C Programming:

Language based capability:

- POSIX thread (pthreads)- Can be used in any OS platform. Can be used as user level or kernel level threads
- Windows thread (win32 threads). Used as kernel level threads for windows systems.

Thread creation:

Thread is a normal process and multithread is mostly used for concurrency. Threads are better than a process where the context switching and creation of thread is simple and easy.

Thread process creation is simple can be done using pthread_create function

Eg:

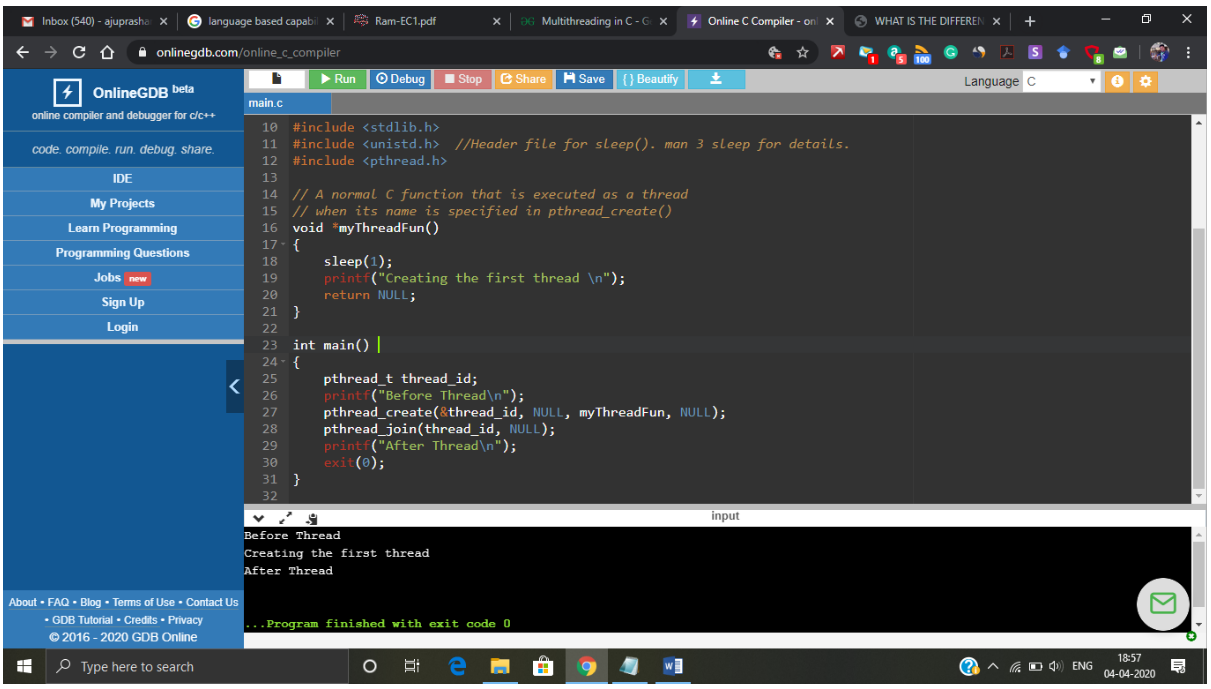
```
pthread_create(&thread_id,     // This contains the thread ID
                    NULL,             // attributes
                    myThreadFun,   //  function which contains the thread operation
                   NULL);             // argument of the function if any
```

Example for pthread creation process:

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.

#include <pthread.h>


// A normal C function that is executed as a thread

// when its name is specified in pthread_create()

void *myThreadFun()

{

   sleep(1);

   printf("Creating the first thread \n");

   return NULL;

}
```

```
int main()

{

   pthread_t thread_id;

   printf("Before Thread\n");

   pthread_create(&thread_id, NULL, myThreadFun, NULL);

   pthread_join(thread_id, NULL);   //  helps to wait for the thread termination

   printf("After Thread\n");

   exit(0);

}
```

If the pthread_join is removed the above thread won't be executed.



```
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun()
{
    sleep(1);
    printf("Creating the first thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

```
Before Thread
Creating the first thread
After Thread


...Program finished with exit code 0
```

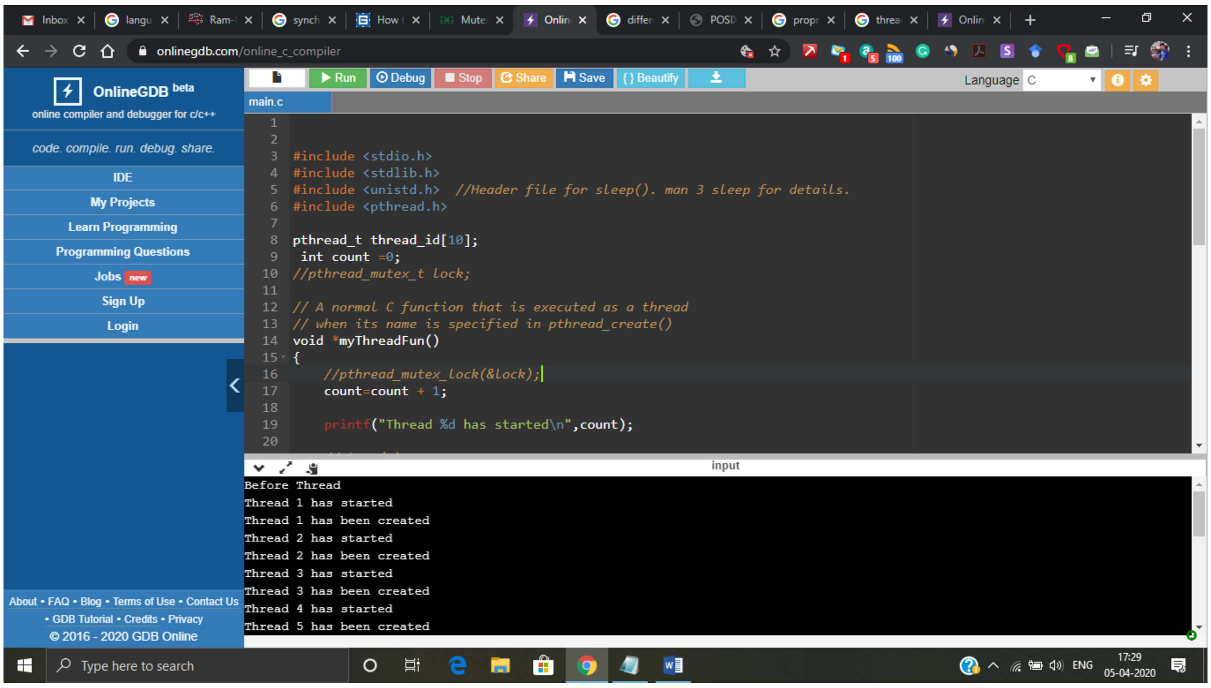
Figure 1: Output screenshot of thread creation

Thread creation is riskier but at the same time it is efficient too. If you take a look at the below code, there are 10 different threads created and executed.

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.

