_thread, &args); | |
|  | if (ret != 0) | |
|  | { | |
|  | perror ("pthread\_create"); | |
|  | exit (1); | |
|  | } | |
|  |  | |
|  | int threads\_to\_reap = MIN (remaining\_Passngrs, free\_seats); | |
|  | int threads\_reaped = 0; | |
|  |  | |
|  | while (threads\_reaped < threads\_to\_reap) | |
|  | { | |
|  |  | |
|  | if (return\_LoadTrain) | |
|  | { | |
|  | exit (1); | |
|  | } | |
|  | if (threads\_completed > 0) | |
|  | { | |
|  | if ((pass % 2) == 0) | |
|  | usleep (random () % 2); | |
|  | threads\_reaped++; | |
|  | station\_on\_board (&station); | |
|  | threads\_completed++; | |
|  |  | |
|  | } | |
|  | } | |
|  |  | |
|  | remaining\_Passngrs -= threads\_reaped; | |
|  | total\_Passngrs\_boarded += threads\_reaped; | |
|  | printf | |
|  | (" \tTRAIN[ %d ] DEPARTED the STATION : New Passengers - %d :\n\n", | |
|  | p, threads\_to\_reap); | |
|  |  | |
|  | pass++; | |
|  | p++; | |
|  | } | |
|  |  | |
|  | if (total\_Passngrs\_boarded == total\_Passngrs) | |
|  | { | |
|  | printf ("\t\t\t ALL PASSENGERS BOARDED!\n"); | |
|  | return 0; | |
|  | } | |
|  | } | |
|  | |
|  | | #include<stdio.h>  #include<stdlib.h> |
|  | | #include<unistd.h> |
|  | | #include<pthread.h> |
|  | |  |
|  | | void station\_init (struct station \*station); |
|  | | void station\_load\_train (struct station \*station, int count); |
|  | | void station\_wait\_for\_train (struct station \*station); |
|  | | void station\_on\_board (struct station \*station); |
|  | |  |
|  | | struct station |
|  | | { |
|  | | /\*Statements \*/ |
|  | | }; |
|  | |  |
|  | | void |
|  | | station\_init (struct station \*station) |
|  | | { |
|  | | /\*statements to execute \*/ |
|  | | } |
|  | |  |
|  | | void |
|  | | station\_load\_train (struct station \*station, int count) |
|  | | { |
|  | | /\*statements to execute \*/ |
|  | | } |
|  | |  |
|  | | void |
|  | | station\_wait\_for\_train (struct station \*station) |
|  | | { |
|  | | /\*statements to execute \*/ |
|  | | } |
|  | |  |
|  | | void station\_on |
|  | | board (struct station \*station) |
|  | | { |
|  | | /\*statements to execute \*/ |

**I.**The output window shows that the number of passengers to be in the train and the number of passengers to be seated.



**THE END.**