Operating Systems
CSE316

Report
Assignment Simulation
Based

06. Programming Problem Operating Systems By Galvin 9th Edition

5.41 A barrier is a tool for synchronizing the activity of a number of threads. When a thread reaches a barrier point, it cannot proceed until all other threads have reached this point as well. When the last thread reaches the barrier point, all threads are released and can resume concurrent execution. Assume that the barrier is initialized to N —the number of threads that must wait at the barrier point:

```
                    init(N);
```

Each thread then performs some work until it reaches the barrier point:

```
                /* do some work for awhile */
                    barrier point();
                /* do some work for awhile */
```

Using synchronization tools described in this chapter, construct a barrier that implements the following API :

- int init(int n) —Initializes the barrier to the specified size.
- int barrier point(void) —Identifies the barrier point. All threads are released from the barrier when the last thread reaches this point.

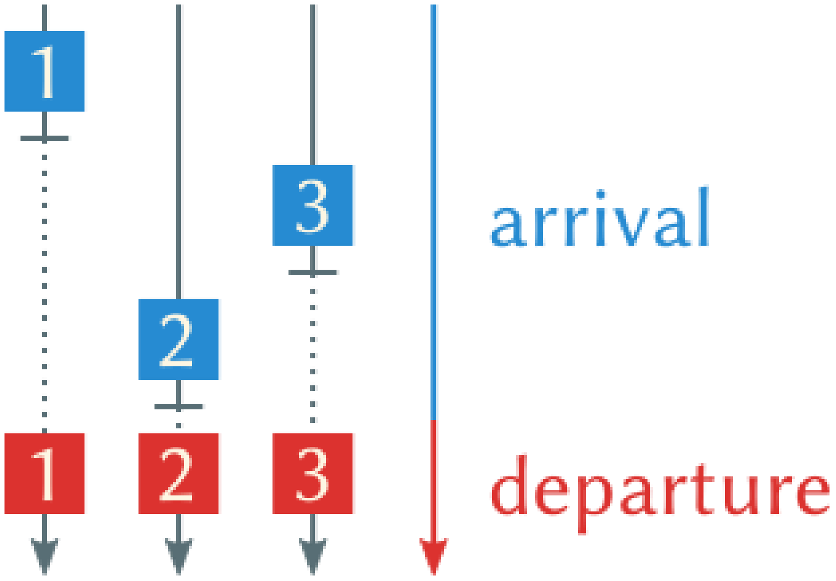
The return value of each function is used to identify error conditions. Each function will return 0 under normal operation and will return −1 if an error occurs. A testing harness is provided in the source code download to test your implementation of the barrier.

Student Name: C Sunilkumar
Student ID: 11706257
Section No: EE033
Roll No.: B37
Email Address: **sunilkumar12499@gmail.com**
GitHub: https://github.com/sunilkumar-c/OS-Assignment_2020

Barriers

a barrier is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

A barrier is a method to implement synchronization. Synchronization ensures that concurrently executing threads or processes do not execute specific portions of the program at the same time. When a barrier is inserted at a specific point in a program for a group of threads [processes], any thread [process] must stop at this point and cannot proceed until all other threads [processes] reach this barrier.



Algorithm:

1. initialize barrier_size and thread_count;
2. create threads
3. threads doing some work
4. threads waiting at the barrier.
5. barrier is released when last thread comes at the thread.
6. all threads complete thier task and exit.
7. exit.

Complexity:

O (n) complexity. “n” is no of thread_count.

PROGRAMMING CODE

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


pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  finish_cond  = PTHREAD_COND_INITIALIZER;
int barrier = 0;
int thread_count;
int barrier_size;
int counter=0;
int invoke_barrier = 0;



/*
 * params : number of threads a process is creating.
 * returns : none.
 *
 * Initialize barrier with total number of threads.
 */
void barrier_init (int n_threads)
{
   if (thread_count < barrier_size) {barrier = thread_count; return;}
   barrier = n_threads;
}



