# **C Programming:**

**Language based capability:**

**User Level threads:** User manages this thread in Application layer. Kernel thread management is not aware of existence of this thread.

* Thread switching does not require Kernel mode privileges.
* User level thread can run on any operating system.
* Scheduling can be application specific in the user level thread.
* User level threads are fast to create and manage.
* In a typical operating system, most system calls are blocking.
* Multithreaded application cannot take advantage of multiprocessing.

**Kernel level threads:** Operating system manages this thread in kernel level. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

* Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
* If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
* Kernel routines themselves can be multithreaded.
* Kernel threads are generally slower to create and manage than the user threads.
* Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.
* 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.

**Reason to choose pthreads over win32 threads:**

* **Separate data types.** In Pthreads, each object has its own data type while in Win32 threads there is a mix of handles and separate types. For Pthreads this means different functions are used for working with each object type.
* **Unambiguous functionality:** Pthreads has a single function to create threads. If you include the C Runtime Library, there are three separate ways to do this for Win32 threads.
* **Persistence of signals:** If there are any errors in proper switching of Win32 even with debugging tools it’s difficult to track the error. Under Pthreads, signals to condition variables are either "caught" by waiting thread(s) or discarded. However, use of a well known coding structure at each access of a condition variable will ensure no signals are "lost" by threads that may not be waiting at the exact time of signaling.
* Pthreads are straightforward and easy to understand when compared to win32 threads.
* Synchronization objects for win32 threads: events, semaphore, mutex, critical sections.
* Synchronization primitives for pthreads: mutex, semaphore, conditional variable.

**How are threads and concurrency related:**

Thread is created by OS to run a stream of instruction. A thread contains stack, ID, heap, scheduling mechanism. Threads are nothing but a task which takes place inside a process. Concurrency is having two tasks run in parallel on separate threads

**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. The pthread\_join() function provides a simple mechanism allowing an application to wait for a thread to terminate. After the thread terminates, the application may then choose to clean up resources that were used by the thread. For instance, after pthread\_join() returns, any application-provided stack storage could be reclaimed. The pthread\_join() function should eventually be called for every thread that is created with the detachstate attribute set to PTHREAD\_CREATE\_JOINABLE so that storage associated with the thread may be reclaimed. Pthread\_join is basically like a wait function for the pthreads.

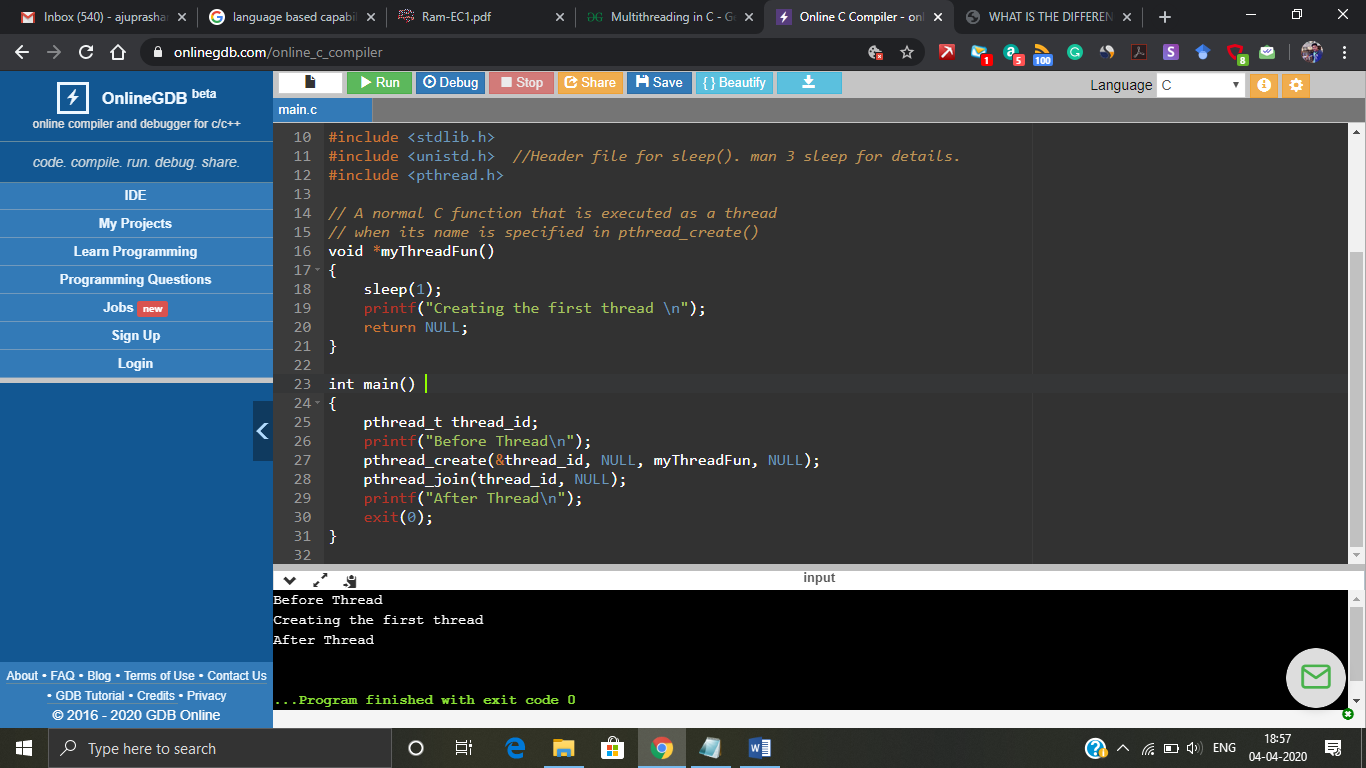


Figure 1: Output screenshot of thread creation

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;

}

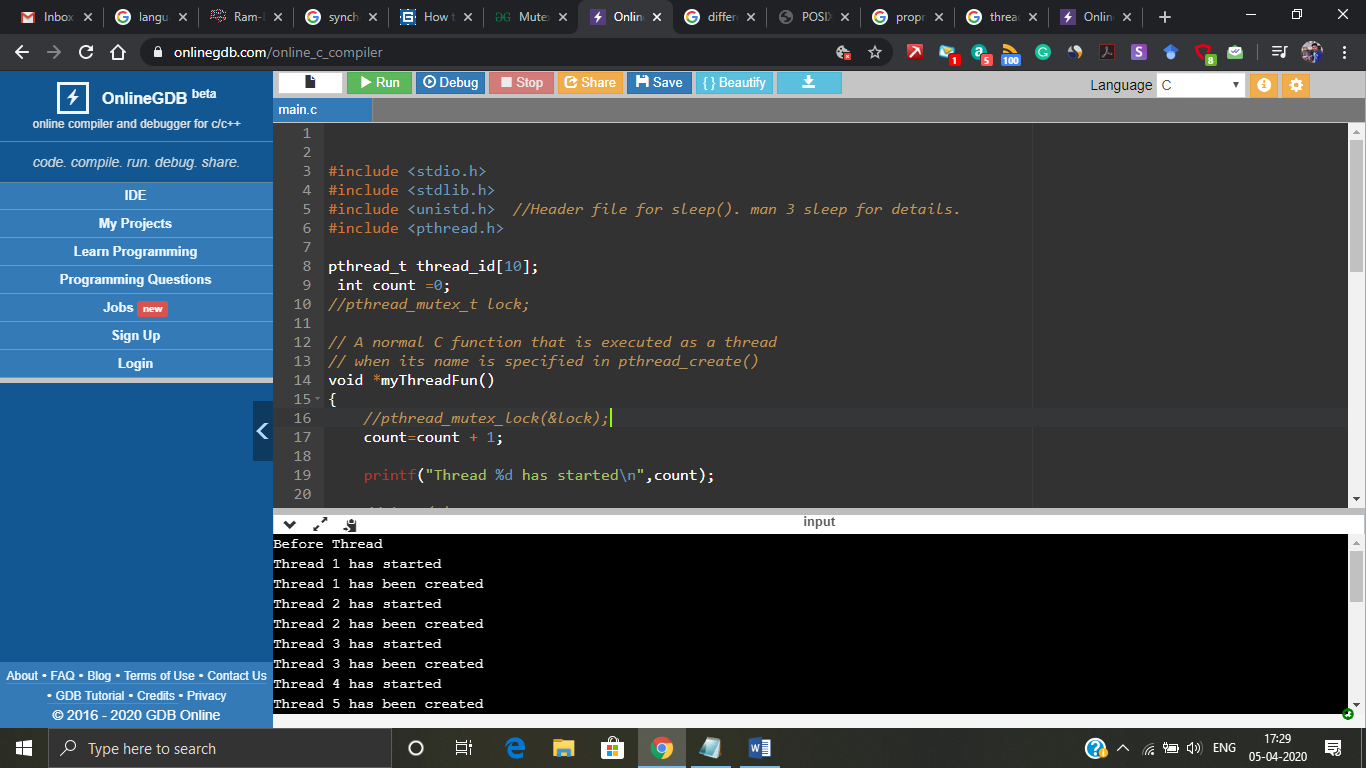


Figure 2: Multiple threads

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.**

**What I was trying to convey here was by just introducing a sleep function into the critical section led all the threads to start at once and use the critical section which caused the problem. This problem is called race condition. Since there are multiple threads trying to access the critical section at the same time improper execution and results can be met which can be seen in figure 3.(The output is getting mixed up with other threads output)**

