

Name: - Apoorva.P

Registration:-11701463

Roll.no: - A-18

Section:-EE029

Email Id: - apoorvapkumar7@gmail.com

GitHub link:-

Submitted To:

Prabhdeep Kaur.

* **CODE:**

(i). **Programming Scenario:**

This problem demonstrates the use of semaphores to coordinate three types ofprocesses.6 Santa Claus sleeps in his shop at the North Pole and can only be wakened by either (1) all nine reindeer being back from their vacation in the South Pacific, or (2) some of the elves having difficulties making toys; to allow Santa to get some sleep, the elves can only wake him when three of them have problems. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop’s door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready. (It is assumed that the reindeer do not want to leave the tropics, and therefore they stay there until the last possible moment.) The last reindeer to arrive must get Santa while the others wait in a warming hut before being harnessed to the sleigh. Using synchronization tools like locks, semaphores and monitors provide a solution to this problem.

**Understanding the concept:**

The following methods can be called by:

* Santa: helpElves( ), prepareSleigh( )
* Reindeer: getHitch( )
* Elves: getHelp( )

The methods getHitch ( ) and getHelp ( ) are thread safe and can be called outside the critical sections. As per instruction give the problem should be solved using Semaphores.

* The Santa must call prepareSleigh ( ) after the arrival of ninth reindeer and all these nine reindeers must call getHitch ( ).
* The Santa must invoke helpElves( ) after the arrival of 3rd Elf and all the three elves invokes getHelp( )
* All the three elves must invoke getHelp ( ) before any other elves enters.

Initialization of variables (global) required for the program:

* int num\_elf =0;
* int num\_reindeer=0;
* santaSema=Semaphore(0);
* reindeerSema=Semaphore(0);
* elfSema=Semaphore(1);
* lock=Semaphore(1);

(ii). **Algorithm of the solution**

Implementation of function Santa( ):

The Santa’s code runs an infinite loop. This code checks two conditions dealing with elves and reindeers. If nine reindeers are waiting then Santa prepares sleigh and signals reindeer semaphore nine times and allows reindeer to invoke getHitch( ).If the elves are waiting then Santa will be invoking helpElves( ).The counter decrementation is not required as the elves do that on their way out.

* **Algorithm:**

1. while (TRUE) do
2. Sleep
3. if (reindeer at door) then
4. tell reindeer to roust other reindeer
5. ready sleigh
6. deliver toys
7. else
8. for (each elve) do
9. solve problem
10. send back to work
11. end of for
12. end of if
13. end of while

Implementation of the function ReindeerComeHome( ):

The ninth reindeer signals Santa and joins other reindeer waiting on the reindeerSema. All the waiting reindeer execute getHitch( ) after Santa signals.

* **Algorithm:**

1. while (TRUE) do
2. while(not tired of South Pacific) do vacation in South Pacific
3. end of while
4. return to North Pole
5. if last reindeer to return then
6. wake Santa
7. else
8. wait in warming hut
9. end of if
10. hook up to sleigh
11. fly Santa around world
12. return to South Pacific
13. end of while

Implementation of the function ElfRequestingHelp():

The first two elves release the elfSema at the same time they release the lock but the last elf holds the elfSema that prevents other three elves to enter till all the two elves invoke getHelp().The last elf to leave releases the elfSema that allows the next group of elves to enter and request help from Santa.