/*
 * params: none.
 * returns: -1 on failure, 0 on success.
 * decrement the count by 1.
 *
 */
int decrement ()
{
   if (barrier == 0) {

      return 0;
   }

   if(pthread_mutex_lock(&lock)! =0)
   {
      Perror ("Failed to take lock.");
      return -1;
   }

   barrier--;
```

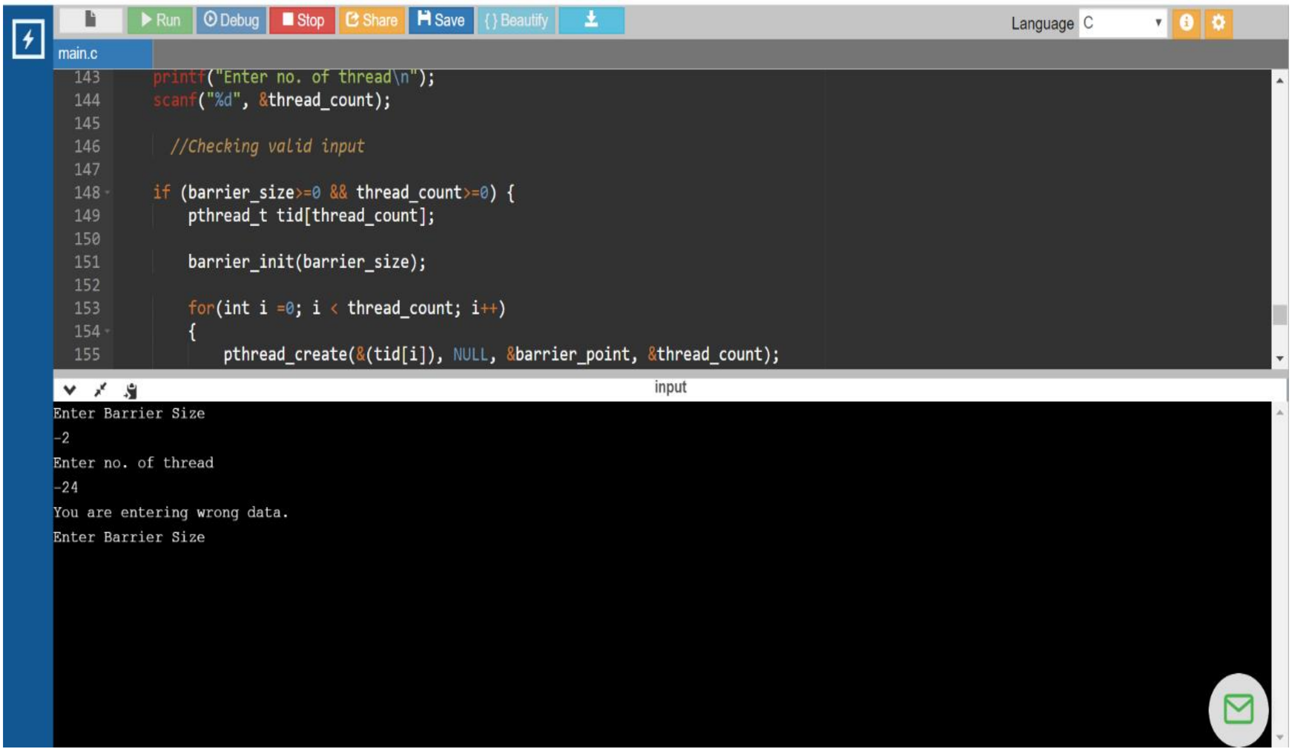
```
    if(pthread_mutex_unlock(&lock)! =0)
    {
        Perror ("Failed to unlock.");
        return -1;
    }

    return 0;
}




/*
 *  params: none.
 *  returns: int: 0 on sucess, -1 on failure.
 *
 *
 *  wait for other threads to complete.
 */
int wait_barrier ()
{
    If (decrement () < 0)
    {
        return -1;
    }

    while (barrier)
    {
        if(pthread_mutex_lock(&lock)! =0)
        {
            Perror ("\n Error in locking mutex");
            return -1;
        }

        If (pthread_cond_wait (&finish_cond, &lock)! =0)
        {
            Perror ("\n Error in cond wait.");
            return -1;
        }
    }