#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;

}

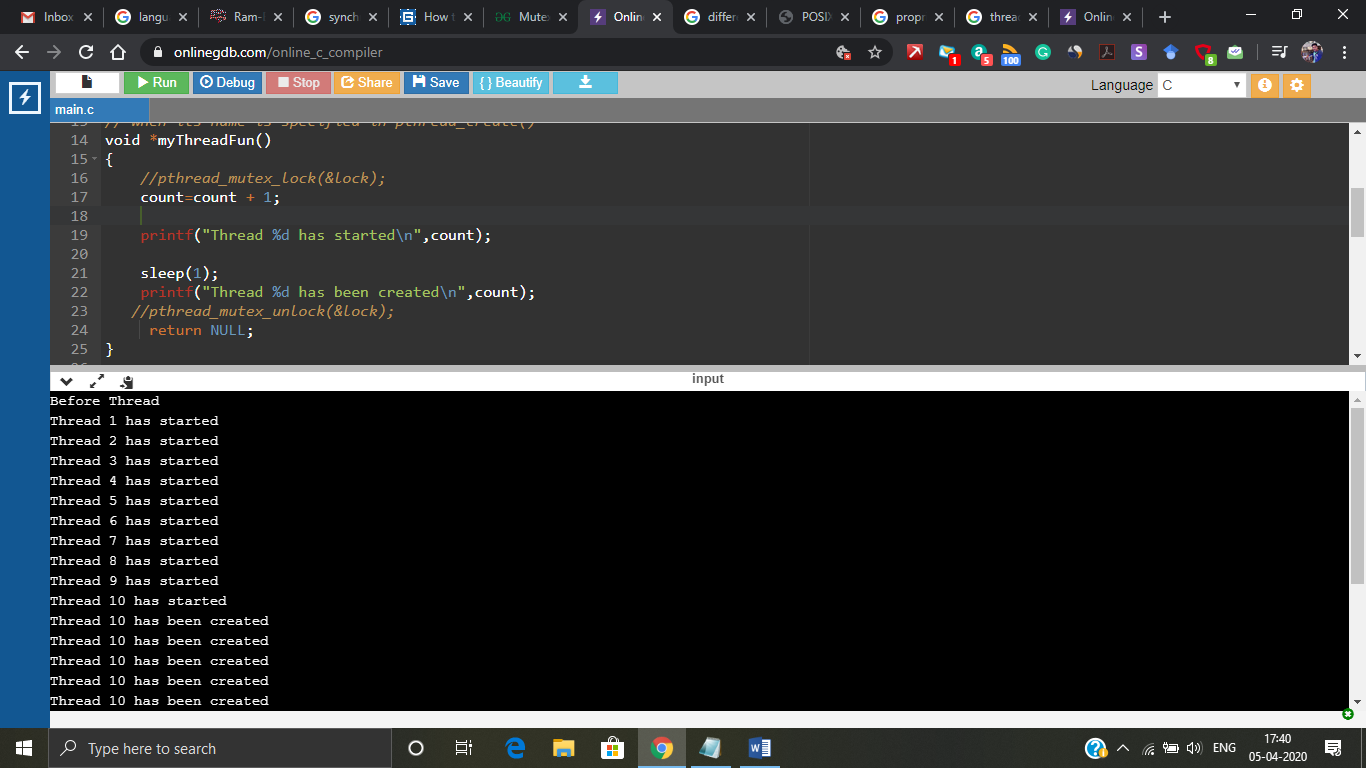


Figure 3: Race condition

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:**

**What is synchronization mechanism?**

A mechanism which has a lock and unlock or wait and release feature that can produce a desired or human understandable output is called as synchronization mechanism. This is achievable by allowing only one thread to access the critical section at a time instance.

**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.

**What is the relation between synchronization and mutual exclusion?**Mutual exclusion deals with at any time there is not more than one process in its critical section. No process is allowed to enter its critical section, when there is already another process in its critical section.

At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. There are many methods to achieve this synchronization so one among that is mutual exclusion.

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. This is also one type of mutual exclusion.

Condition variables- When you want to sleep a thread, condition variable can be used. In C under Linux, there is a function pthread\_cond\_wait() to wait or sleep.  
On the other hand, there is a function pthread\_cond\_signal() to wake up sleeping or waiting thread. Threads can wait on a condition variable. A condition variable essentially is a container of threads that are waiting for a certain condition.Types of Mutual exclusion techniques:

* **Recursive/ Spin lock**: 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.
* **Semaphores**
* **Monitors:** In concurrent programming, a monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become false. Monitors also have a mechanism for signaling other threads that their condition has been met. A monitor consists of a mutex (lock) object and condition variables. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.
* **Message passing:** In computer science, **message passing** is a technique for invoking behavior on a computer. The invoking program sends a message to a process and relies on that process and its supporting infrastructure to select and then run the code it selects. Message passing differs from conventional programming where a process, subroutine, or function is directly invoked by name. Message passing is key to some models of concurrency and object-oriented programming.
* **Tuple space:** A tuple space is an implementation of the associative memory paradigm for parallel/distributed computing. It provides a repository of tuples that can be accessed concurrently.Tuple space may be thought as a form of distributed shared memory.

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. The kernel decides which thread will execute next. So, the order in which the thread executes is unpredictable each and every time when we compile and try to run, the thread may execute in different order.

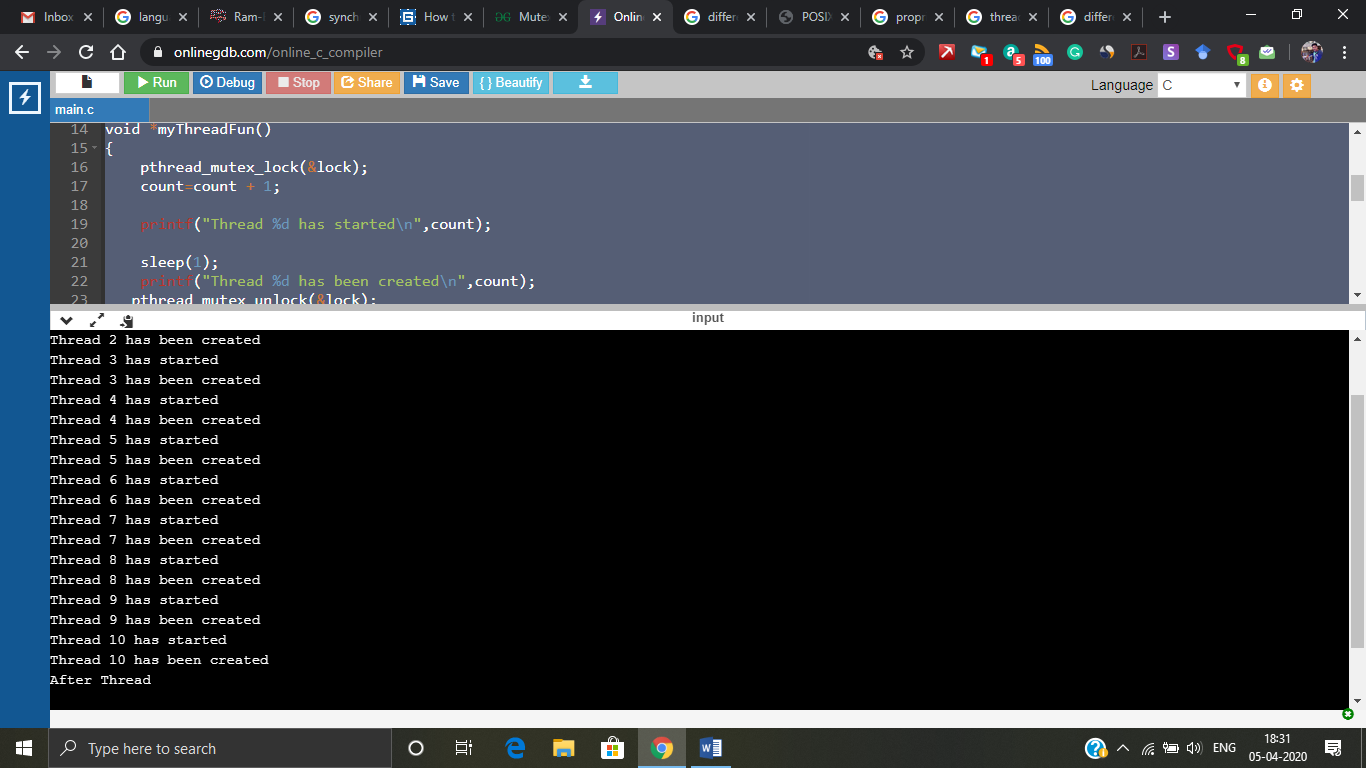


Figure 4: Mutex synchronization

This race condition in the above figure 3 can be solved using semaphore synchronization mechanism too. As told before two or more threads are trying to use or enter into critical section that is the reason for race condition.