#include <pthread.h>


pthread_t thread_id[10];
```

```
 int count =0;

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun()
{
    count=count + 1;
    printf("Thread %d has started\n",count);
    printf("Thread %d has been created\n",count);
     return NULL;
}

int main()
{
    int i=0;
    printf("Before Thread\n");

    while(i < 10)
    {
    pthread_create(&thread_id[i], NULL, myThreadFun, NULL);
    i++;
    }

    pthread_join(thread_id[0], NULL);
    pthread_join(thread_id[1], NULL);
    pthread_join(thread_id[2], NULL);
    pthread_join(thread_id[3], NULL);
    pthread_join(thread_id[4], NULL);
    pthread_join(thread_id[5], NULL);
    pthread_join(thread_id[6], NULL);
    pthread_join(thread_id[7], NULL);
```

```
   pthread_join(thread_id[8], NULL);

   pthread_join(thread_id[9], NULL);

   printf("After Thread\n");

   return 0;

}
```



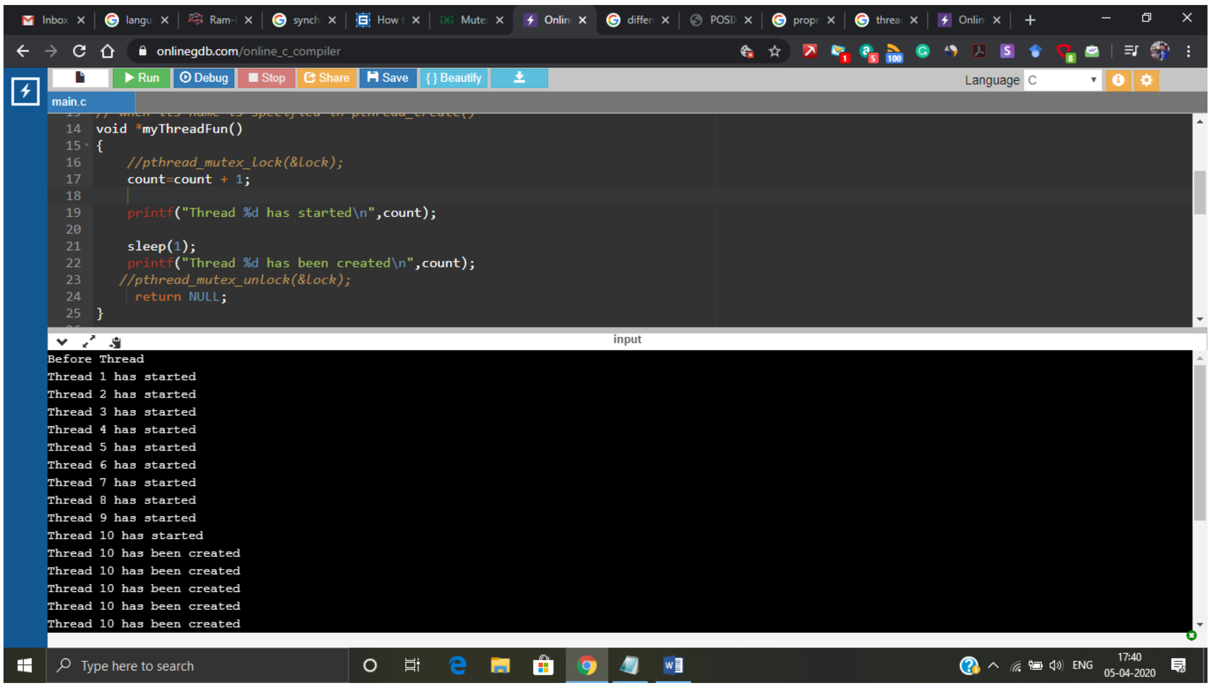
```


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

pthread_t thread_id[10];
 int count =0;
//pthread_mutex_t lock;

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun()
{
    //pthread_mutex_lock(&lock);
    count=count + 1;

    printf("Thread %d has started\n",count);

```

```
Before Thread
Thread 1 has started
Thread 1 has been created
Thread 2 has started
Thread 2 has been created
Thread 3 has started
Thread 3 has been created
Thread 4 has started
Thread 5 has been created
```

Figure 2: Multiple threads

The problem arises when we start creating multiple threads and start executing it. Just the small change in the code in the thread function can make a drastic change or trouble in the result. If I try to add one sleep function in between thread start and create printf statement we may arrive at a problem called **race condition.**

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.

#include <pthread.h>


pthread_t thread_id[10];

 int count =0;


// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun()
{
    count=count + 1;
    printf("Thread %d has started\n",count);
    sleep(2);                    // Change made is here
    printf("Thread %d has been created\n",count);
     return NULL;
}

int main()
{
    int i=0;
    printf("Before Thread\n");
    while(i < 10)
    {
    pthread_create(&thread_id[i], NULL, myThreadFun, NULL);
    i++;
    }

    pthread_join(thread_id[0], NULL);
    pthread_join(thread_id[1], NULL);
    pthread_join(thread_id[2], NULL);
    pthread_join(thread_id[3], NULL);
    pthread_join(thread_id[4], NULL);
    pthread_join(thread_id[5], NULL);
    pthread_join(thread_id[6], NULL);
    pthread_join(thread_id[7], NULL);
    pthread_join(thread_id[8], NULL);
    pthread_join(thread_id[9], NULL);
    printf("After Thread\n");

    return 0;

}
```



```
void *myThreadFun()
{
    //pthread_mutex_lock(&lock);
    count=count + 1;

    printf("Thread %d has started\n",count);

    sleep(1);
    printf("Thread %d has been created\n",count);
   //pthread_mutex_unlock(&lock);
     return NULL;
}
```

```
Before Thread
Thread 1 has started
Thread 2 has started
Thread 3 has started
Thread 4 has started
Thread 5 has started
Thread 6 has started
Thread 7 has started
Thread 8 has started
Thread 9 has started
Thread 10 has started
Thread 10 has been created
Thread 10 has been created
Thread 10 has been created
Thread 10 has been created
Thread 10 has been created
```

Figure 3: Race condition

If you can see in figure 3 just for single change of wait function all the threads have been started and confusion arises which thread is created because at last it tells thread 10 has been multiple number of times. So, here is where we need a synchronization mechanisms.

Synchronization mechanisms:

Why we need synchronization mechanism and what this can do?

When the system goes into race condition because of multiple threads accessing critical section the execution process may become tedious for the machine. So, to avoid this synchronization mechanism was introduced. The synchronization mechanism allows only one thread at a time, if one thread acquires the critical section it makes the other threads to wait till it completes the whole process.

There are different synchronization mechanisms in C multithreading namely,

- Mutex (Mutual Exclusion). – This is a lock and release mechanism.
- Semaphores – It follows a wait and release algorithm. Types are binary semaphore and counting semaphore.
- Condition variables

Types of Mutual exclusion techniques:

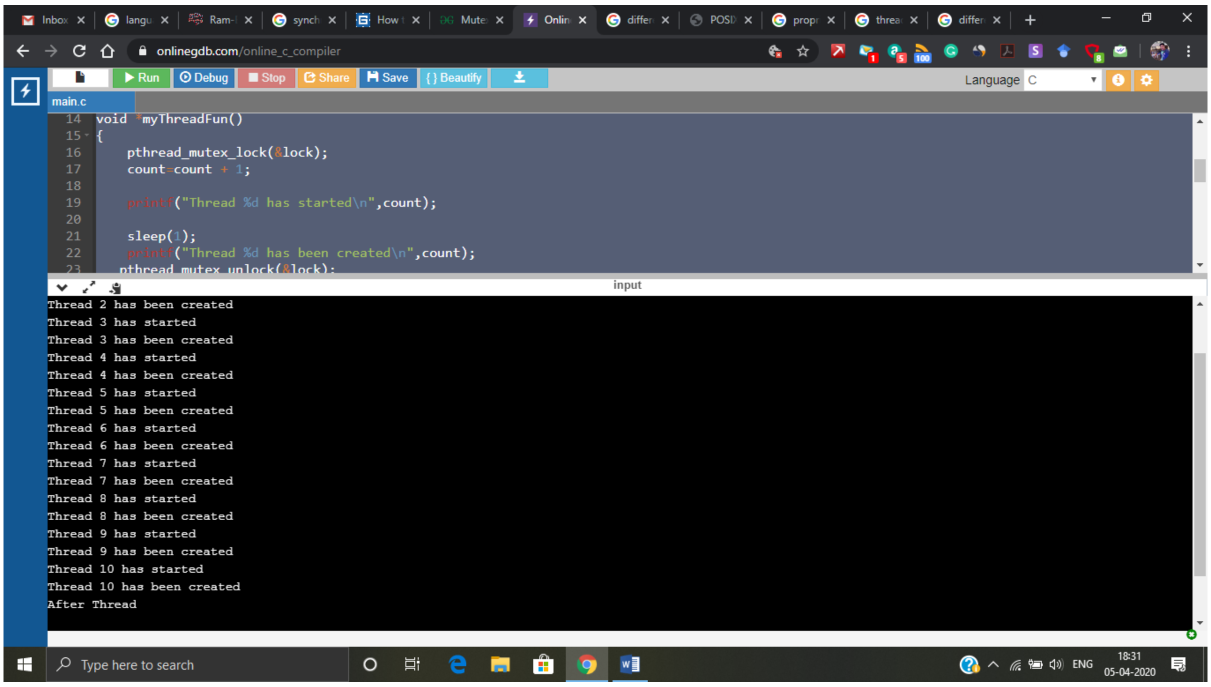
- **Recursive**: allows a thread holding the lock to acquire the same lock again which may be necessary for recursive algorithms.
- **Queuing**: allows for *fairness* in lock acquisition by providing FIFO ordering to the arrival of lock requests. Such mutexes may be slower due to increased overhead and the possibility of having to wake threads next in line that may be sleeping.
- **Reader/Writer**: allows for multiple readers to acquire the lock simultaneously. If existing readers have the lock, a writer request on the lock will block until all readers have given up the lock. This can lead to writer starvation.
- **Scoped**: RAII-style semantics regarding lock acquisition and unlocking.

So, the above race condition problem can be solved by **adding mutex synchronization** mechanism like this,

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.

#include <pthread.h>


pthread_t thread_id[10];

 int count =0;

pthread_mutex_t lock;   // creating a mutex synchronization


// A normal C function that is executed as a thread

// when its name is specified in pthread_create()

void *myThreadFun()

{

   pthread_mutex_lock(&lock);             // mutex lock function

   count=count + 1;


   printf("Thread %d has started\n",count);


   sleep(1);

   printf("Thread %d has been created\n",count);
  pthread_mutex_unlock(&lock);                  // mutex release function
    return NULL;
}

int main()
{
   int i=0;
   printf("Before Thread\n");
   pthread_mutex_init(&lock, NULL);         // mutex initialization

   while(i < 10)
   {
   pthread_create(&thread_id[i], NULL, myThreadFun, NULL);
   i++;
   }

   pthread_join(thread_id[0], NULL);
   pthread_join(thread_id[1], NULL);
   pthread_join(thread_id[2], NULL);
   pthread_join(thread_id[3], NULL);
   pthread_join(thread_id[4], NULL);
   pthread_join(thread_id[5], NULL);
   pthread_join(thread_id[6], NULL);
   pthread_join(thread_id[7], NULL);
   pthread_join(thread_id[8], NULL);
   pthread_join(thread_id[9], NULL);
  pthread_mutex_destroy(&lock);             // Destroying the mutex at the end
   printf("After Thread\n");
   return 0;
```

}

Before the thread function you can notice mutex lock taking place which means one thread acquires the lock and doesn't leave access to any other thread until it completes and at the end of the function mutex releases the lock of that particular thread and next thread will acquire the lock. In figure 4 you can notice that mutex synchronization makes the threads to execute in an orderly fashion.



```c
void *myThreadFun()
{
    pthread_mutex_lock(&lock);
    count=count + 1;

    printf("Thread %d has started\n",count);

    sleep(1);
    printf("Thread %d has been created\n",count);
  pthread_mutex_unlock(&lock);
```

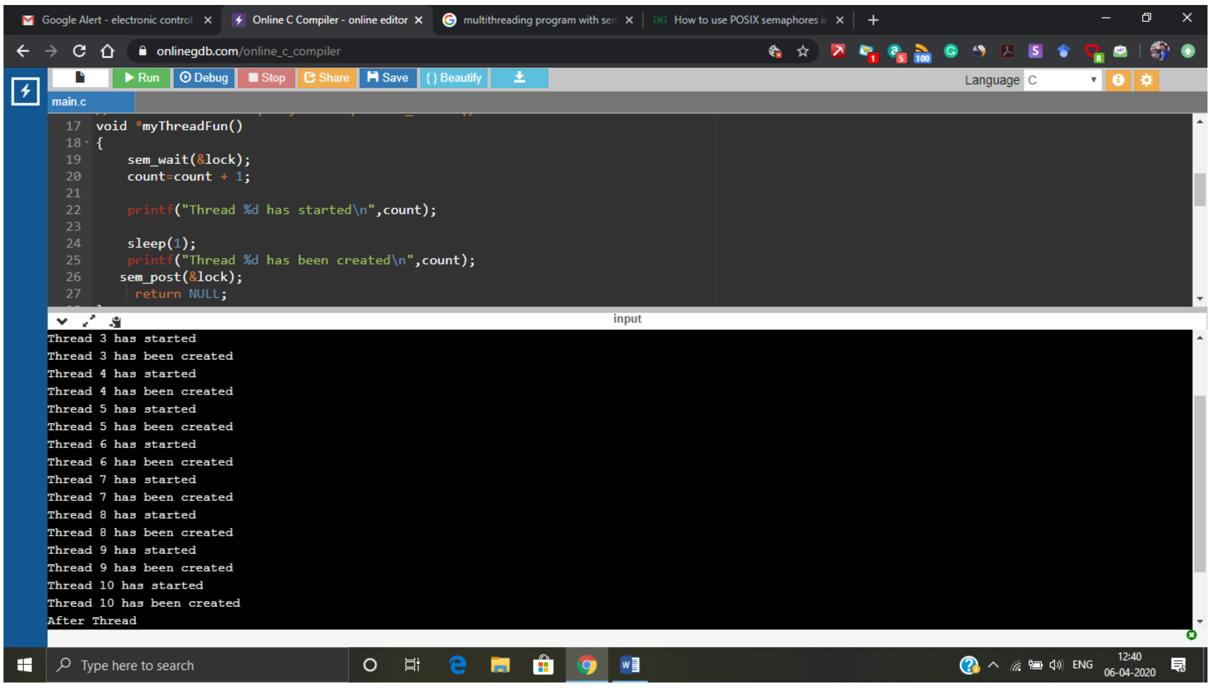
```
Thread 2 has been created
Thread 3 has started
Thread 3 has been created
Thread 4 has started
Thread 4 has been created
Thread 5 has started
Thread 5 has been created
Thread 6 has started
Thread 6 has been created
Thread 7 has started
Thread 7 has been created
Thread 8 has started
Thread 8 has been created
Thread 9 has started
Thread 9 has been created
Thread 10 has started
Thread 10 has been created
After Thread
```

Figure 4: Mutex synchronization

This race condition in the above figure 3 can be solved using semaphore synchronization mechanism too.