* **Algorithm:**

1. while (TRUE) do
2. while(no problem) do
3. build toy
4. end of while
5. ElfCount++
6. ElfCount < 3
7. wait
8. else if ElfCount = 3 then
9. gather other 2 elves with problems
10. ElfCount -= 3
11. End of if
12. if a group of elves went to see Santa
13. have your group wait
14. end of if
15. see Santa
16. On returning, send another group of elves if there is one waiting
17. end of while

(iii). **Program and Solution:**

* **Test Cases**

1. **Santa’s Case:**

santSema.wait()

mutex.wait()

if (num\_reindeer ==9)

{

prepareSleigh();

reindeerSema.signal(9);

}

elseif(num\_elves = =3)

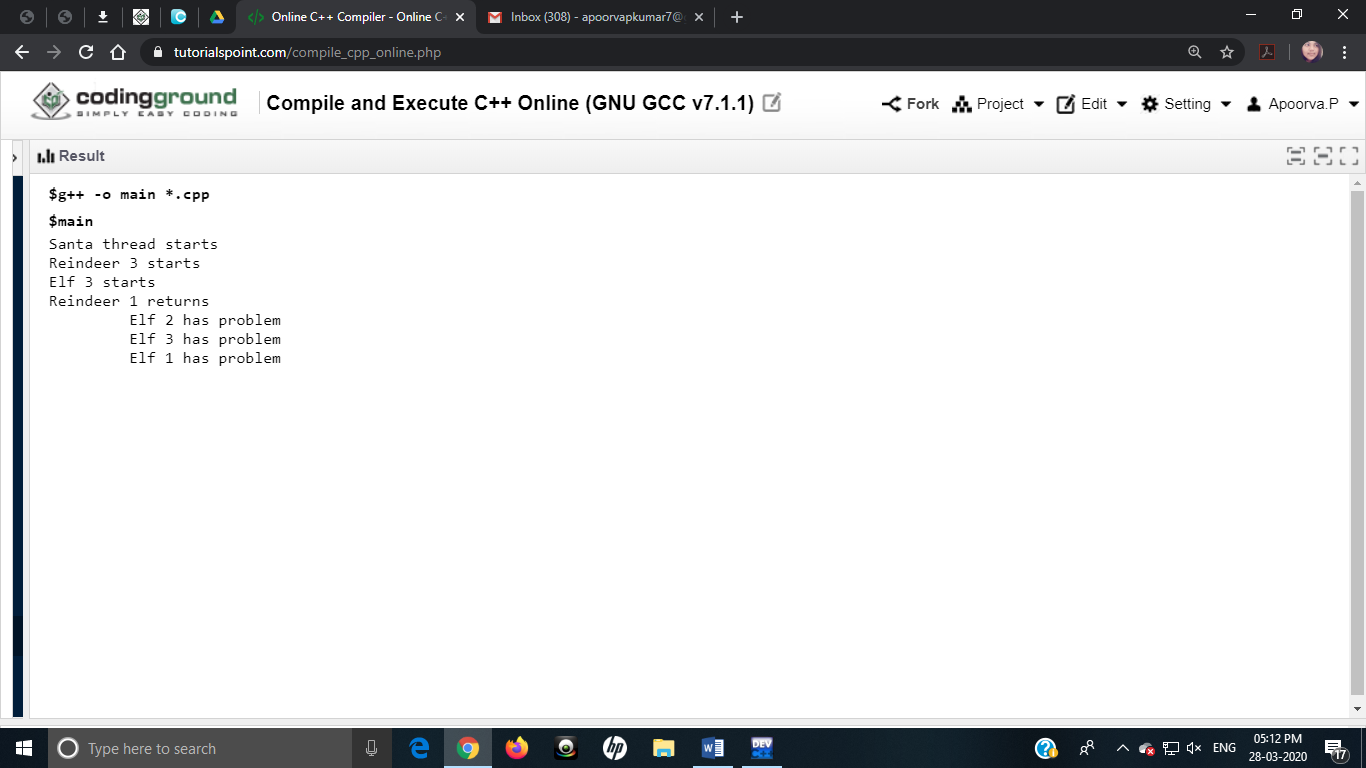
{

helpElves();

elfSemas.signal(3)

}

mutex.signal()



1. **Reindeer’s Case:**

mutex.wait()

num\_reindeer +=1

if(num\_reindeer == 9)

{

santaSema.signal()