    /*
     * last thread will execute this.
     */
    If (0 == barrier)
    {
        if(pthread_mutex_unlock(&lock)! =0)
        {
            Perror ("\n Error in locking mutex");
            return -1;
        }
        if(pthread_cond_signal(&finish_cond)! =0)
```

```
      {
         Perror ("\n Error while signaling.");
         return -1;
      }
   }

   return 0;
}

void * barrier_point (void *numthreads)
{

    int r = rand () % 5;

    printf ("\nThread %d \nPerforming init task of length %d sec\n”, ++counter, r);
    sleep(r);

    wait_barrier ();
    if (barrier_size! =0) {
     if ((thread_count - (invoke_barrier++)) % barrier_size == 0) {
       printf ("\nBarrier is Released\n");
     }
     printf ("\nI am task after barrier\n");

    }
    //printf ("Thread completed job.\n");

    return  NULL;
}




int main ()
{

   Printf ("Enter Barrier Size\n");
   Scanf ("%d", &barrier_size);

   Printf ("Enter no. of thread\n");
   scanf ("%d", &thread_count);

     //Checking valid input

   if (barrier_size>=0 && thread_count>=0) {
      pthread_t tid [thread_count];

      barrier_init(barrier_size);

      for (int i =0; i < thread_count; i++)
      {
         pthread_create(&(tid[i]), NULL, &barrier_point, &thread_count);
```

```
        }


        For (int j = 0; j < thread_count; j++)
        {
           pthread_join (tid[j], NULL);
        }
    }
    //when user give wrong input then this section will execute.
    else{
      printf ("You are entering wrong data.\n");
      main ();
    }

    return 0;
}
```

Test Cases:

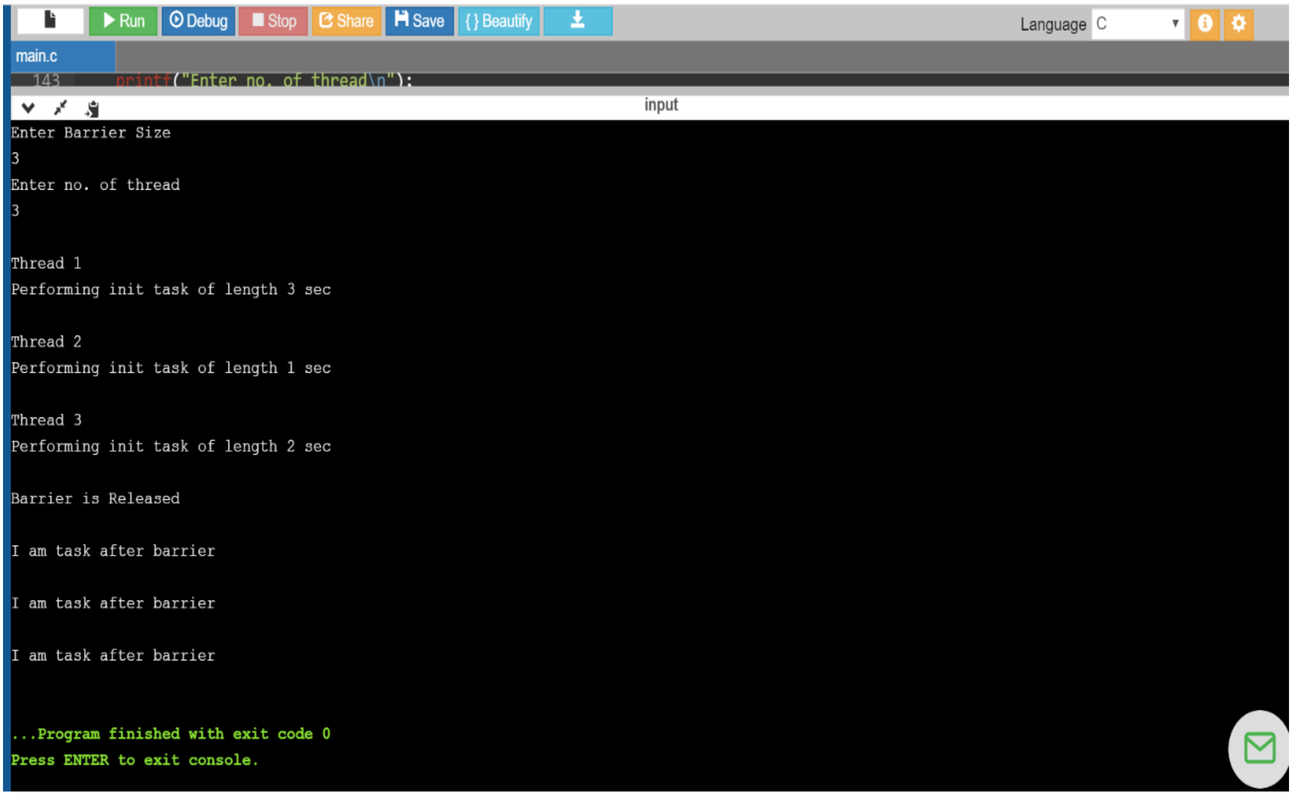
Case 1: when user enter invalid input like – string, double, float, negative no. etc.