```c
#include <stdio.h>

#include <stdlib.h>

#include <semaphore.h>

#include <unistd.h>     //Header file for sleep(). man 3 sleep for details.

#include <pthread.h>


pthread_t thread_id[10];

 int count =0;

sem_t lock;
```

```
// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun()
{
    sem_wait(&lock);       // makes the other threads to wait
    count=count + 1;


    printf("Thread %d has started\n",count);


    sleep(1);
    printf("Thread %d has been created\n",count);
   sem_post(&lock);    // releases the thread that has executed and produced result.
     return NULL;
}


int main()
{
    int i=0;
    printf("Before Thread\n");
    sem_init(&lock, 0,1);
  /* initiating the semaphore and 1 represents only one thread can enter inside critical section at a
time */
    while(i < 10)
    {
    pthread_create(&thread_id[i], NULL, myThreadFun, NULL);
    i++;
    }


    pthread_join(thread_id[0], NULL);
    pthread_join(thread_id[1], NULL);
    pthread_join(thread_id[2], NULL);
    pthread_join(thread_id[3], NULL);
    pthread_join(thread_id[4], NULL);
    pthread_join(thread_id[5], NULL);
    pthread_join(thread_id[6], NULL);
    pthread_join(thread_id[7], NULL);
    pthread_join(thread_id[8], NULL);
    pthread_join(thread_id[9], NULL);
    sem_destroy(&lock);

    printf("After Thread\n");
    return 0;
}
```



Google Alert - electronic control
Online C Compiler - online editor
multithreading program with sem
How to use POSIX semaphores in
onlinegdb.com/online_c_compiler
Run
Debug
Stop
Share
Save
{ } Beautify
Language C
main.c
input
Type here to search
ENG
12:40
06-04-2020

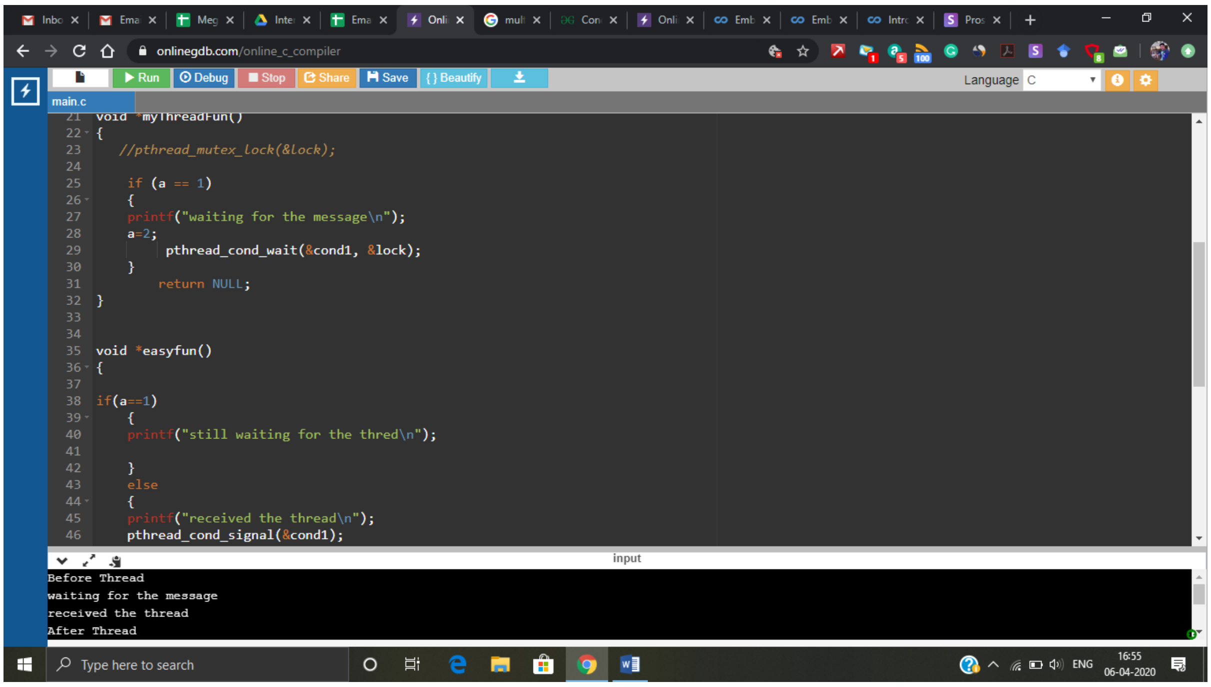
```
void *myThreadFun()
{
    sem_wait(&lock);
    count=count + 1;

    printf("Thread %d has started\n",count);

    sleep(1);
    printf("Thread %d has been created\n",count);
   sem_post(&lock);
    return NULL;
```

```
Thread 3 has started
Thread 3 has been created
Thread 4 has started
Thread 4 has been created
Thread 5 has started
Thread 5 has been created
Thread 6 has started
Thread 6 has been created
Thread 7 has started
Thread 7 has been created
Thread 8 has started
Thread 8 has been created
Thread 9 has started
Thread 9 has been created
Thread 10 has started
Thread 10 has been created
After Thread
```

Figure 5: Semaphore mechanism

Figure 5 shows that the race condition has been overcome using semaphore mechanism. The race condition which was showcased above can't be done using conditional variable synchronization mechanism the best suited one would be semaphore or mutex. But conditional variable mechanism are used when we need to play around using conditions.

```
#include <stdio.h>

#include <stdlib.h>

#include <semaphore.h>

#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.

#include <pthread.h>


pthread_t thread_id;

pthread_t easy_id;

 int count =0;

// Declaration of thread condition variable

pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;

// declaring mutex

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

 int a =1;



// A normal C function that is executed as a thread

// when its name is specified in pthread_create()

void *myThreadFun()

{

   //pthread_mutex_lock(&lock);


    if (a == 1)

    {

    printf("waiting for the message\n");

    a=2;

         pthread_cond_wait(&cond1, &lock);

    }

              return NULL;
}



void *easyfun()
{


if(a==1)
   {
   printf("still waiting for the thred\n");
   }
   else
   {
   printf("received the thread\n");
   pthread_cond_signal(&cond1);
}
          //pthread_mutex_unlock(&lock);
          return NULL;
}

int main()
{
   printf("Before Thread\n");
   pthread_create(&thread_id, NULL, myThreadFun, NULL);
   sleep(1);
   pthread_create(&easy_id, NULL, easyfun, NULL);
   pthread_join(thread_id, NULL);
   pthread_join(easy_id, NULL);
   printf("After Thread\n");
   return 0;
}
```



```c
void *myThreadFun()
{
   //pthread_mutex_lock(&lock);

    if (a == 1)
    {
    printf("waiting for the message\n");
    a=2;
         pthread_cond_wait(&cond1, &lock);
    }
        return NULL;
}


void *easyfun()
{

if(a==1)
    {
    printf("still waiting for the thred\n");

    }
    else
    {
    printf("received the thread\n");
    pthread_cond_signal(&cond1);
```