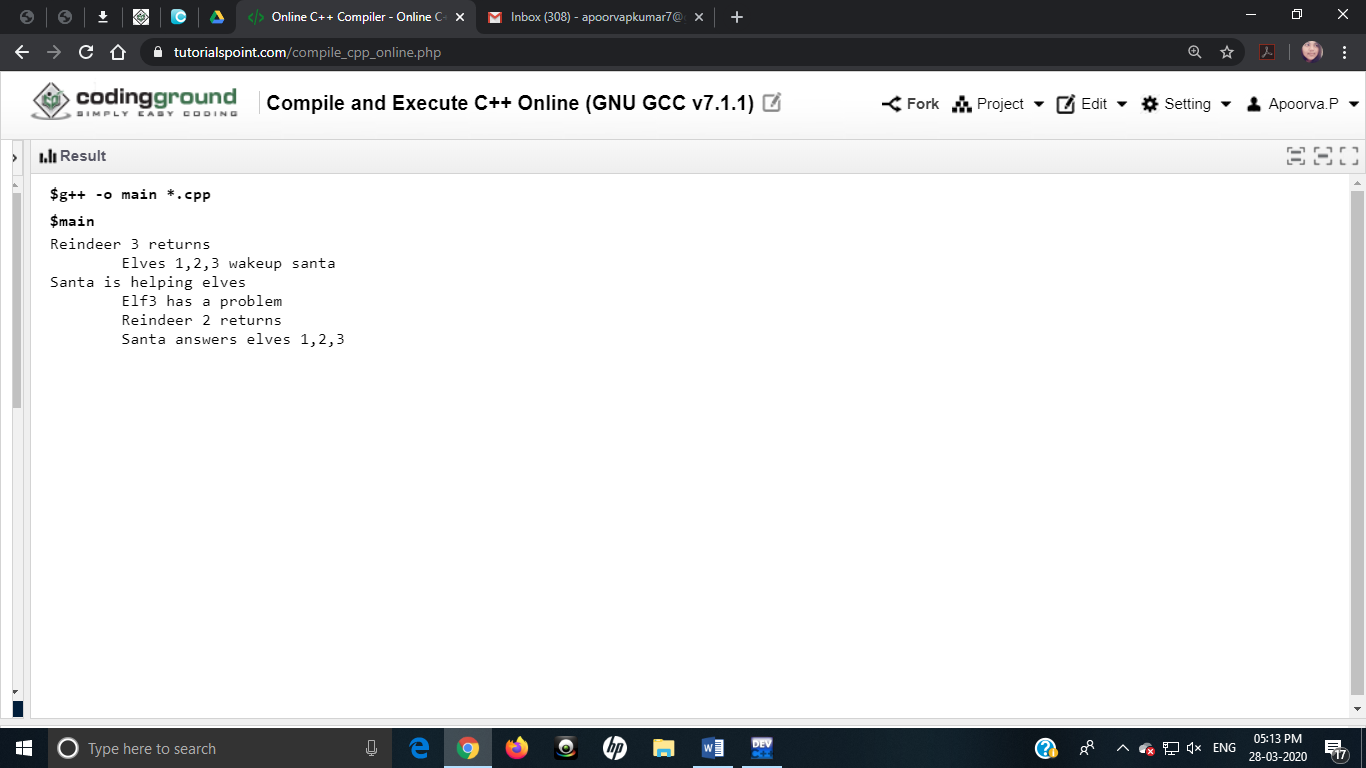
}

mutex.signal()

reindeerSema.wait()

getHitch()

}



1. **Elves’s Case:**

elf\_mutex.wait()

mutex.wait()

num\_elves += 1

if (num\_elves == 3)

{

santaSem.signal()

}

else

{

elfSema.signal()