#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;

}

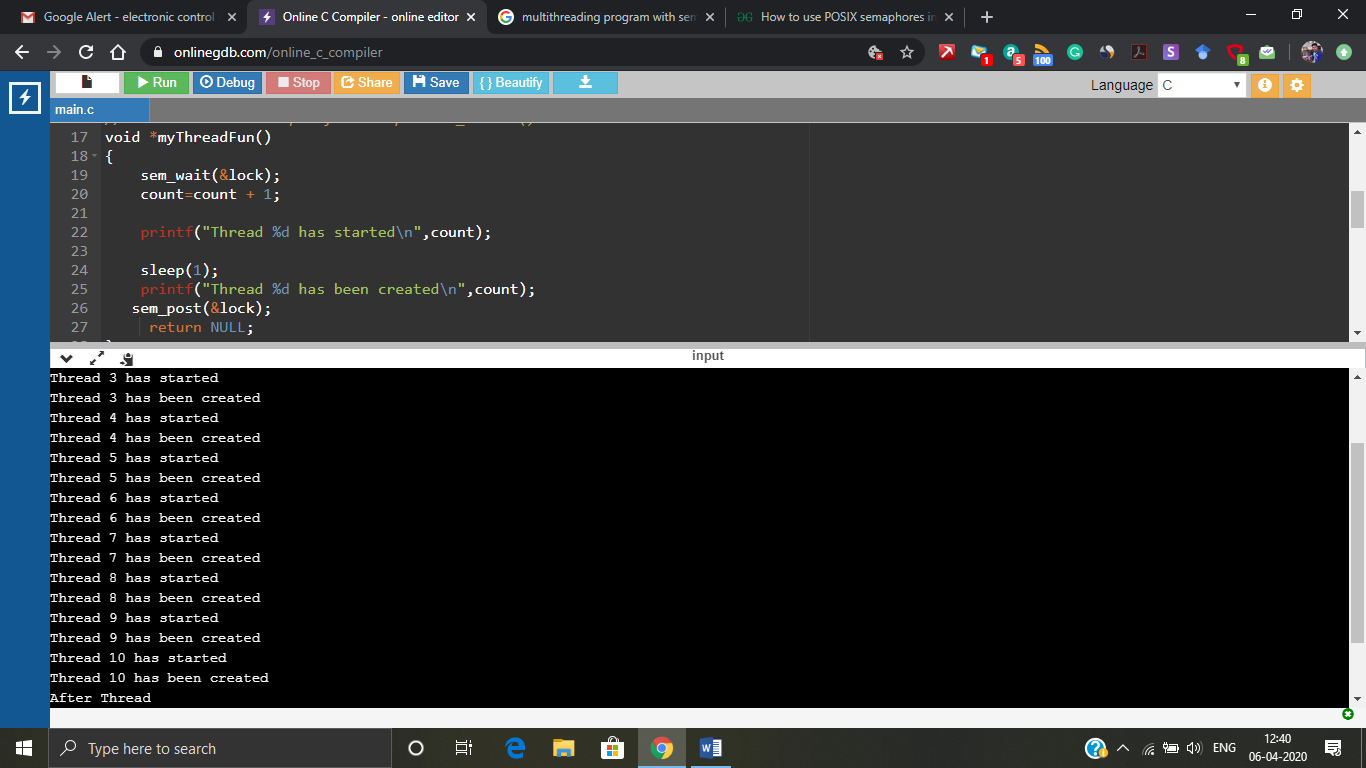


Figure 5: Semaphore mechanism

Figure 5 shows that the race condition has been overcome using semaphore mechanism. The mutex has lock and unlock mechanism where as semaphore has wait and signal strategy. Initially semaphore value is 1 before it enters to wait it decrements and becomes 0 and after the thread completes execution it again increments and returns to 1. When value is 0 means it’s in waiting state and thread is one thread already occupied. When the value is 1 it has released and next thread can access.

Conditional variable synchronization doesn’t suit the race condition which was showcased in figure 3. So, there is an different scenario shown below.

#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;

}

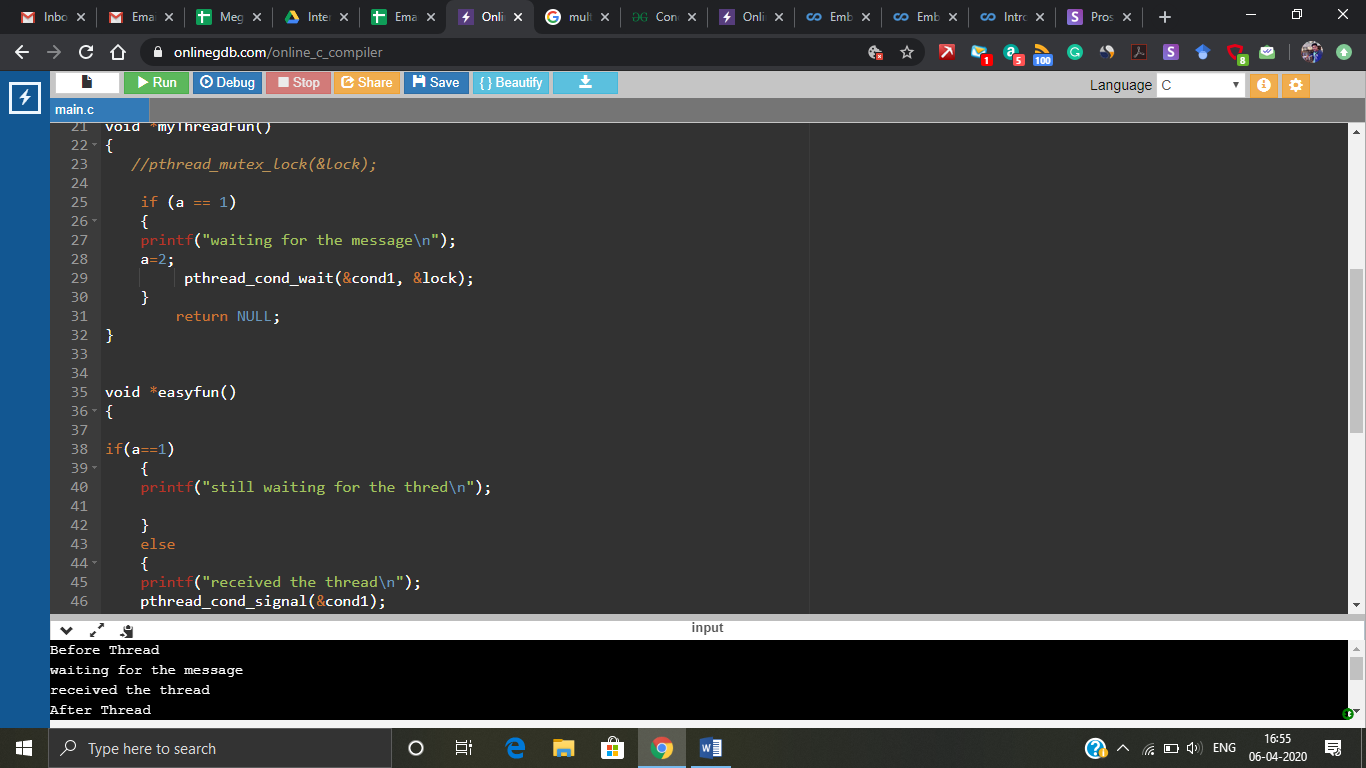


Figure 6: conditional variable mechanism

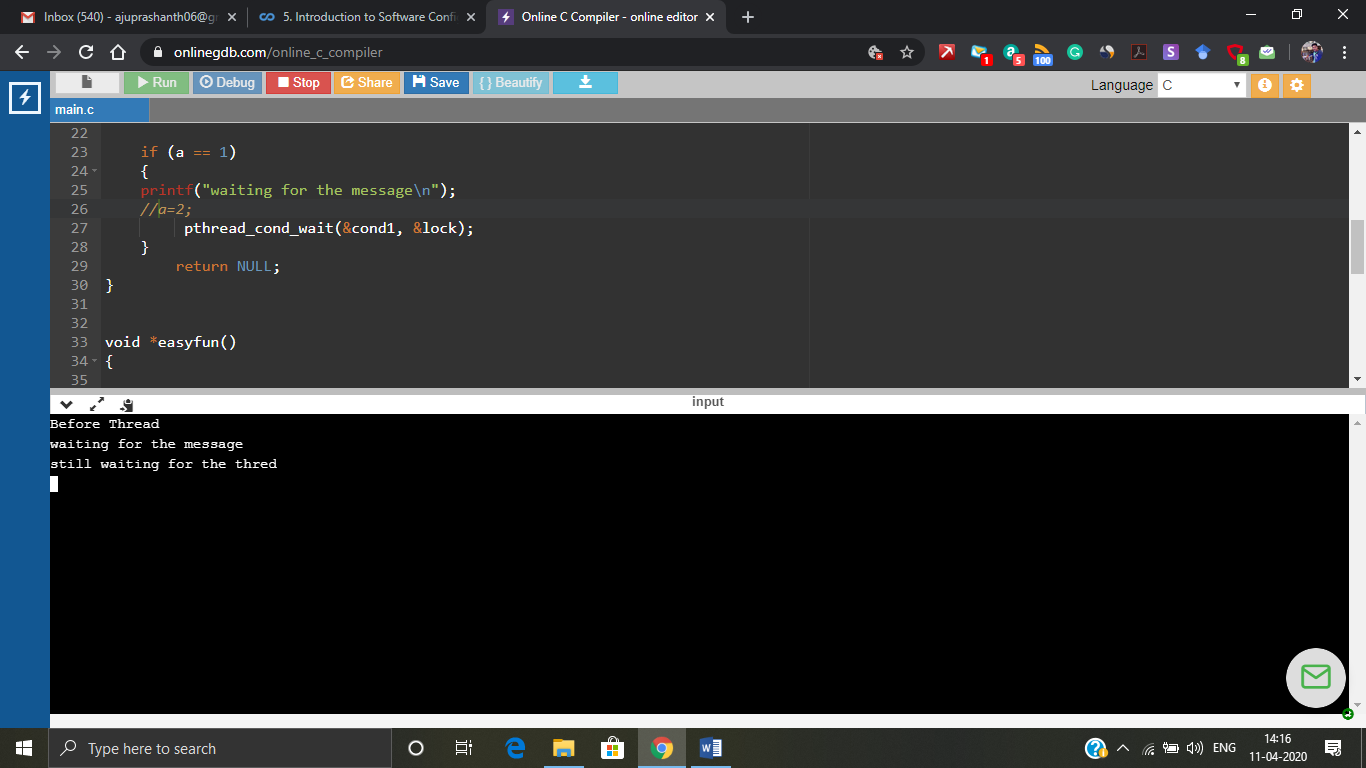
Conditional variable is kind of wait and receive mechanism. In the above example the thread mythreadfun is made to wait by giving pthread\_cond\_wait function and this can be wake up in another thread too. So in mythreadfun thread it is made to wait and the value of a is changed to 2 so in another thread easyfun it’s waken up and it checks for a value its 2 so it prints the else part of it. If the a value is not changed it doesn’t stop at all it will be keep on executing.