```
Before Thread
waiting for the message
received the thread
After Thread
```

Figure 6: conditional variable mechanism

Conditional variable is kind of wait and receive mechanism.

Java Programming:

Language based capability:

- Java threads were implemented with non-native threads called Green threads. Green threads can run on multicore processor but cannot take any advantage of multiple cores. Green threads are application level only.
- Linux native threads (POSIX thread) have better input- output and synchronization operation than green threads. Native thread model uses underlying OS support.
- ALL Java programs use Threads - even "common" single-threaded ones.
- The creation of new Threads requires Objects that implement the Runnable Interface, which means they contain a method "public void run( )" . Any descendant of the Thread class will naturally contain such a method. (In practice the run( ) method must be overridden / provided for the thread to have any practical functionality.)
- Creating a Thread Object does not start the thread running - To do that the program must call the Thread's "start( )" method. Start( ) allocates and initializes memory for the Thread, and then calls the run( ) method. (Programmers do not call run( ) directly.)
- Because Java does not support global variables, Threads must be passed a reference to a shared Object in order to share data, in this example the "Sum" Object.
- Note that the JVM runs on top of a native OS, and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision is JVM implementation dependant, and may be one-to-one, many-to-many, or many to one. (On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads)

Thread creation:

JAVA and C has a lot of syntax oriented difference and the thread creation process also differs so the threads can be created in two different ways

- Implement runnable
  Eg:- Class thread implement runnable
- Extends thread
  Eg:- Class Mythread extends Thread

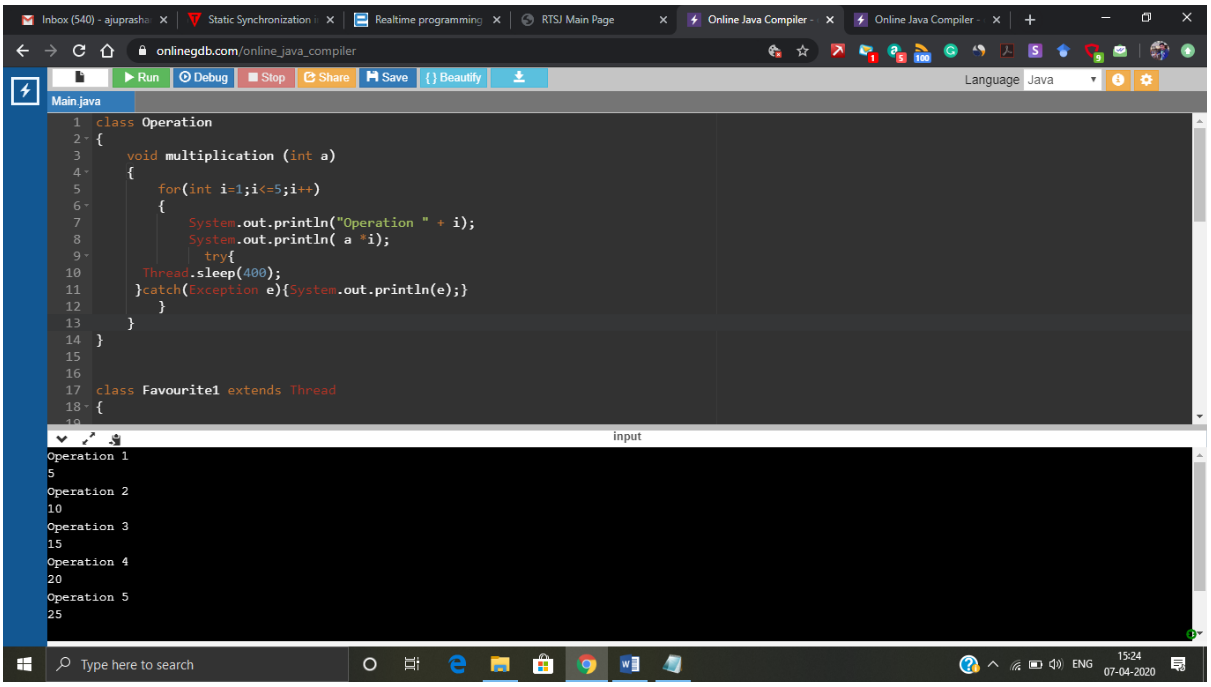
These two ways can be followed and given in the class for thread creation.

Thread creation is a simple process which can be seen in the below program,

```
class Operation

{

  void multiplication (int a)
```

```
   {
       for(int i=1;i<=5;i++)
       {
           System.out.println("Operation " + i);
           System.out.println( a *i);
             try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
       }
   }
}

class Favourite1 extends Thread
{

    Operation t;
Favourite1(Operation t){
this.t=t;
}
  public void run()
   {

   t.multiplication(5);
   }
}

public class Main              /* Main function*/
{
      public static void main(String[] args)
```

```
        {
        Operation obj = new Operation();//only one object
                Favourite1 t1= new Favourite1(obj);
                t1.start();
        }
    }
```



```
class Operation
{
    void multiplication (int a)
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("Operation " + i);
            System.out.println( a *i);
              try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
        }
    }
}


class Favourite1 extends Thread
{
```

```
Operation 1
5
Operation 2
10
Operation 3
15
Operation 4
20
Operation 5
25
```

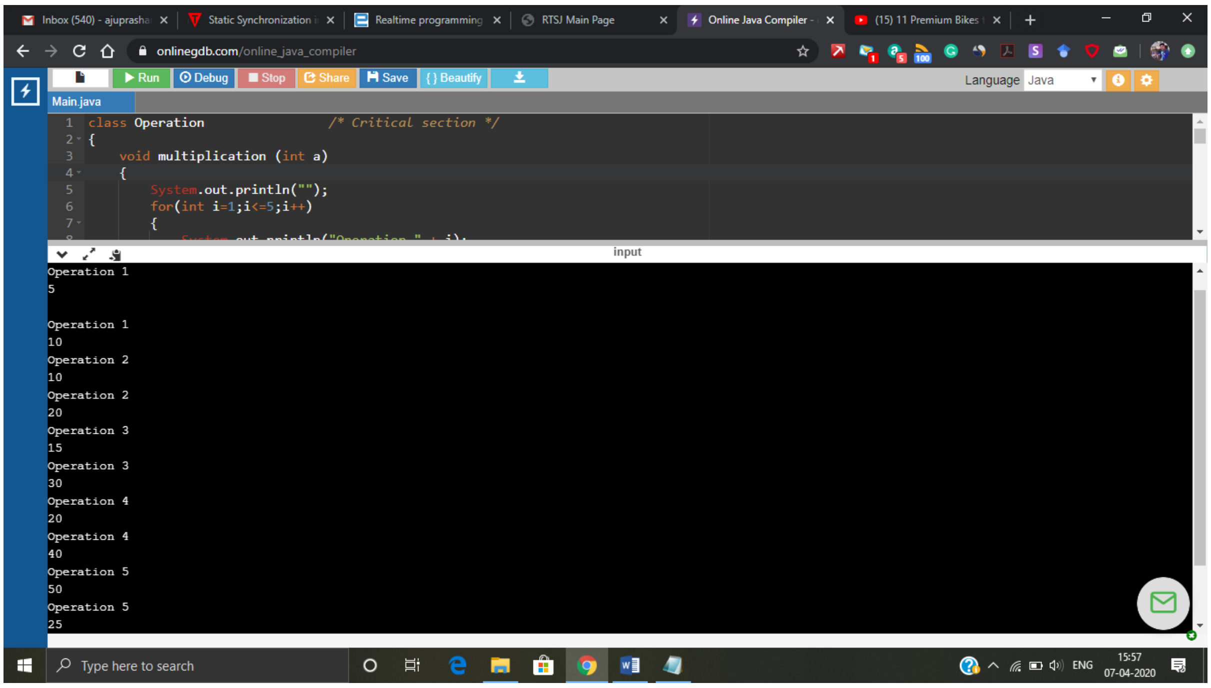
Figure 7: Thread creation

In the above figure we can see that thread creation and execution is a pretty simple process but as we saw before in C programming where we created multiple threads and it became hard to understand the result. Now we will experiment what happens to the result when we create more than one thread and both the threads are trying to actuate the critical section.