}

mutex.signal()

elfSema.wait()

getHelp()

mutex.wait()

num\_elves - = 1

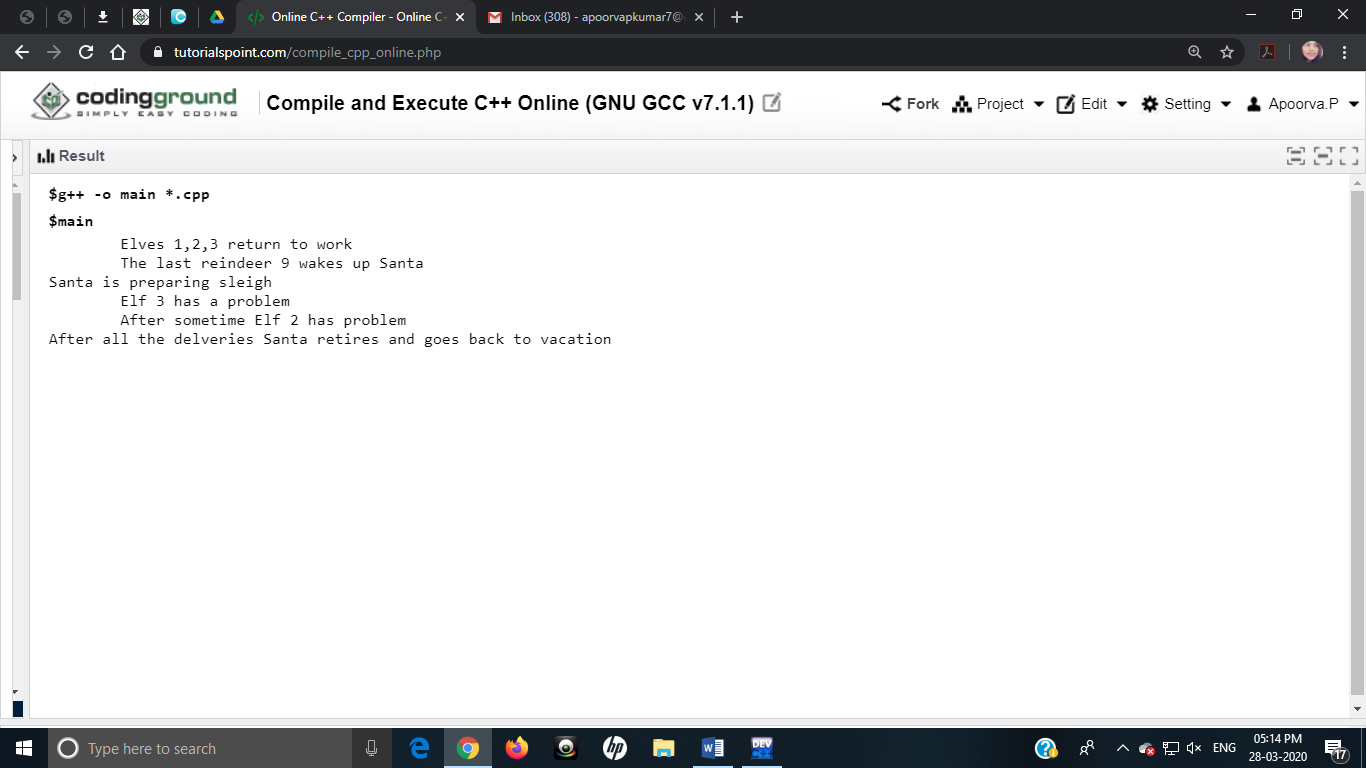
if (num\_elves == 0)

{

elfSema.signal()

}

Mutex.signal()



* **Program Code:**

#include<conio.h>

#include<stdlib.h>

#include<pthread.h>

#include<iostream>

Using namespace std;

int num\_elves=0;

int num\_reindeer=0;

int santaSema=0;

int reindeerSema=0;

int elfSema=1;

int lock=1;

void main()

{

void santa()

{

while(1)

{

santaSema.p();

lock.p();

cout<<"Santa thread starts\n";

cout<<"Reindeer 3 starts\n";

if (num\_reindeer==9)