```
printf("Enter no. of thread\n");
scanf("%d", &thread_count);

  //Checking valid input

if (barrier_size>=0 && thread_count>=0) {
    pthread_t tid[thread_count];

    barrier_init(barrier_size);

    for(int i =0; i < thread_count; i++)
    {
        pthread_create(&(tid[i]), NULL, &barrier_point, &thread_count);
```

```
Enter Barrier Size
-2
Enter no. of thread
-24
You are entering wrong data.
Enter Barrier Size
```

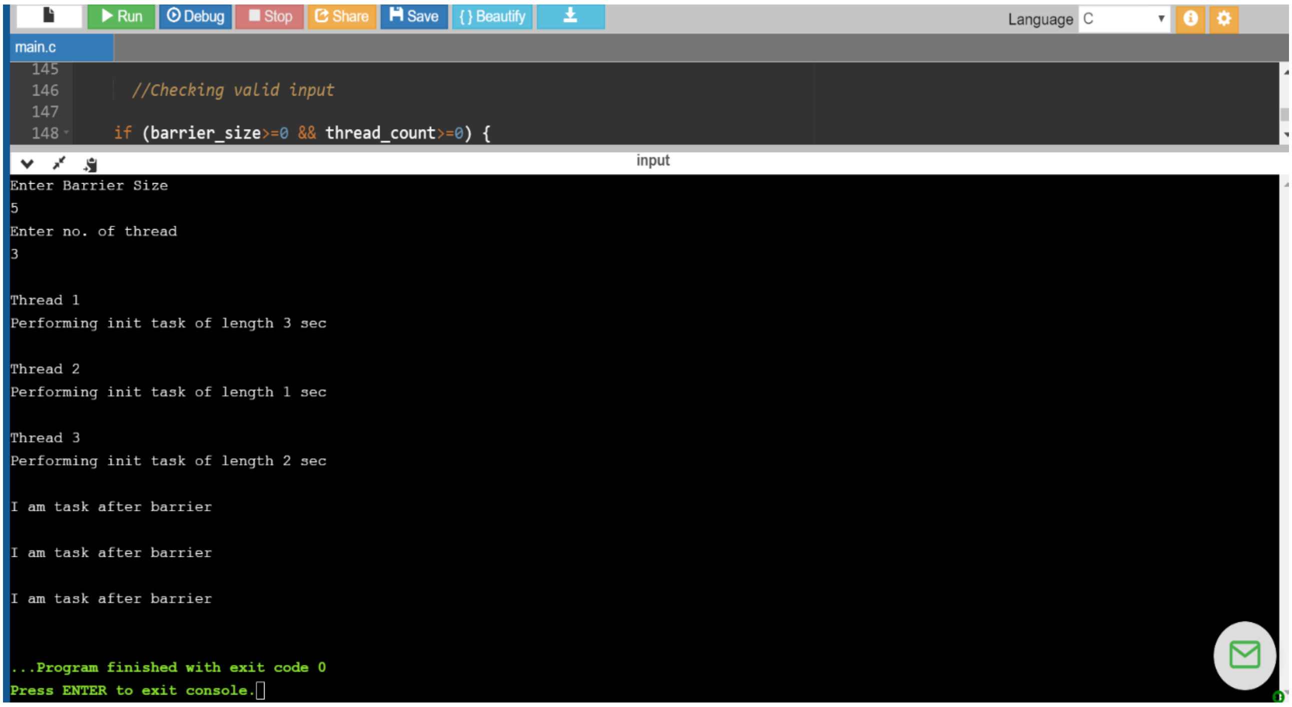
Case 2: when no. of thread equal to size of barrier.