```
class Operation             /* Critical section */
{
  void multiplication (int a)
  {
    System.out.println("");
    for(int i=1;i<=5;i++)
    {
      System.out.println("Operation " + i);
      System.out.println( a *i);
```

```
            try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
       }
    }
}


class Favourite1 extends Thread         /* thread 1 */
{
     Operation t;
Favourite1(Operation t){
this.t=t;
}

  public void run()
    {
    t.multiplication(5);
    }
}

class Favourite2 extends Thread    /* Thread 2*/
{
    Operation t;
Favourite2(Operation t){
this.t=t;
}
   public void run()
    {
    t.multiplication(10);
    }
```

```
}


public class Main
{
        public static void main(String[] args)
        {
        Operation obj = new Operation();//only one object
                Favourite1 t1= new Favourite1(obj);
                Favourite2 t2= new Favourite2(obj);
                t1.start();
                t2.start();
        }
}
```



```
class Operation                   /* Critical section */
{
    void multiplication (int a)
    {
        System.out.println("");
        for(int i=1;i<=5;i++)
        {
```

```
Operation 1
5

Operation 1
10
Operation 2
10
Operation 2
20
Operation 3
15
Operation 3
30
Operation 4
20
Operation 4
40
Operation 5
50
Operation 5
25
```

Figure 8: Multiple threads output

The figure 8 shows the output of the program which is written above but nothing can be understood since both the threads are trying to handle the critical section. So, this can be solved using some synchronization mechanism which allows one thread at a time to access the critical section.

Synchronization mechanisms:

- Synchronization method
- Synchronization block

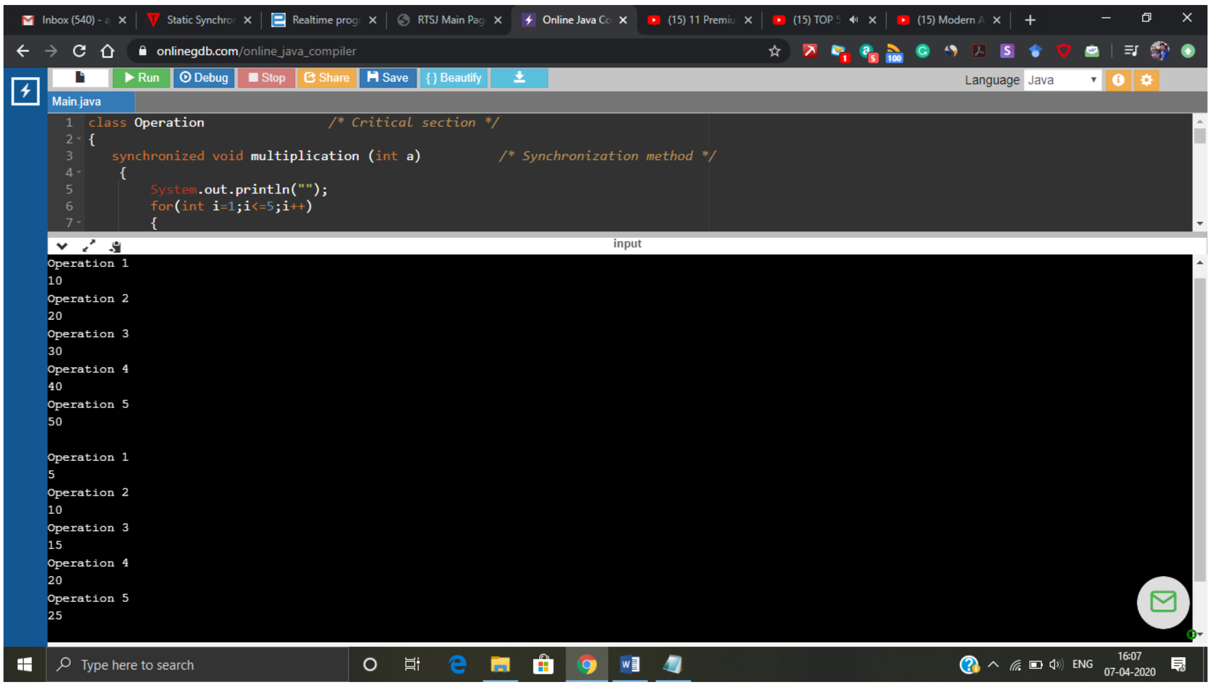
- Static synchronization
- Inter thread synchronization- In this you have wait(), notify() and notifyall().

The above problem of figure 8 problem can be solved by adding “**synchronization method”,** just by adding the word synchronized before the critical section function will introduce synchronization to the program.

```
class Operation              /* Critical section */
{
  synchronized void multiplication (int a)        /* Synchronization method */
   {
      System.out.println("");
      for(int i=1;i<=5;i++)
      {
         System.out.println("Operation " + i);
         System.out.println( a *i);
           try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
      }
   }
}

class Favourite1 extends Thread           /* Thread1 */
{
    Operation t;
Favourite1(Operation t){
this.t=t;
}
  public void run()
```

```
   {
   t.multiplication(5);
   }
}

class Favourite2 extends Thread              /* Thread 2   */
{
   Operation t;
Favourite2(Operation t){
this.t=t;
}
  public void run()
   {
   t.multiplication(10);
   }
}

public class Main                                /* Main block*/
{
        public static void main(String[] args)
        {
        Operation obj = new Operation();//only one object
                Favourite1 t1= new Favourite1(obj);
                Favourite2 t2= new Favourite2(obj);
                t1.start();
                t2.start();
        }
}
```

Figure 9 shows the output after the synchronization method has been added to the program. Which has make the output in a well organised way.



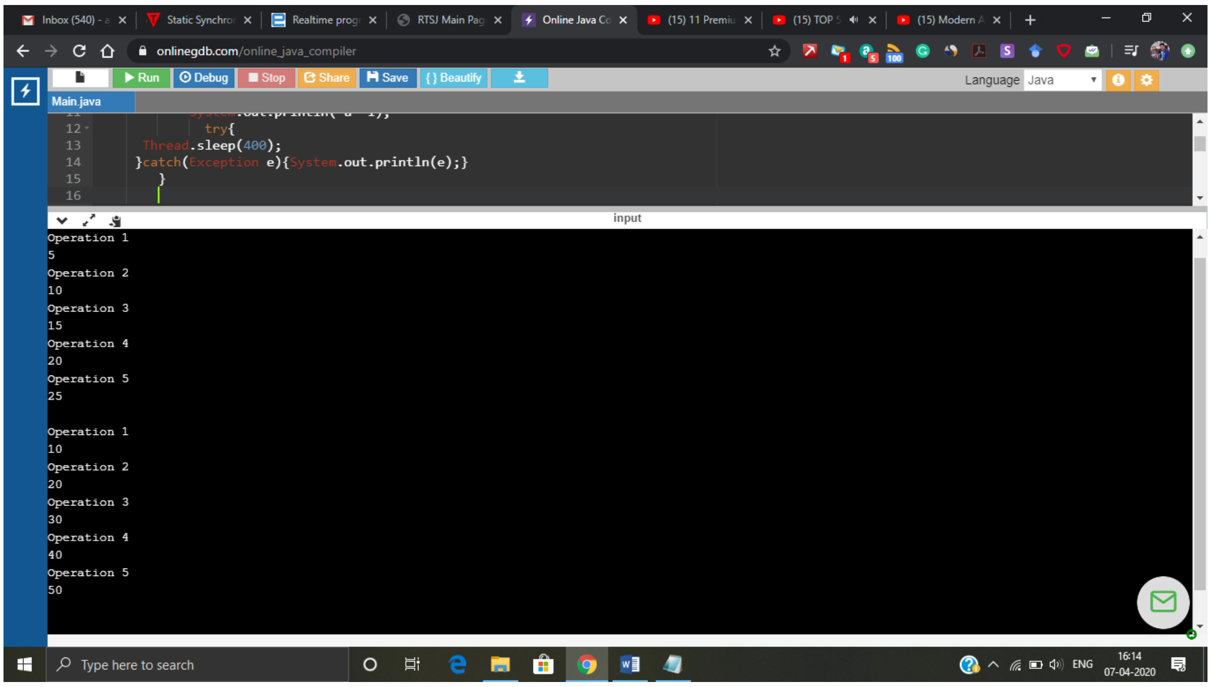
```
class Operation                /* Critical section */
{
   synchronized void multiplication (int a)          /* Synchronization method */
    {
        System.out.println("");
        for(int i=1;i<=5;i++)
        {
```