{

prepareSleigh();

reindeerSema.v();

num\_reindeer-=9;

cout<<"Elf 3 starts\n";

cout<<"Reindeer 1 returns\n";

}

elseif(num\_elves==3)

helpElves();

}

lock.v();

cout<<"\t Elf 2 has problem\n";

cout<<"\t Elf 3 has problem\n";

cout<<"\t Elf 1 has problem\n";

}

ReindeerComesHome()

{

lock.p

}

ReindeerComesHome()

{

lock.P();

num\_reindeer += 1;

cout<<"Reindeer 3 returns\n";

cout<<"\tElves 1,2,3 wakeup santa\n";

if (num\_reindeer == 10)

{

santaSema.V();

}

lock.v();

cout<<"Santa is helping elves\n"

cout<<"\tElf3 has a problem\n";

reindeerSema.p();

getHitch();

cout<<"\tReindeer 2 returns\n";

cout<<"\tSanta answers elves 1,2,3\n";

}

ElfRequestsHelp()

{

elfSem.P();

lock.P();

num\_elves += 1;

cout<<"\tElves 1,2,3 return to work \n";

cout<<"\tThe last reindeer 9 wakes up Santa \n";

if (num\_elves == 5)

{

santaSemsa.v();

cout<<"Santa is preparing sleigh\n";

}

else

{

elfSem.v();

}

lock.V();

getHelp();

lock.P();

cout<<"\tElf 3 has a problem\n";

num\_elves - = 1;

if (num\_elves == 0)

{

elfSem.V();

}

lock.V();

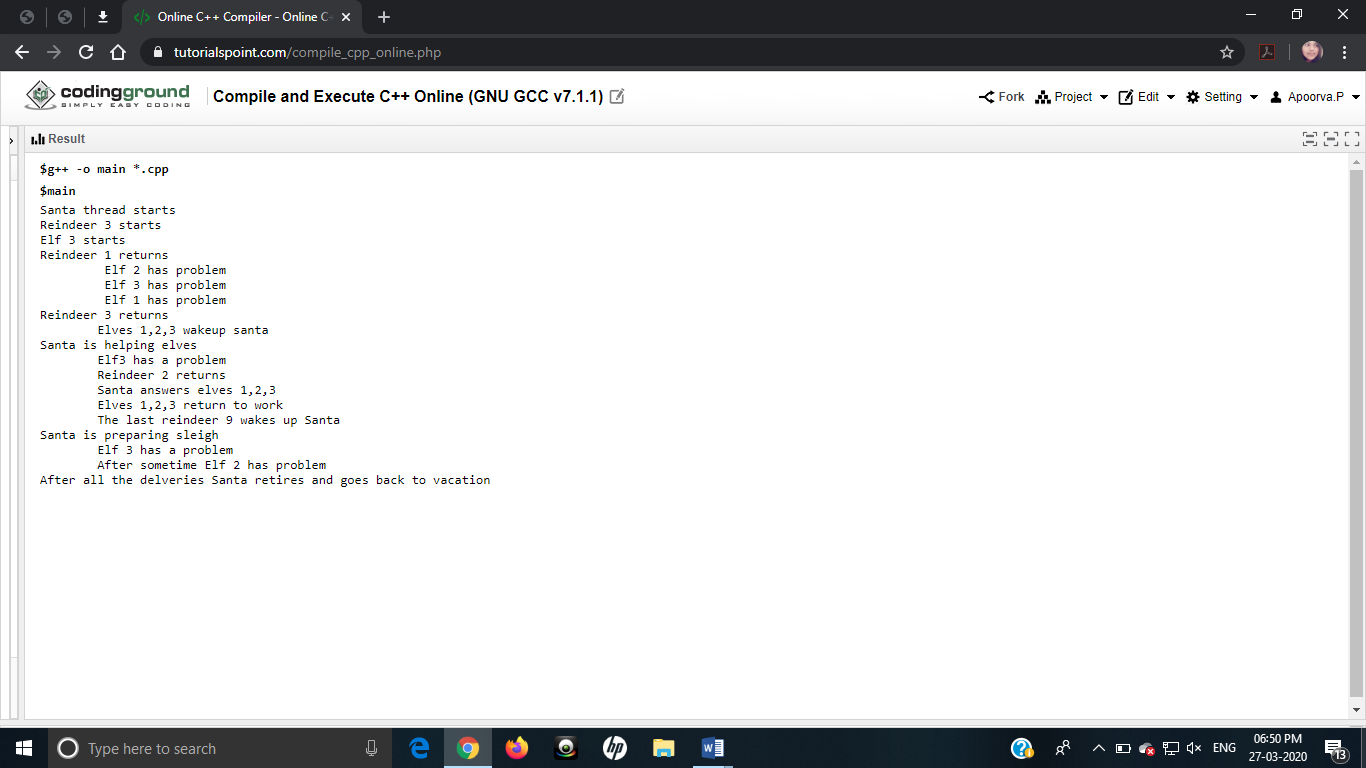
cout<<"\tAfter sometime Elf 2 has problem\n";