```
    printf("Enter no. of thread\n");
```

```
Enter Barrier Size
3
Enter no. of thread
3

Thread 1
Performing init task of length 3 sec

Thread 2
Performing init task of length 1 sec

Thread 3
Performing init task of length 2 sec

Barrier is Released

I am task after barrier

I am task after barrier

I am task after barrier


...Program finished with exit code 0
Press ENTER to exit console.
```

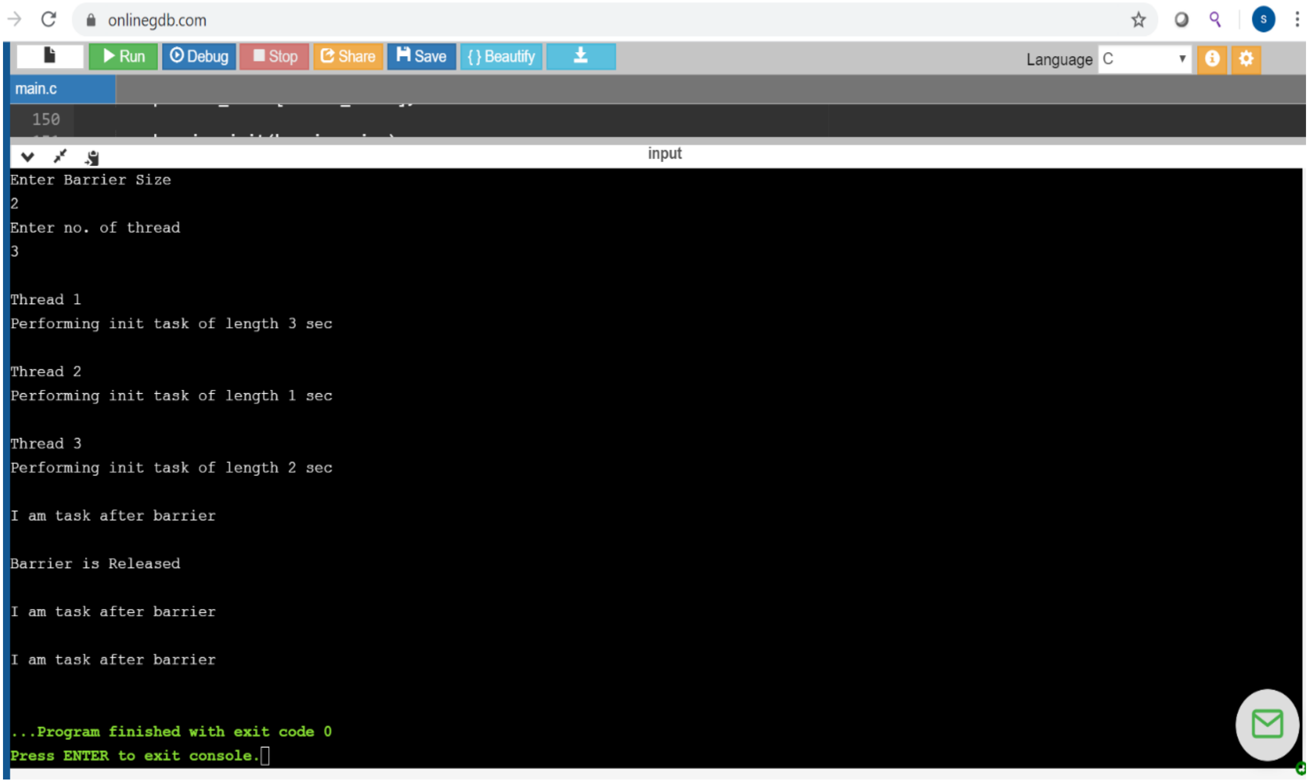
Case 3: when no. of thread is less than size of barrier .