Figure 7: waiting for result

# **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();

}

}

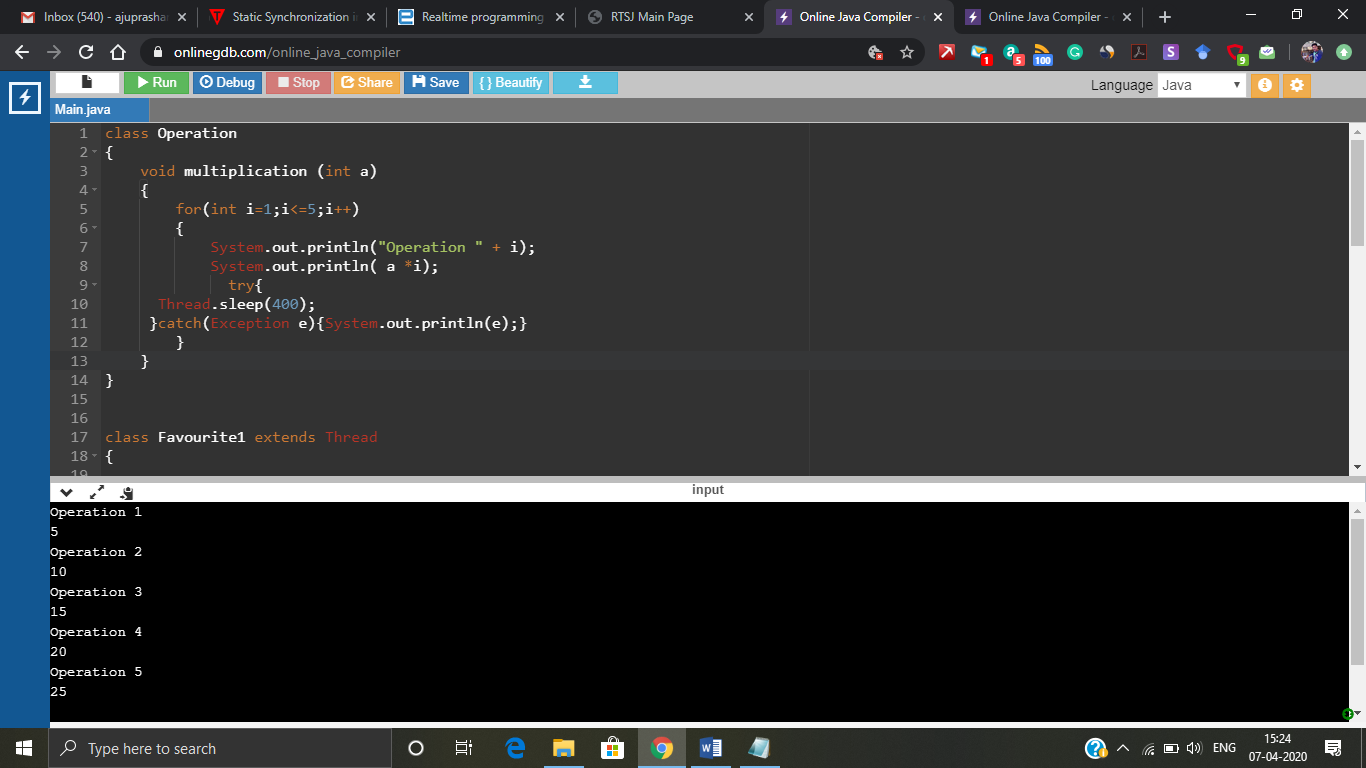


Figure 8: Thread creation

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();

}

}

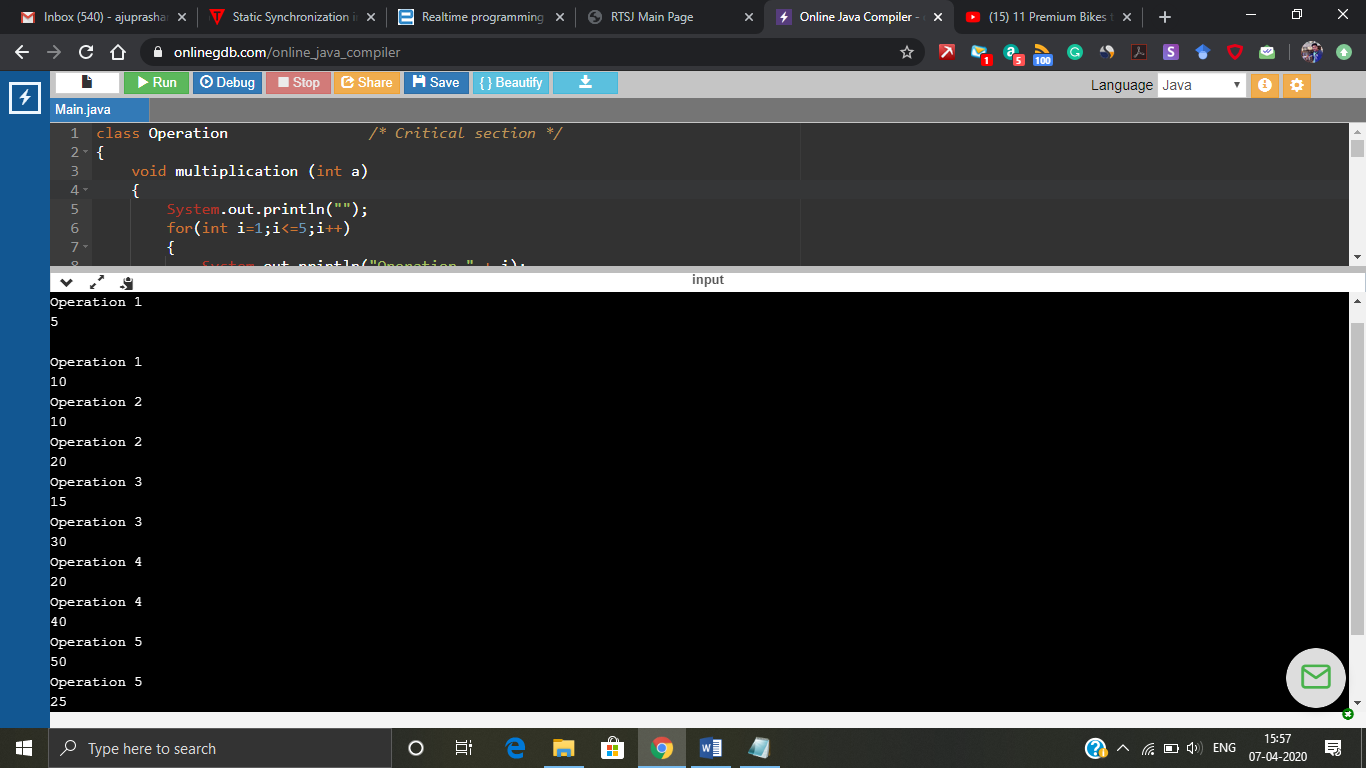


Figure 9: Multiple threads output

The output got in figure 8 is not ordered properly we can see that two threads are created favourite1 and favourite2 they both try to execute the critical section at the same time so the output got is clumsy and when two threads trying to access the same variables memory inconsistency occurs. (The outputis getting mixed up with other threads that’s the simple reason) So, this can be solved using some synchronization mechanism which allows one thread at a time to access the critical section.

**Synchronization mechanisms:**

* **Mutual exclusion (under this we have synchronization method, synchronization block, static synchronization).**
* **Cooperation(under this we have inter thread synchronization).**
* **These are different synchronization mechanisms that JAVA has,Synchronization method:** If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.**Synchronization block:** Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block. If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.**Static synchronization**: If you make any static method as synchronized, the lock will be on the class not on object. If you want to acquire the lock in class level we use this synchronization.**Inter thread synchronization-** Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other. Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of Object class: wait(), notify() and notifyal().
* Cyclic Barrier Synchronization: CyclicBarriers are used in programs in which we have a fixed number of threads that must wait for each other to reach a common point before continuing execution.