```
Operation 1
10
Operation 2
20
Operation 3
30
Operation 4
40
Operation 5
50

Operation 1
5
Operation 2
10
Operation 3
15
Operation 4
20
Operation 5
25
```

Figure 9: Synchronization method

The above problem of figure 8 problem can be solved by adding “**synchronization block”,** Just a single change was made in the program synchronized (this) was added

```
class Operation              /* Critical section */

{

  void multiplication (int a)

   {

       synchronized (this)              /* Synchronization method */

      {

      System.out.println("");

      for(int i=1;i<=5;i++)

      {

         System.out.println("Operation " + i);

         System.out.println( a *i);

           try{

     Thread.sleep(400);

    }catch(Exception e){System.out.println(e);}

      }

      }
    }
}


class Favourite1 extends Thread            /* Thread1 */
{
     Operation t;
Favourite1(Operation t){
this.t=t;
}
  public void run()
    {
    t.multiplication(5);
    }
}

class Favourite2 extends Thread              /* Thread 2   */
{
    Operation t;
Favourite2(Operation t){
this.t=t;
}
   public void run()
    {
    t.multiplication(10);
    }
}
public class Main                                /* Main block*/
{
        public static void main(String[] args)

        {

        Operation obj = new Operation();//only one object

                Favourite1 t1= new Favourite1(obj);

                Favourite2 t2= new Favourite2(obj);

                t1.start();

                t2.start();

        }

}
```



```
            try{
 Thread.sleep(400);
}catch(Exception e){System.out.println(e);}
   }
```

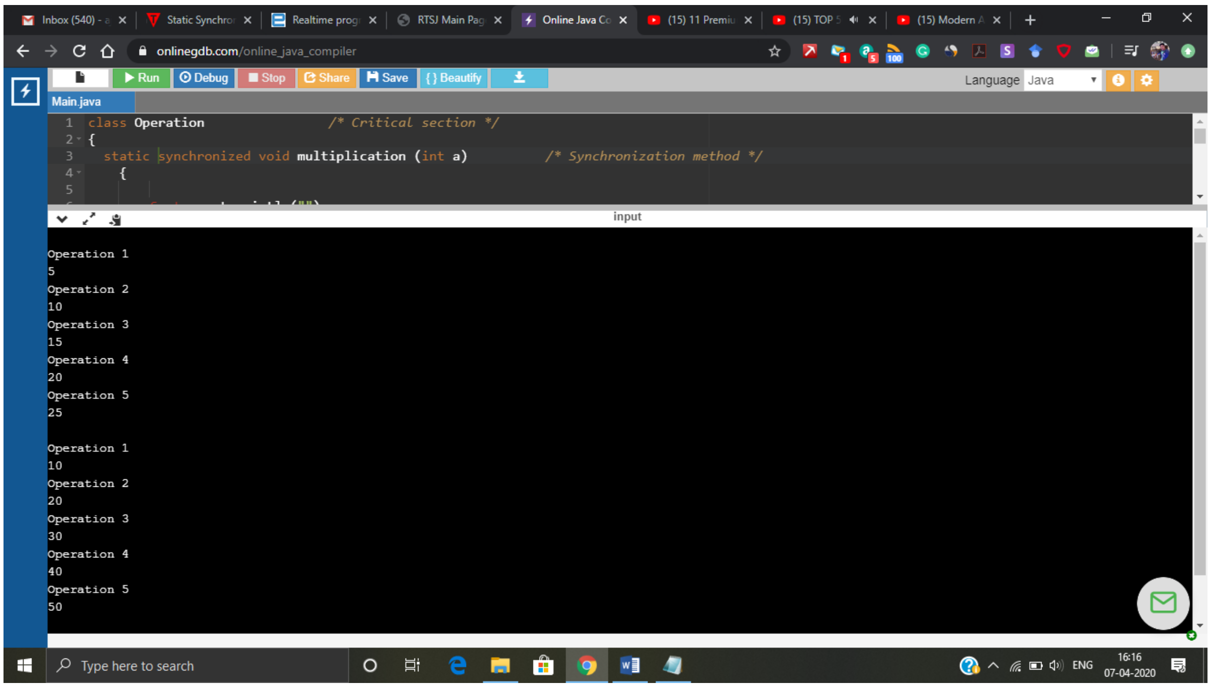
```
Operation 1
5
Operation 2
10
Operation 3
15
Operation 4
20
Operation 5
25

Operation 1
10
Operation 2
20
Operation 3
30
Operation 4
40
Operation 5
50
```

Figure 10: Synchronization block

The above problem of figure 8 problem can be solved by adding "**synchronization static",** Just a single change was made in the program static synchronization was added.

```
class Operation            /* Critical section */

{

 static synchronized void multiplication (int a)       /* Synchronization method */

  {

     System.out.println("");

     for(int i=1;i<=5;i++)

     {
            System.out.println("Operation " + i);

            System.out.println( a *i);

              try{

      Thread.sleep(400);

     }catch(Exception e){System.out.println(e);}

        }

    }

}


class Favourite1 extends Thread             /* Thread1 */

{

     Operation t;

Favourite1(Operation t){

this.t=t;

}

  public void run()

    {

    t.multiplication(5);

    }

}

class Favourite2 extends Thread              /* Thread 2   */

{

    Operation t;


Favourite2(Operation t){

this.t=t;

}

   public void run()

    {

    t.multiplication(10);

