**Student Name: Ayush Baiswar**

**Student ID: 11808611**

**Email Address:** [**ayushbaiswar908@gmail.com**](mailto:ayushbaiswar908@gmail.com)

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

**Problem:**

**Ques25.**

Indian Rail has decided to improve its efficiency by automating not just its train but its passengers. Each passenger and each train is controlled by thread. You have hired to write synchronization that will guarantee orderly loading of train. You must define a struct station, plus several function describe below.

When train arrives in the station and has opened its doors, invoke the function station\_load\_train(struct station \*station ,int count) where count indicates how many seats are available in train. The function must not return until train is satisfactorily loaded (all passengers are in their seats, and either the train is full or all waiting passengers have boarded).

When all passengers arrive in the station invokes the function

Station\_wait\_for\_train(struct station \*station)

This function must not return until a train is in the station(call tostation\_load\_train is in the progess)and there are enoughfree seats on this train for the passengers on board the train and into a seats, and either or the train is full or all waiting passengers have boarded). When a passengers arrive in the station it first invokes the function station\_wait\_for\_train(struct station \*station)

This function must not return until a train is in the station(call to station\_load\_train is in progress) and are enough free seats on the train for this passengers to sit down. Once the function return, the passengers robort will move the passengers in the boared into the seat(you do not to worry about the mechanism works).once the passenger is seated it will call the function

Station \_on\_boared(struct station \*station)

To let the train know that’s in the boar

d.

Create a file IndianRail.c that contains a decelaration for struct station\_int which will invoke to intilaze the station objects when Indian railway boots. In addition

You must write solution in c using locks and condition variable:

**Ans**

We can solve this by using algorithms of operating system

1. Round Robin Scheduling Algorithm

**Algorithm:**

Step1: Start the process

Step2: Initializes semaphore SEMA to VALUE. A semaphore is

nonnegative integer along with two atomic operators for

manipulating it.

Step3: Down or "P" operation on a semaphore. Waits for SEMA's value to become positive and then atomically decrements it.

Step4: Keep the waiting list sorted, highest priority first.

list\_insert\_ordered (&sema->waiters, &thread\_current ()->elem,more\_prio, NULL)thread\_block ().

Step5: Down or "P" operation on a semaphore, but only if thesemaphore is not already 0. Returns true if the semaphore isdecremented, false otherwise.

Step6: Up or "V" operation on a semaphore. Increments SEMA's valueand wakes up one thread of those waiting for SEMA, if any.This function may be called from an interrupt handler.sWithin an interrupt context, sema\_up () always returns.

**Description**

To implement the above problem we have to initializes semaphore SEMA to value.Wait for value to become positive then increment it.

**Code**

include "threads/synch.h"

#include <stdio.h>

#include <string.h>

#include "threads/interrupt.h"

#include "threads/thread.h"

void

sema\_init (struct semaphore \*sema, unsigned value)

{

ASSERT (sema != NULL);

sema->value = value;

list\_init (&sema->waiters);

}

void

sema\_down (struct semaphore \*sema)

{

enum intr\_level old\_level;

ASSERT (sema != NULL);

ASSERT (!intr\_context ());

old\_level = intr\_disable ();

while (sema->value == 0)

{

list\_insert\_ordered (&sema->waiters, &thread\_current ()->elem,

more\_prio, NULL);

thread\_block ();

}

sema->value--;

intr\_set\_level (old\_level);

}

bool

sema\_try\_down (struct semaphore \*sema)

{

enum intr\_level old\_level;

bool success;

ASSERT (sema != NULL);

old\_level = intr\_disable ();

if (sema->value > 0)

{

sema->value--;

success = true;

}

else

success = false;

intr\_set\_level (old\_level);

return success;

}

void

sema\_up (struct semaphore \*sema)

{

enum intr\_level old\_level;

bool yield = false;

ASSERT (sema != NULL);

old\_level = intr\_disable ();

if (!list\_empty (&sema->waiters))

{

struct thread \*t = list\_entry (list\_pop\_front (&sema->waiters),

struct thread, elem);

thread\_unblock (t);

if (t->priority > thread\_current ()->priority)

yield = true;

}

sema->value++;

intr\_set\_level (old\_level);

if (yield)

{

if (!intr\_context ())

thread\_yield ();

else

intr\_yield\_on\_return ();

}

}

static void sema\_test\_helper (void \*sema\_);

void

sema\_self\_test (void)

{

struct semaphore sema[2];

int i;

printf ("Testing semaphores...");

sema\_init (&sema[0], 0);

sema\_init (&sema[1], 0);

thread\_create ("sema-test", PRI\_DEFAULT, sema\_test\_helper, &sema);

for (i = 0; i < 10; i++)

{

sema\_up (&sema[0]);

sema\_down (&sema[1]);

}

printf ("done.\n");

}

static void

sema\_test\_helper (void \*sema\_)

{

struct semaphore \*sema = sema\_;

int i;

for (i = 0; i < 10; i++)

{

sema\_down (&sema[0]);

sema\_up (&sema[1]);

}

}

void

lock\_init (struct lock \*lock)

{

ASSERT (lock != NULL);

lock->holder = NULL;

sema\_init (&lock->semaphore, 1);

}

void

lock\_acquire (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

ASSERT (!lock\_held\_by\_current\_thread (lock));

if (!lock\_try\_acquire (lock))

{

sema\_down (&lock->semaphore);

lock->holder = thread\_current ();

}

}

bool

lock\_try\_acquire (struct lock \*lock)

{

bool success;

ASSERT (lock != NULL);

ASSERT (!lock\_held\_by\_current\_thread (lock));

success = sema\_try\_down (&lock->semaphore);

if (success)

lock->holder = thread\_current ();

return success;

}

void

lock\_release (struct lock \*lock)

{

ASSERT (lock != NULL);

ASSERT (lock\_held\_by\_current\_thread (lock));

lock->holder = NULL;

sema\_up (&lock->semaphore);

}

bool

lock\_held\_by\_current\_thread (const struct lock \*lock)

{

ASSERT (lock != NULL);

return lock->holder == thread\_current ();

}

struct semaphore\_elem

{

struct list\_elem elem;

struct semaphore semaphore;

};

void

cond\_init (struct condition \*cond)

{

ASSERT (cond != NULL);

list\_init (&cond->waiters);

}

void

cond\_wait (struct condition \*cond, struct lock \*lock)

{

struct semaphore\_elem waiter;

ASSERT (cond != NULL);

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

ASSERT (lock\_held\_by\_current\_thread (lock));

sema\_init (&waiter.semaphore, 0);

list\_push\_back (&cond->waiters, &waiter.elem);

lock\_release (lock);

sema\_down (&waiter.semaphore);

lock\_acquire (lock);

}

void

cond\_signal (struct condition \*cond, struct lock \*lock UNUSED)

{

ASSERT (cond != NULL);

ASSERT (lock != NULL);

ASSERT (!intr\_context ());

ASSERT (lock\_held\_by\_current\_thread (lock));

if (!list\_empty (&cond->waiters))

sema\_up (&list\_entry (list\_pop\_front (&cond->waiters),

struct semaphore\_elem, elem)->semaphore);

}

void

cond\_broadcast (struct condition \*cond, struct lock \*lock)

{

ASSERT (cond != NULL);

ASSERT (lock != NULL);

while (!list\_empty (&cond->waiters))

cond\_signal (cond, lock);

}