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. This makes the whole method to be synchronized and allows one thread at a time into the critical section.

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. The favourite2 (i.e the thread 2) takes the control and executes first then the first thread favourite 1 does the execution. This execution is unpredictable it can be either way (i.e) thread1 first and thread2 second or thread2 first and thread1 second. Since the kernel schedules this so, it’s unpredictable.

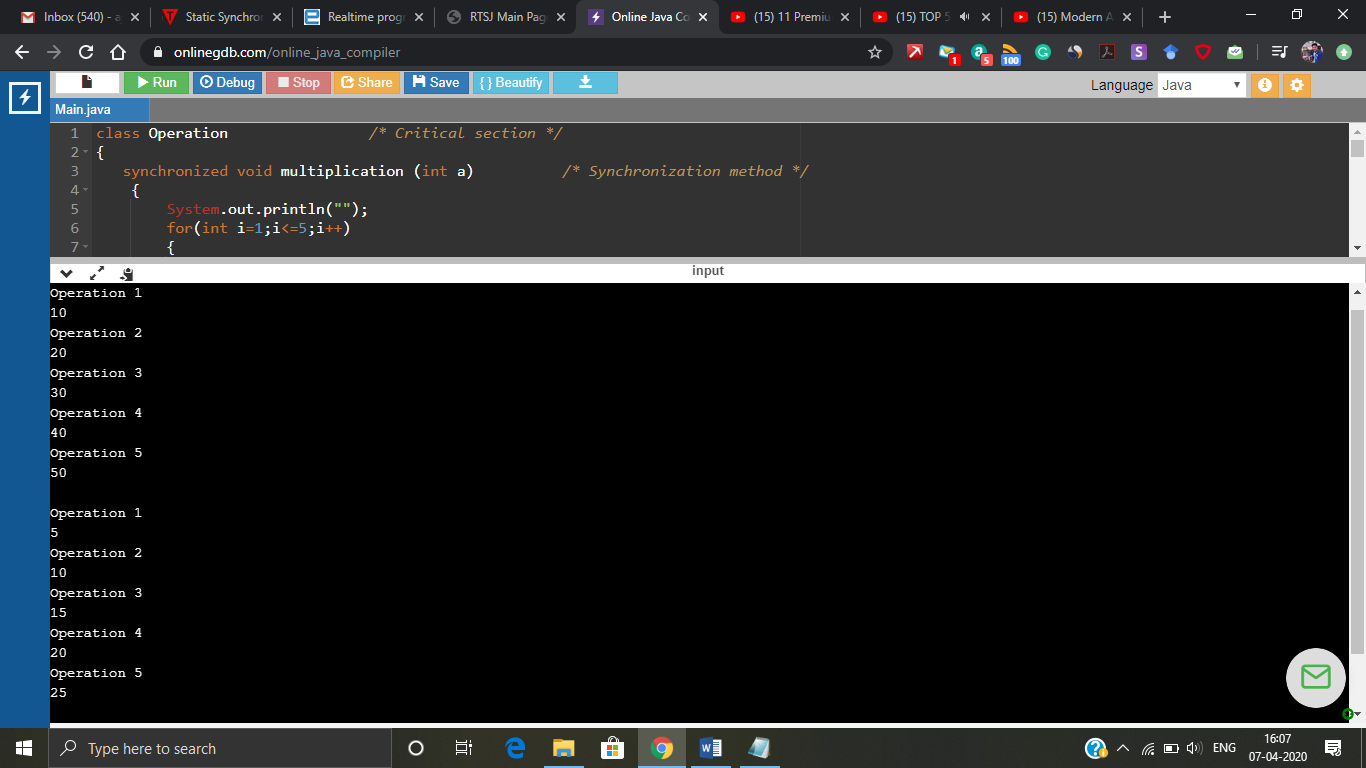


Figure 10: Synchronization method

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();

}

}

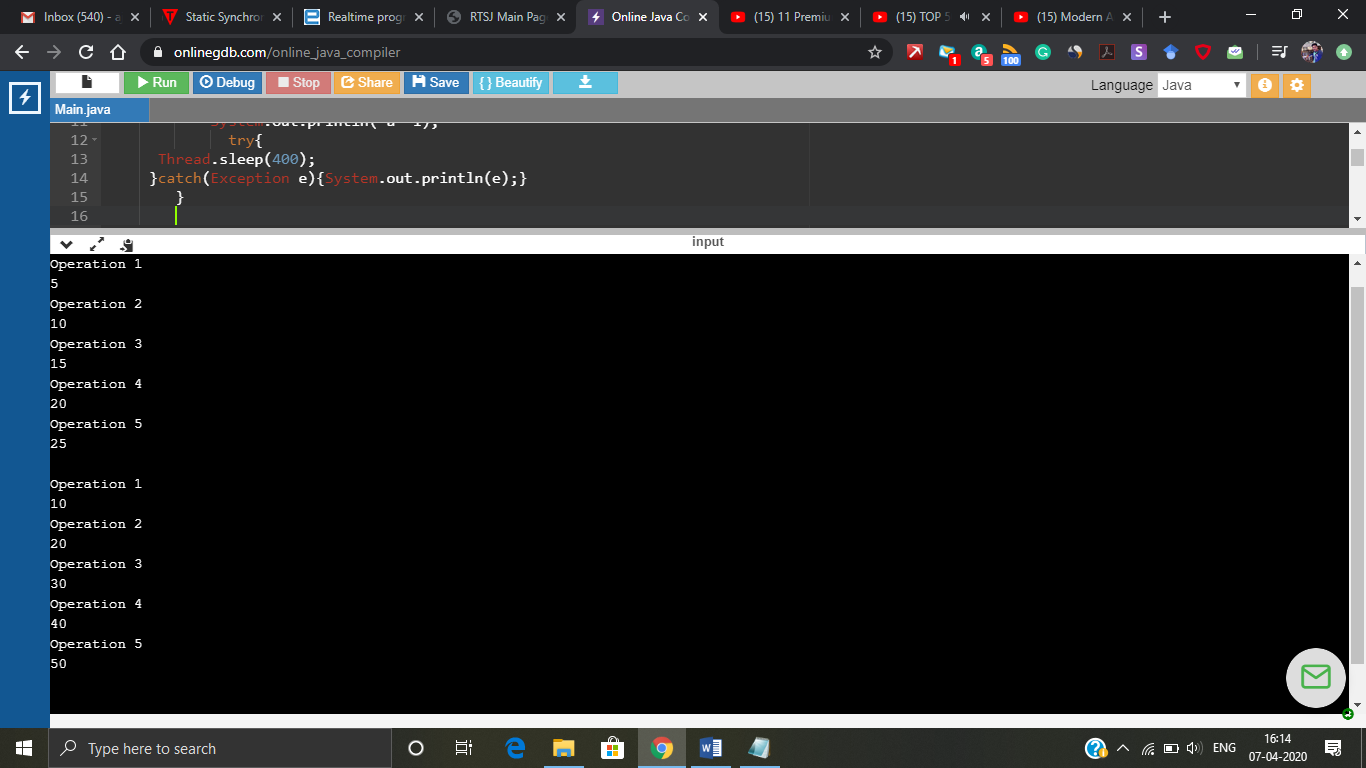


Figure 11: Synchronization block

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.Synchronization method applies synchronization to the method, Synchronization block applies synchronization to the specific block inside the method whereas static synchronization applies synchronization to the whole class. So, the area covered is wide.

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();