```

      //Checking valid input

    if (barrier_size>=0 && thread_count>=0) {
```

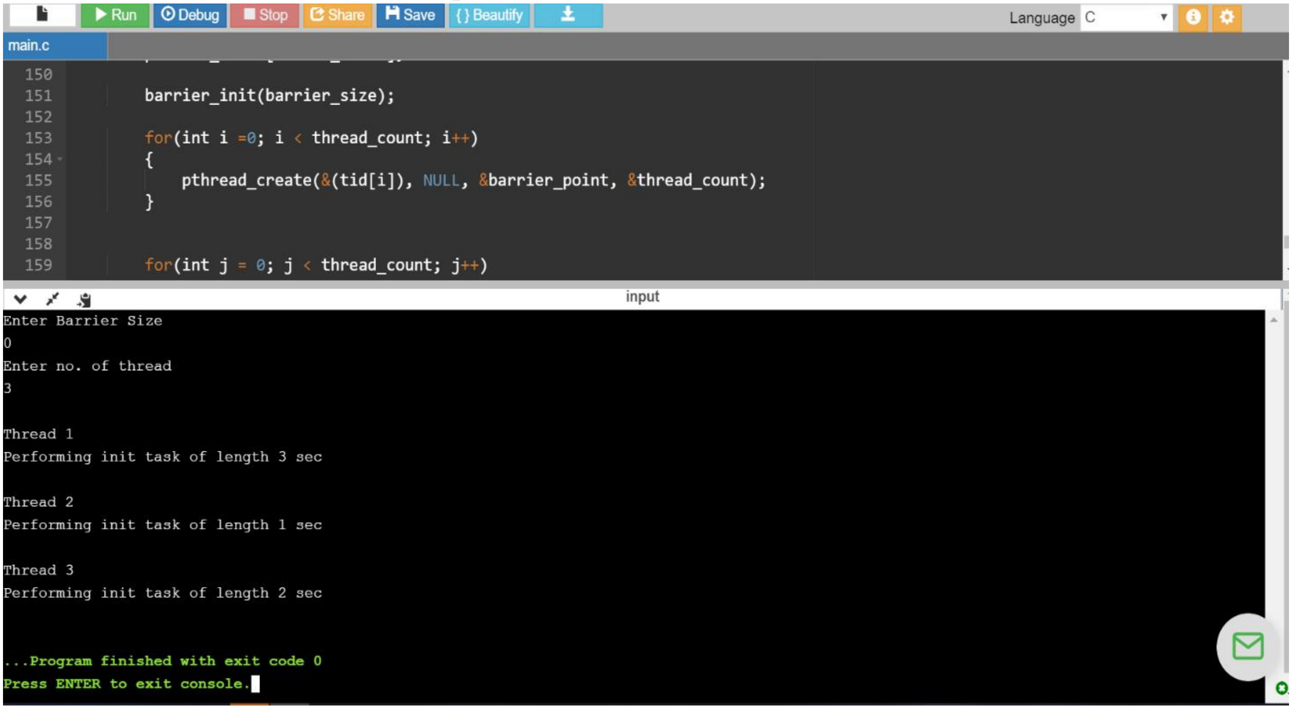
```
Enter Barrier Size
5
Enter no. of thread
3

Thread 1
Performing init task of length 3 sec

Thread 2
Performing init task of length 1 sec

Thread 3
Performing init task of length 2 sec

I am task after barrier

I am task after barrier

I am task after barrier


...Program finished with exit code 0
Press ENTER to exit console.
```

Case 4: when no. of thread is greater than size of Barrier.