cout<<"After all the delveries Santa retires and goes back to vacation\n";

}

}

}



(iv). **Algorithms used in solving the problem:**

In solving the problem we have made use of algorithms of semaphores and mutex.

* **Semaphores:**

In solving the above problem we make use of semaphores.

Semaphores: Using semaphore is a very significant technique to manage concurrent processes by using a simple integer value. It is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

* It is a object that consists of a counter, a waiting list of two methods: Signal and Wait.
* **Algorithm of Semaphore Method Wait:**

void wait(sem s)

{

s.count - -;

if(s.count<0)

{

add the caller to the waiting list;

block( );

}

}

* **Algorithm of Semaphore Method Signal:**

void signal(sem s)

{

s.count ++;

if(s.count<=0)

{

remove a process from the waiting list;

resume(p );

}

}

* **Need and Usage of Semaphores:**

1. In counting semaphore – integer value can range over an unrestricted domain.
2. In Binary semaphore – integer value can range between 0 & 1 only.
3. In implementing a counting semaphore S as binary semaphores.
4. Can solve various synchronization problems.

Ex: Let P1 and P2 that happen before S1 and S2 then :

Lets create a semaphore ‘sync’ initialized to 0

P1:

S1;

signal(sync);

P2:

wait(sync);

S2;

* **Mutex Lock:**

We also made use of mutex lock in solving the given problem

* Mutex: A mutual exclusion (mutex) is a program object that prevents simultaneous access to any shared resource. Only one thread or process owns the mutex at a time that acts as a lock which prevents other threads or process from accessing resources concurrently
* A mutex lock is a binary variable that provides locking mechanism and also to provide mutual exclusion to a part(section) of code i.e., only one process can work on a particular code part(section) at a time.
* Mutex lock is the most basic synchronization tool available that comes into picture when two threads work on same data at same time on which it acts as a lock.
* **Peterson’s Algorithm:**  The mutex locks used in solving the problem makes use of Peterson’s Algorithm (or Peterson’s solution). It can be defined as a concurrent programming algorithm for mutex. It allows two or more process to share a single resource without any conflict and only shared memory for communication.
* **Peterson’s Algorithm:**

flag[1]=true;

do

{

flag[0] = T;

turn=1;

while(flag[1] && turn = =1)

Critical Section

flag[0] = F;

Reminder Section

}while[1];

* **Need and Usage of Mutex Locks and Peterson’s Algorithm:**

1. Both the mutex locks and Peterson’s algorithm is used in solving problems of critical section and synchronization.
2. A mutex can be used as a lockable object designed to signal when critical sections of code need exclusive access and in preventing other threads from concurrently accessing and executing same memory locations.
3. Peterson’s Algorithm (or Peterson’s Solution) is used in synchronizing two process by making use two variables a bool array flag and an int variable turn.

***THE END***