}

}

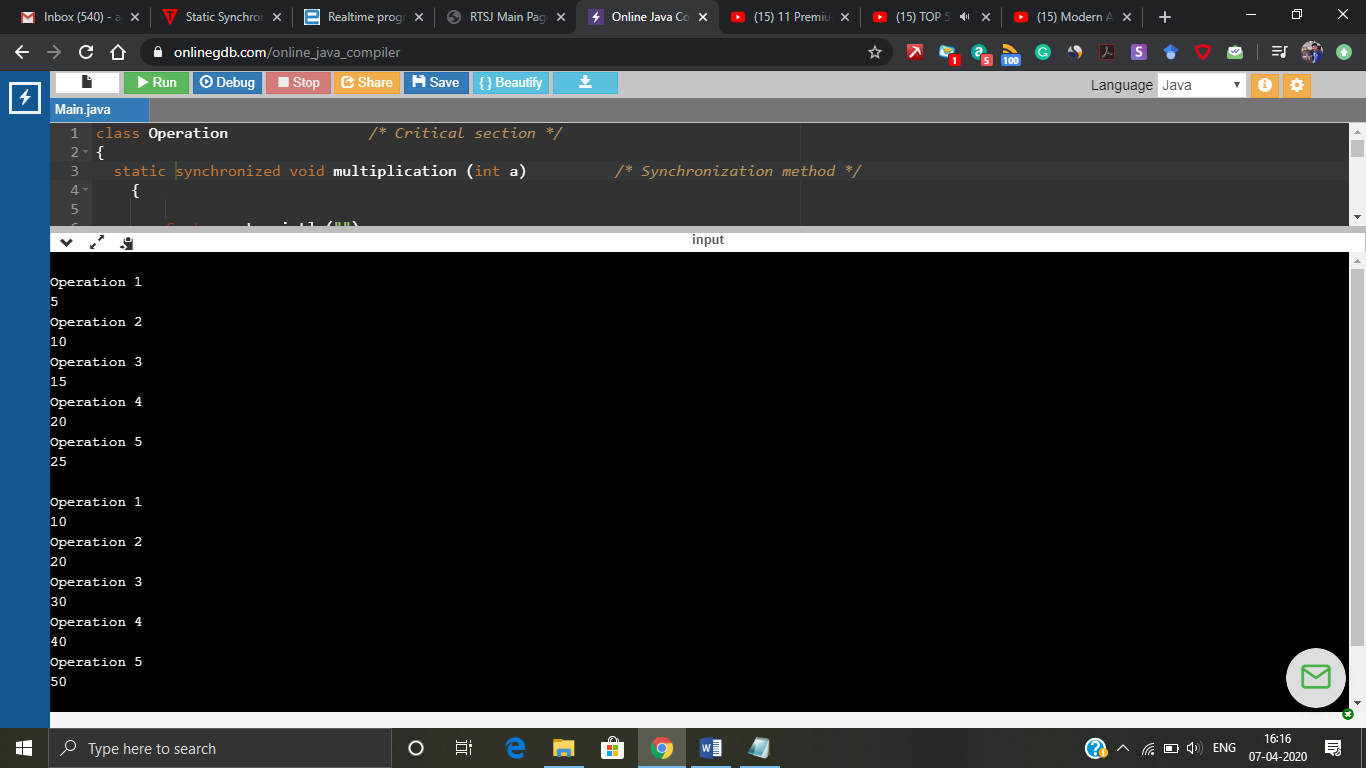


Figure 12: Static synchronization.

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.. 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();

}

}

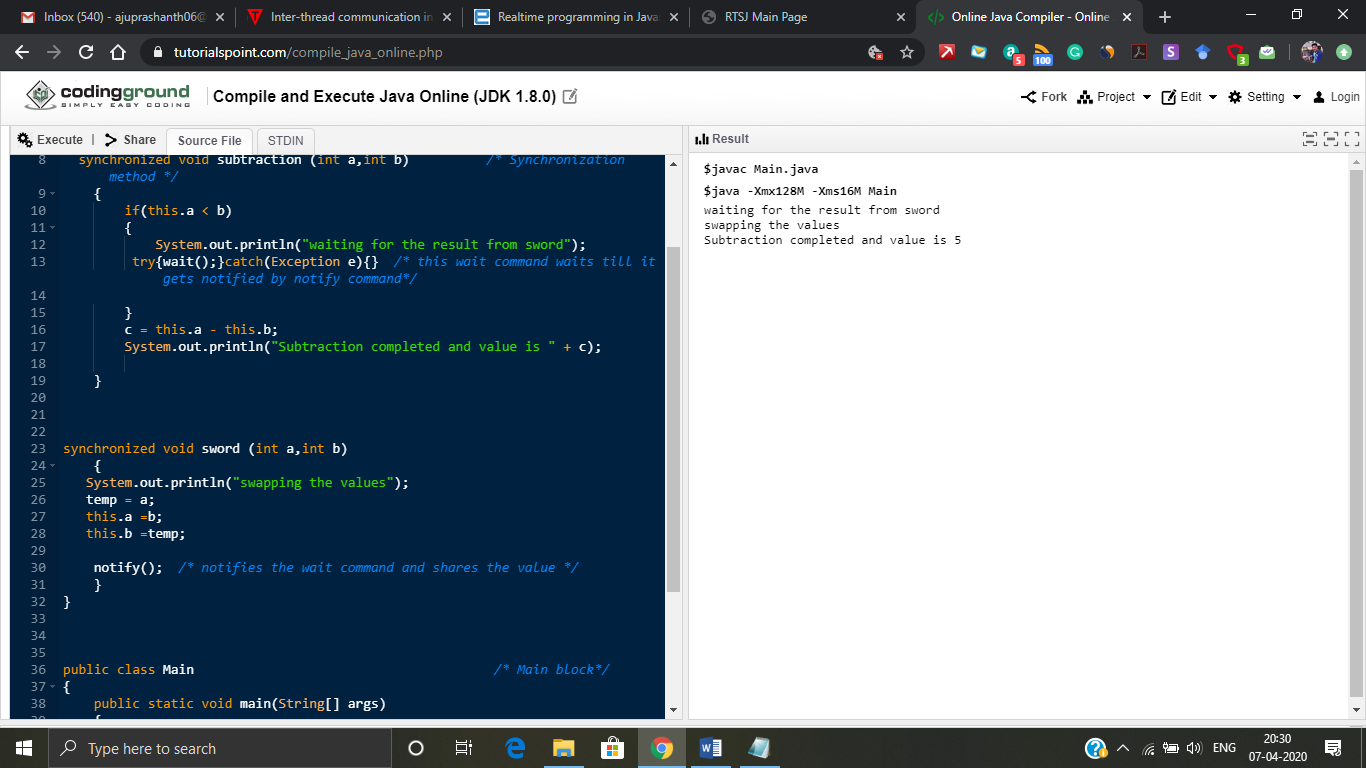


Figure 13: Inter thread synchronization

**Wait()** Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. **Notify()** wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. **notifyAll()**wakes up all threads that are waiting on this object's monitor.

Ths below example shows that the wait and notify can be used in different class too.

class Reader extends Thread {

Calculator c;

public Reader(Calculator calc) {

c = calc;

}

public void run() {

synchronized(c) { //line 9

try {

System.out.println("Waiting for calculation...");

c.wait();

} catch (InterruptedException e) {}

System.out.println("Total is: " + c.total);

}

}

}

class Calculator extends Thread {

int total=0;

public void run() {

synchronized(this) { //Line 31

for(int i=0;i<50;i++) {

total += i;

}

notifyAll();

}

}

}

public class Main /\* Main block\*/

{

public static void main(String [] args) {

Calculator calculator = new Calculator();

new Reader(calculator).start();

calculator.start();

}

}

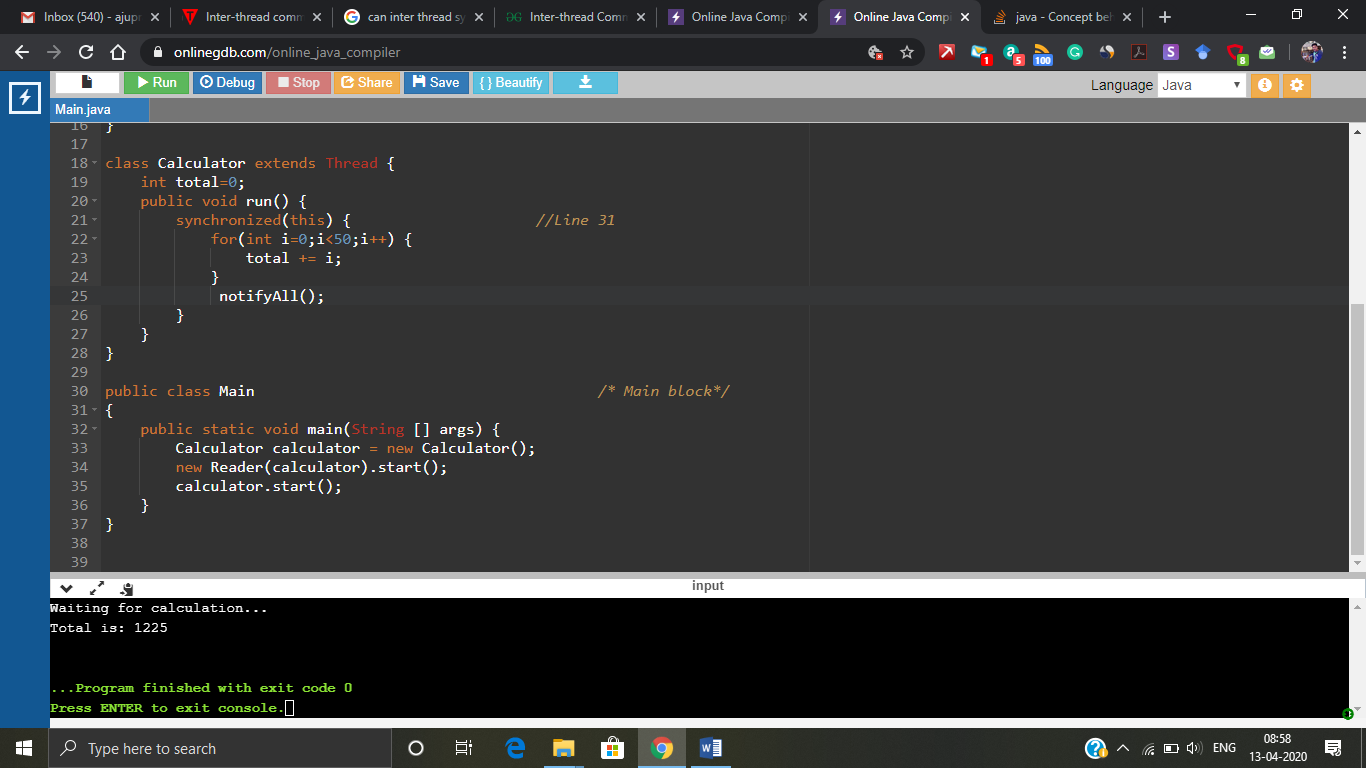


Figure 14: Usage in different class.