```
Enter Barrier Size
2
Enter no. of thread
3

Thread 1
Performing init task of length 3 sec

Thread 2
Performing init task of length 1 sec

Thread 3
Performing init task of length 2 sec

I am task after barrier

Barrier is Released

I am task after barrier

I am task after barrier


...Program finished with exit code 0
Press ENTER to exit console.
```

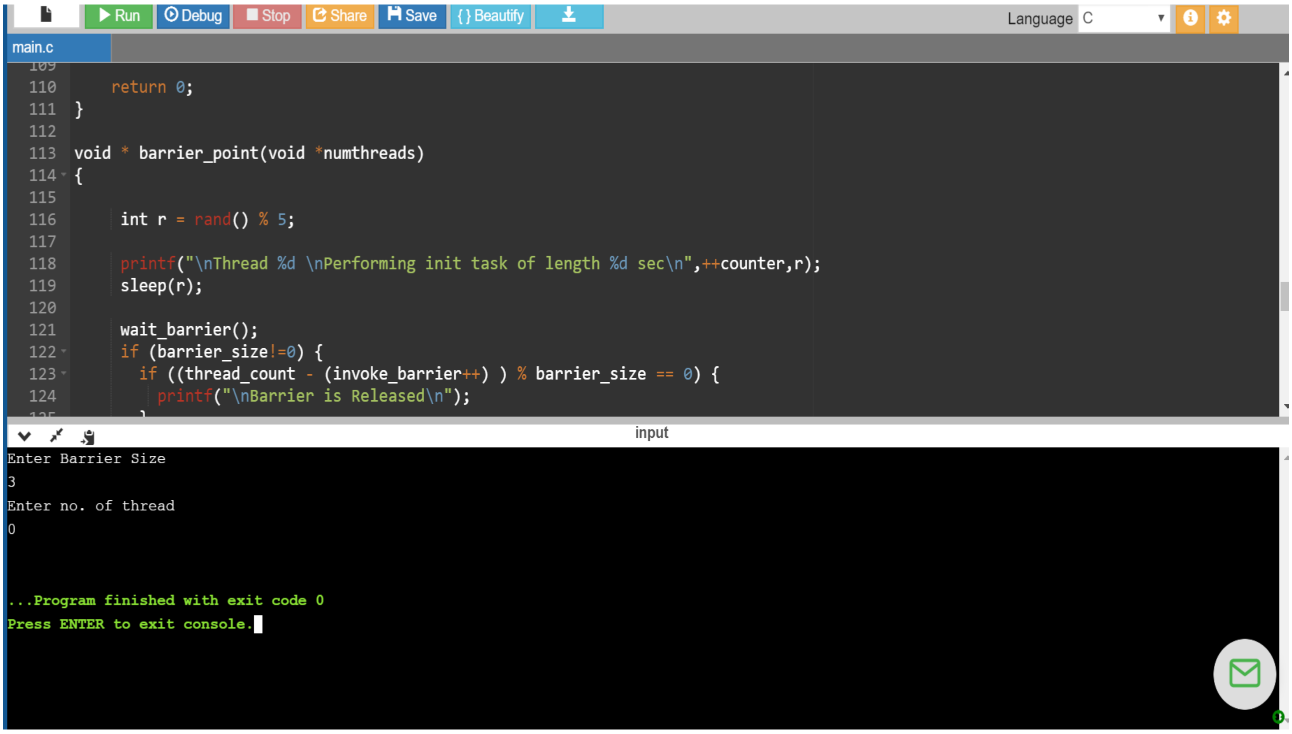
Case 5: when size of Barrier equal to ‘0’.



```
    barrier_init(barrier_size);

    for(int i =0; i < thread_count; i++)
    {
        pthread_create(&(tid[i]), NULL, &barrier_point, &thread_count);
    }


    for(int j = 0; j < thread_count; j++)
```

```
Enter Barrier Size
0
Enter no. of thread
3

Thread 1
Performing init task of length 3 sec

Thread 2
Performing init task of length 1 sec

Thread 3
Performing init task of length 2 sec


...Program finished with exit code 0
Press ENTER to exit console.
```

Case 6: when thread equal to ‘0’.



```
    return 0;
}

void * barrier_point(void *numthreads)
{

    int r = rand() % 5;

    printf("\nThread %d \nPerforming init task of length %d sec\n",++counter,r);
    sleep(r);

    wait_barrier();
    if (barrier_size!=0) {
      if ((thread_count - (invoke_barrier++) ) % barrier_size == 0) {
        printf("\nBarrier is Released\n");
```

```
Enter Barrier Size
3
Enter no. of thread
0


...Program finished with exit code 0
Press ENTER to exit console.
```