    }
```

```
}

public class Main                         /* Main block*/
{
        public static void main(String[] args)
        {
        Operation obj = new Operation();//only one object
                Favourite1 t1= new Favourite1(obj);
                Favourite2 t2= new Favourite2(obj);
                t1.start();
                t2.start();
        }
}
```



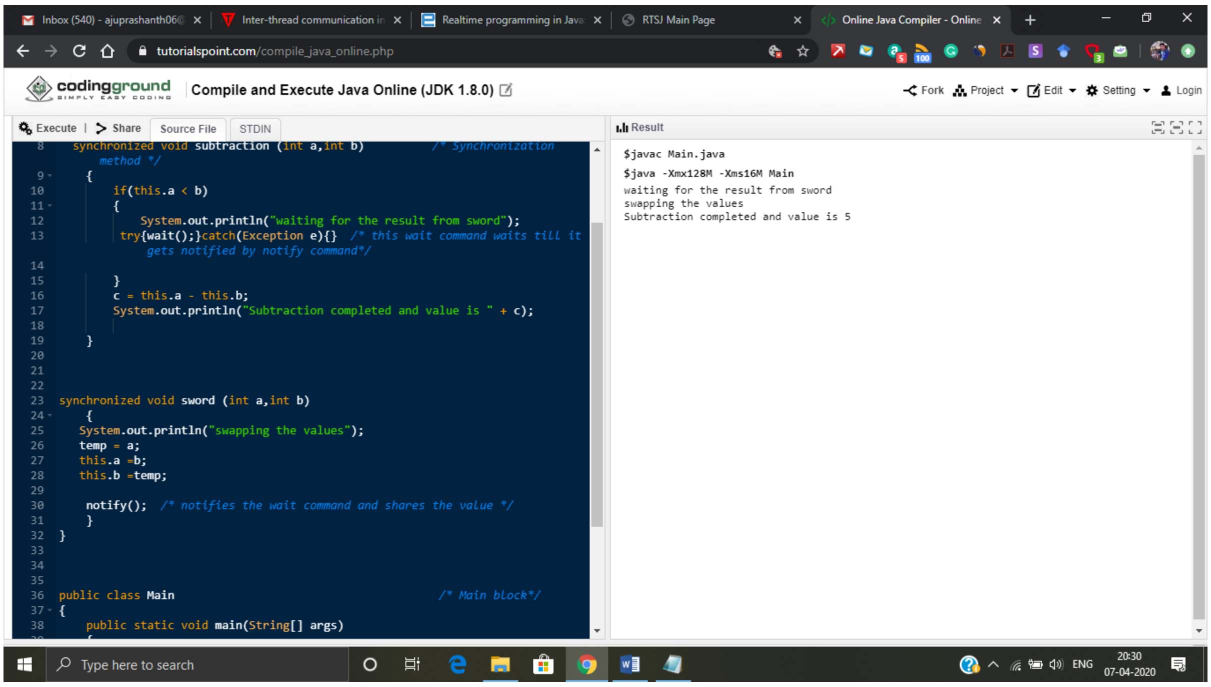
```
class Operation                  /* Critical section */
{
  static synchronized void multiplication (int a)          /* Synchronization method */
    {
```

```
Operation 1
5
Operation 2
10
Operation 3
15
Operation 4
20
Operation 5
25

Operation 1
10
Operation 2
20
Operation 3
30
Operation 4
40
Operation 5
50
```

Figure 11: Static synchronization.

We can't perform an **Inter thread synchronization** for the above problem**.** It's mostly based on the conditional variable synchronization which we saw in multithread C programming. Inter thread synchronization can be used for different functions inside a same class. So, the example has to be slightly changed for this.

```
class Operation             /* Critical section */
{
   int a=5;

   int b=10;

   int c=0;

   int temp=0;


  synchronized void subtraction (int a,int b)         /* Synchronization method */

   {

       if(this.a < b)

       {

          System.out.println("waiting for the result from sword");

        try{wait();}catch(Exception e){}  /* this wait command waits till it gets notified by notify
command*/

       }

       c = this.a - this.b;

       System.out.println("Subtraction completed and value is " + c);

   }


synchronized void sword (int a,int b)

   {

  System.out.println("swapping the values");

  temp = a;

  this.a =b;

  this.b =temp;

   notify();  /* notifies the wait command and shares the value */

   }

}


public class Main                                    /* Main block*/

{

            public static void main(String[] args)
        {
        final Operation c=new Operation();
new Thread(){
public void run(){c.subtraction(5,10);}
}.start();
new Thread(){
public void run(){c.sword(5,10);}
}.start();
        }
}
```



tutorialspoint.com/compile_java_online.php
codingground
Compile and Execute Java Online (JDK 1.8.0)
Execute
Share
Source File
STDIN
Result

```
  synchronized void subtraction (int a,int b)          /* Synchronization
    method */
     {
         if(this.a < b)
         {
             System.out.println("waiting for the result from sword");
          try{wait();}catch(Exception e){}  /* this wait command waits till it
              gets notified by notify command*/

         }
         c = this.a - this.b;
         System.out.println("Subtraction completed and value is " + c);

     }



synchronized void sword (int a,int b)
     {
   System.out.println("swapping the values");
   temp = a;
   this.a =b;
   this.b =temp;

    notify();  /* notifies the wait command and shares the value */
     }
}



public class Main                                      /* Main block*/
{
    public static void main(String[] args)
```

```
$javac Main.java
$java -Xmx128M -Xms16M Main
waiting for the result from sword
swapping the values
Subtraction completed and value is 5
```

Figure 12: Inter thread synchronization

Real time application development for both C and Java:

support by the programming language examples: Ada, Java advantages: readability, OS independence, checking of interactions by compiler

support by libraries and the operating system examples: C/C++ with POSIX advantages: multi-language composition, possibly more efficient, OS standards.

Real time systems are to be programmed in JAVA using RTSJ. In RTSJ the threads and scheduler will also communicate with garbage collector. Even if we design a Real time system using Java there are no features of letting us know when will the task be completed because there are no priority inversion algorithm. In short, there are not guarantees about execution order in standard Java.

Even if Java is the best language for developers of commercial applications and Web applets, it may still not be appropriate as an embedded language. The key issue in real-time systems is deterministic behaviour. Unfortunately, the current generation of garbage collectors is inherently non-deterministic. Moreover, garbage collection is an integral part of the Java language. Any variables that are not primitive types are objects. Because garbage collection cannot be eliminated from the language, several groups are working to create deterministic garbage collectors. Developers of real-time systems will not want to use Java until such alternatives become available. Java is not the most efficient language either. For now, embedded developers wanting to use Java must make do with a Java VM. Unfortunately, that means slow execution-sometimes less than 10% as fast as a similar program written in C.

WHAT IS THE DIFFERENCE BETWEEN C AND JAVA?

- C is structure/procedure oriented programming language whereas Java is object oriented programming language.
- C language program design is top down approach whereas Java is using bottom up approach.
- C language is middle level language whereas Java is high level language.
- Exception handling is not present in C programming language. Whereas exception handling is present in Java.
- Polymorphism, virtual function, inheritance, Operator overloading, namespace concepts are not available in C programming language. Whereas Java supports all these concepts and features.
- Java is a platform independent language where C isn't.

References:

- https://www.geeksforgeeks.org/multithreading-c-2/
- https://fresh2refresh.com/c-programming/c-interview-questions-answers/what-is-the-difference-between-c-and-java/
- https://randu.org/tutorials/threads/
- https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html
- https://computing.llnl.gov/tutorials/pthreads/#Thread
- https://www.geeksforgeeks.org/use-posix-semaphores-c/
- https://www.geeksforgeeks.org/condition-wait-signal-multi-threading/
- https://www.fi.muni.cz/~xpelanek/IA158/slides/programming-concepts.pdf
- https://www.javatpoint.com/synchronization-in-java
- https://www.embedded.com/realtime-programming-in-java-part-1/
- https://barrgroup.com/embedded-systems/how-to/embedded-java