**Barrier synchronization:**

The below program demonstrates the cyclic barrier synchronization. This program is based on our shared cab when 2 person gets inside the car the cab starts.

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

*/\* The above 2 packages are to be used for achieving Cyclic barrier synchronization\**/

class PassengerThread extends Thread {

private int duration;

private CyclicBarrier barrier;

public PassengerThread(int duration, CyclicBarrier barrier, String pname) {

super(pname);

this.duration = duration;

this.barrier = barrier;

}

*/\* This method has 3 arguments passenger name, duration of sleep and synchronization method \**/

@Override

public void run() {

try {

Thread.sleep(duration);

System.out.println(Thread.currentThread().getName() + " is arrived.");

int await = barrier.await();

/\* *This barrier.await method makes all threads to meat at one point and produces the result \**/

if(await == 0) {

System.out.println("two passengers have arrived so cab is going to start..");

}

} catch (InterruptedException | BrokenBarrierException e) {

e.printStackTrace();

}

}

}

/\*\*

\* \* Java Program to demonstrate how to use CyclicBarrier, Its used when number

\* of threads \* needs to wait for each other before starting again.

\*/

public class ClientTest {

public static void main(String args[]) throws InterruptedException, BrokenBarrierException {

System.out.println(Thread.currentThread().getName() + " has started");

CyclicBarrier barrier = new CyclicBarrier(2);

*/\* The number 2 indicates number of threads that are to be met \**/

PassengerThread p1 = new PassengerThread(1000, barrier, "John");

PassengerThread p2 = new PassengerThread(2000, barrier, "Martin");

PassengerThread p3 = new PassengerThread(1000, barrier, "Pipa");

PassengerThread p4 = new PassengerThread(2000, barrier, "Dolly");

p1.start();

p2.start();

p3.start();

p4.start();

System.out.println(Thread.currentThread().getName() + " has finished");

}

}

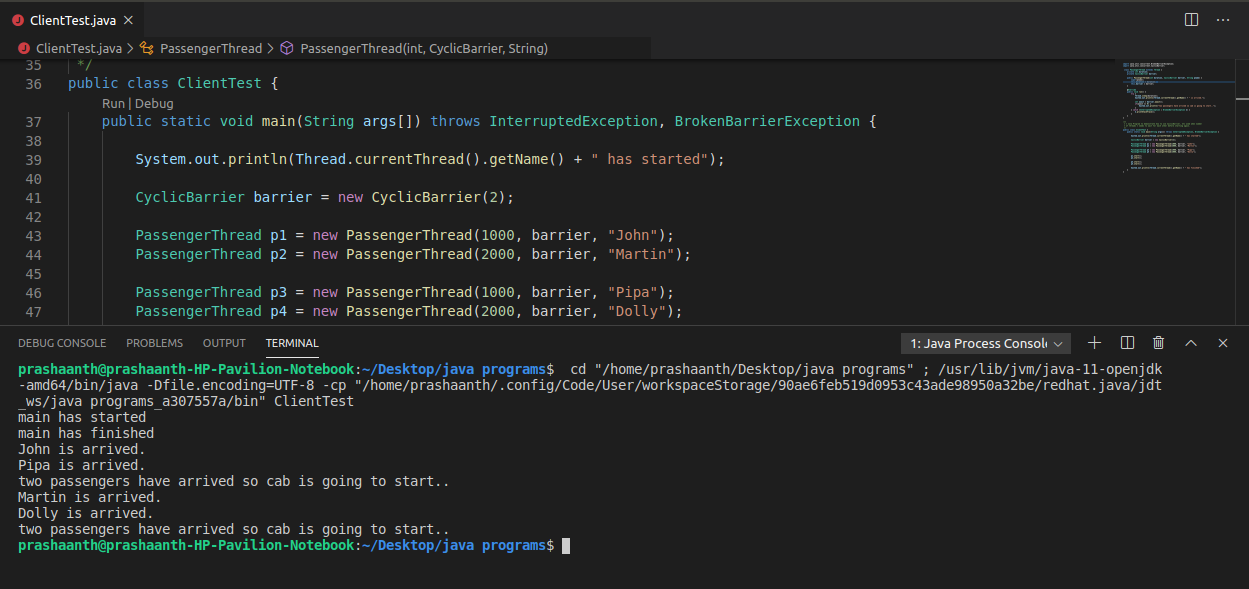
****

Figure 15: Cyclic Barrier synchronization.

Figure 15 shows the cyclic barrier synchronization output where the first thread p1 goes to the PassengerThread class and passenger name is printed and waited for particular amount of time with the given duration variable which comes from ClientTest class. Again p2 thread goes and same thing happens and finally since the cyclic barrier number of threads is 2 the await condition gets satisfied and prints the output.

**Monitors:**

Monitors provides 3 capabilities to concurrent programming

* Only one thread at a time can have access to the critical section.
* Threads running in a monitor can lock awaiting certain conditions to be true.
* A thread can notify one or more threads that conditions they’re waiting on have been met.

In the Java virtual machine, every object and class is logically associated with a monitor. To claim a monitor region which means data not accessible by more than one thread, Java provide synchronized statements and synchronized methods. Once the code is embedded with synchronized keyword, it is a monitor region. The key idea with the concept of monitors is to combine object oriented programming with mutual exclusion.

**Montors (wait/notify), barrier synchronization are not discussed here. more**

**Focus on some reaserach questions such as**

* **Above discussed mechansisms are enough to do real time programming? Otherwise What are the changes needed for C and Java language to include real time capabilities?**
* **Do we need platform support for real time capabilities for both C and Java**
* **Existing packages available to support real time functionalites in both C nad Java**
* **What are the real time applications that support concurrency?**

The Real-time system is one that must perform operations within rigid timing constraints. The correctness of real time system does not depend only on the logical correctness but also on the time it takes to produce the result.

**Thread scheduling in JAVA:**

* SCHED\_OTHER = 40 priorities, for normal java
* SCHED\_FIFO = 99 priorities, for RT java
* SCHED\_RR = For RT java

**Above discussed mechanisms are enough to do real time programming? Otherwise What are the changes needed for C and Java language to include real time capabilities?**

|  |  |
| --- | --- |
| **Java** | **C** |
| * On top of Monitors, wait() and notify() java also allows other basic synchronization techniques such as semaphores and mutex which can also be used. But the above discussed monitor is more than enough for non-real time java application. * But for achieving real time capabilities javax.realtime.RealtimeThread  can be added and this is later discussed more in detail. This javax.realtime package uses pthread and mutex synchronization algorithm. In JAVA priority inheritance and priority inversion can be solved using native threads by using pthread\_mutex lock services. | * The discussed synchronization mechanism are more than enough for satisfying real time capabilities. * Based on the platform we use the header files are included. Eg: for xenomai we include <native/task.h> |

**Do we need platform support for real time capabilities for both C and Java:**

|  |  |
| --- | --- |
| **Java:** | **C:** |
| * IBM's WebSphere Real Time * Reference implementation from Timesys * Sun Microsystems's Java SE Real-Time Systems are mostly appropriate for soft real time system. Using Java SE for hard real-time development is also possible, but generally requires the use of more specialized techniques such as the use of NoHeapRealtimeThread abstractions, as described in the Real-Time Specification for Java (JSR 1), or the use of the somewhat simpler ManagedSchedulable abstractions of the Safety Critical Java Technology specification. * PTC Perc from PTC * [JamaicaVM](https://en.wikipedia.org/wiki/JamaicaVM" \o "JamaicaVM) from [aicas](https://en.wikipedia.org/wiki/Aicas" \o "Aicas) this has preemptable garbage collector. It is optimized for intelligent systems and critical applications and widely used in the automotive, industrial automation, military, medical and financial application. * OVM is a good preference. It is open-sourced, supports most of the RTSJ features and is still an active research project [24, 16] with some documents. There are two most important issues with OVM: first, its JIT compiler is quite simple and incomplete (lack of dynamic class loading); second, OVM is obviously designed for a single processor. * jRate is an extension to the GNU GCJ compiler and a collection of runtime libraries. It implemented most features needed by RTSJ * Jikes RVM is not designed for real time purpose, it is the most completed open-source JVM. Actually the prototype of the Metronome garbage collector is implemented on Jikes RVM * Javolution an open source software RT tool based on RTSJ. Javolution can be either a Pure Java Solution or a Pure Native Solution (C++ mirror), small (less than 400 KBytes jar) and fully produced through maven (Java and C++) | * These OS platforms can be used Vx Works, Xenomai, FREE RTOS * Even RTlinux can be used but it is still under development and not fully developed. |

JAVA SE, IBM WebSphere and Jamaica VM are the 3 promising platforms I think. As far as I researched these three are open source and Websphere runs on RT linux. Jamaica VM can also be used in any Linux platform. Even Javolution I tried to explore but more examples of embedded system development are not available online.

**Hardware improvements for java real time (processors)**

* Komodo.
* SHAP is another Java processor that is designed specifically for real-time systems. It implements rapid context switching and concurrent Garbage Collector (GC).
* JOP is a well-developed Java processor which is WCET analyzable. Method cache in JOP simplifies the evaluation of WCET in control flow, due to the fact most effective thread switching can introduce cache misses. Tools for performing WCET analysis on JOP is provided in [12]. The excessive degree WCET analysis is primarily based on ILP and a low level timing model is provided by JOP properties.

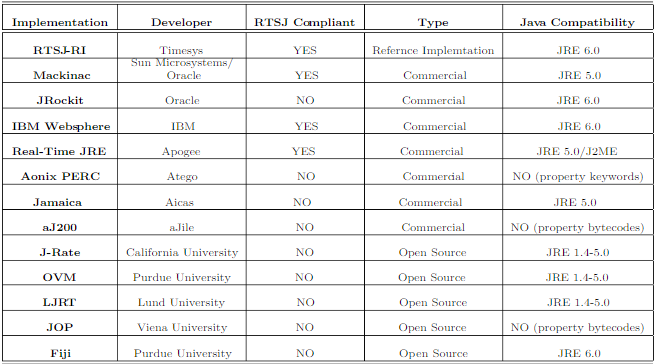


Figure 16: Real time java solutions

**Exisitng packages available to support real time functionalities in both C and Java:**

|  |  |
| --- | --- |
| **Java:** | **C:** |
| * import java.util.concurrent.\* = Package for concurrent programming * import java.util.concurrent.Semaphore = Package for semaphore * LJRT package for real time java (Lund Java Real TIme) semaphore - se.lth.cs.realtime.semaphore.Semaphore se.lth.cs.realtime.semaphore.MutexSem se.lth.cs.realtime.semaphore.CountingSem * In RTSJ there are these packages  1. Javolution package for real time application. (Open source) import static javolution.context.ConcurrentContext 2. import javax.realtime.\*; this is the package for adding the things which are present below this targets embedded JAVA. 3. The degree of temporal guarantees provided to an activity depends on the type of thread in which the activity is executing: java.lang.Thread or javax.realtime.RealtimeThread thread types. Standard java.lang.Thread (JLT) threads are supported for non-real-time activities. JLT threads can use the 10 priority levels specified by the Thread class, but these are not suitable for real-time activities because they provide no guarantees of temporal execution | * Pthread.h library  1. Semaphore library #include <semaphore.h> 2. Xenomai platform provides <native/task.h> that provides an opportunity to add real time capabilities.  * Win32 threads |

**Real Time applications that support concurrency:**

* Microwave Oven
* Washing Machine
* Traffic signal control
* Calculator
* Digital Camera
* AIRBAG
* ATM
* Elevator
* Voice recognition
* Bike instrument cluster
* Smart Tv
* Collission avoidence
* Robot
* Missile Guiding system
* Aircraft navigation system or GPS navigation
* Space rover
* Weapon defense system.
* Traction control system (Automobile).

**Different unpredictable issue in JAVA for real time systems:**

A number of factors may render the timing of execution unpredictable and therefore may cause a standard Java task to miss its deadline. Here are the most common.

* **Operating-system scheduling.** In Java technology, threads are created by the JVM [\*](https://www.oracle.com/technetwork/articles/javase/index-137216.html" \l "jvm_note) but are ultimately scheduled by the operating-system scheduler. In order for the JVM to provide guarantees of temporal latency, that is, the time delay in reaction after an action, the operating system must provide scheduling-latency guarantees as well. Hence, the operating system must offer capabilities such as a high-resolution timer, program-defined low-level interrupts, and a robust priority-based scheduler.
* **Priority inversion.** One hazard in an application in which threads can have different priorities is priority inversion. If a lower-priority thread shares a resource with a higher-priority thread, and if that resource is guarded by a lock, the lower-priority thread may be holding the lock at the moment when the higher-priority thread needs it. In this case, the higher-priority thread is unable to proceed until the lower-priority thread has completed its work -- and this can cause the higher-priority thread to miss its deadline.  
     
  In addition, if some other task with a medium priority that does not depend on the shared resource attempts to run in the interim, it will take precedence over both the low- and the high-priority tasks. The effect of priority inversion is to "downgrade" the priority of the higher-priority thread to that of the lower-priority thread.
* **Class loading, initialization, and compilation.** The Java language specification requires classes to be initialized lazily -- when an application first uses them. Such instantiations might execute user code, creating jitter, a variance in latency, the first time that a class is used. In addition, the specification allows classes to be lazily loaded. Because loading of classes may require going to disk or across the network to find the class definition, referencing a previously unreferenced class can cause an unexpected -- and potentially huge -- delay. In addition, the JVM has some latitude to decide when, if ever, to translate a class from byte code into native code. Typically, a method is compiled only when it is executed frequently enough to warrant the cost of compilation.
* **Garbage collection.** The primary source of unpredictability in Java applications is garbage collection (GC). The GC algorithms that standard JVMs use all involve a stop-the-world pause, in which the application threads are stopped so that the garbage collector can run without interference. Applications with hard response-time requirements cannot tolerate long GC pauses. Despite a large amount of work in recent years on reducing GC pauses, a so-called low-pause collector is still not enough to guarantee predictability and still may require significant tuning and testing.
* **The application.** Another major source of unpredictability is the application itself, including any libraries that it uses. Most applications consist of multiple computational activities that compete equally for CPU resources. Java applications typically do not use thread priorities, partly because the JVM offers such weak guarantees for thread priorities. In a properly provisioned application, enough CPU cycles should be available to go around, but completing a task may sometimes take longer than expected -- which may cause other tasks to have to wait for CPU resources.  
     
  Most developers treat this problem iteratively: They look for the most serious bottleneck, improve it, then find the next bottleneck and improve it, repeating these steps until the application reaches acceptable performance. However, because this development process is itself unpredictable, it can often affect delivery schedules or require significant additional testing.
* **Other activities in the system.** Other high-priority activities that occur in the system -- such as hardware interrupts, other real-time applications, and so forth -- can cause jitter in the application, thereby affecting determinism.

**Real time application development for both C and Java:**

|  |  |
| --- | --- |
| **Java** | **C programming** |
| **support by the programming language examples:** Ada, Java advantages: readability, OS independence, checking of interactions by compiler  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.  **Synchronization:** Synchronized code uses monitors to protect critical code sections from more than one simultaneous access. Even though Java implements mutual extension, it does not prevent unbounded priority inversions, an unacceptable condition for real-time systems  **RTSJ:** The first consortium is the “Real-Time for Java Expert Group,” under the leadership of Sun Microsystems, which has developed the “Real-Time Specification for Java” (RTSJ) 2. The formal scope for the development of this specification is the Java Community Process (JCP), where the RTSJ runs as Java Specification Request (JSR)-000001.   * RTSJ enhances java in this 7 fields,   Thread Scheduling and Dispatching   * Memory Management   Synchronization and Resource Sharing   * Asynchronous Event Handling   Asynchronous Transfer of Control   * Asynchronous Thread Termination * Physical Memory Access   **RTCE, RTDA:** Another 2 java consortium for developing real-time java capabilities but RTSJ is close and efficient. | **support by libraries and the operating system examples:** C/C++ with POSIX advantages: multi-language composition, possibly more efficient, OS standards. |

**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://computing.llnl.gov/tutorials/pthreads/" \l "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>
* <https://software.intel.com/en-us/forums/intel-moderncode-for-parallel-architectures/topic/311081>
* <https://www.slideshare.net/abufayez/pthreads-vs-win32-threads>
* <https://www.programcreek.com/2011/12/monitors-java-synchronization-mechanism/>
* [http://web.cs.iastate.edu/~cs486/Multithreaded%20programming%20in%20C%20and%20JAVA.pdf](http://web.cs.iastate.edu/~cs486/Multithreaded programming in C and JAVA.pdf)
* <https://www.javaworld.com/article/2077413/semaphore.html>
* <http://fileadmin.cs.lth.se/cs/education/EDA040/LJRTmain.pdf>
* <http://javolution.org/apidocs/javolution/util/package-summary.html#package_description>
* <https://www.oracle.com/technical-resources/articles/java/nilsen-realtime-pt1.html> =JAVA SE traction control experiment.
* <https://www.oracle.com/technical-resources/articles/javase/Jsr-1.html>
* <https://www.ibm.com/developerworks/java/library/j-rtj3/> =Websphere demostration
* <https://www.rtsj.org/docs/OSPlatforms.html> = OS requirements for these platforms
* <http://rtjava.blogspot.com/2009/07/real-time-java-vms.html> = Different RTSJ compatible platforms
* <https://www.javaworld.com/article/2077857/open-source-tools-realistically-real-time.html?page=3> = Javolution
* <https://www.ibm.com/developerworks/library/j-devrtj1/index.html> = JAVA SE Real time capabilities
* [https://www.usenix.org/legacy/publications/library/proceedings/jvm01/full\_papers/siebert/siebert\_html/index.html](https://www.usenix.org/legacy/publications/library/proceedings/jvm01/full_papers/siebert/siebert_html/index.html" \t "_blank)
* <https://dl.acm.org/doi/pdf/10.1145/1620405.1620421>
* <https://www.aicas.com/cms/sites/default/files/jtres07_siebert.pdf>
* <https://www-users.cs.york.ac.uk/~burns/papers/c-posix.pdf>